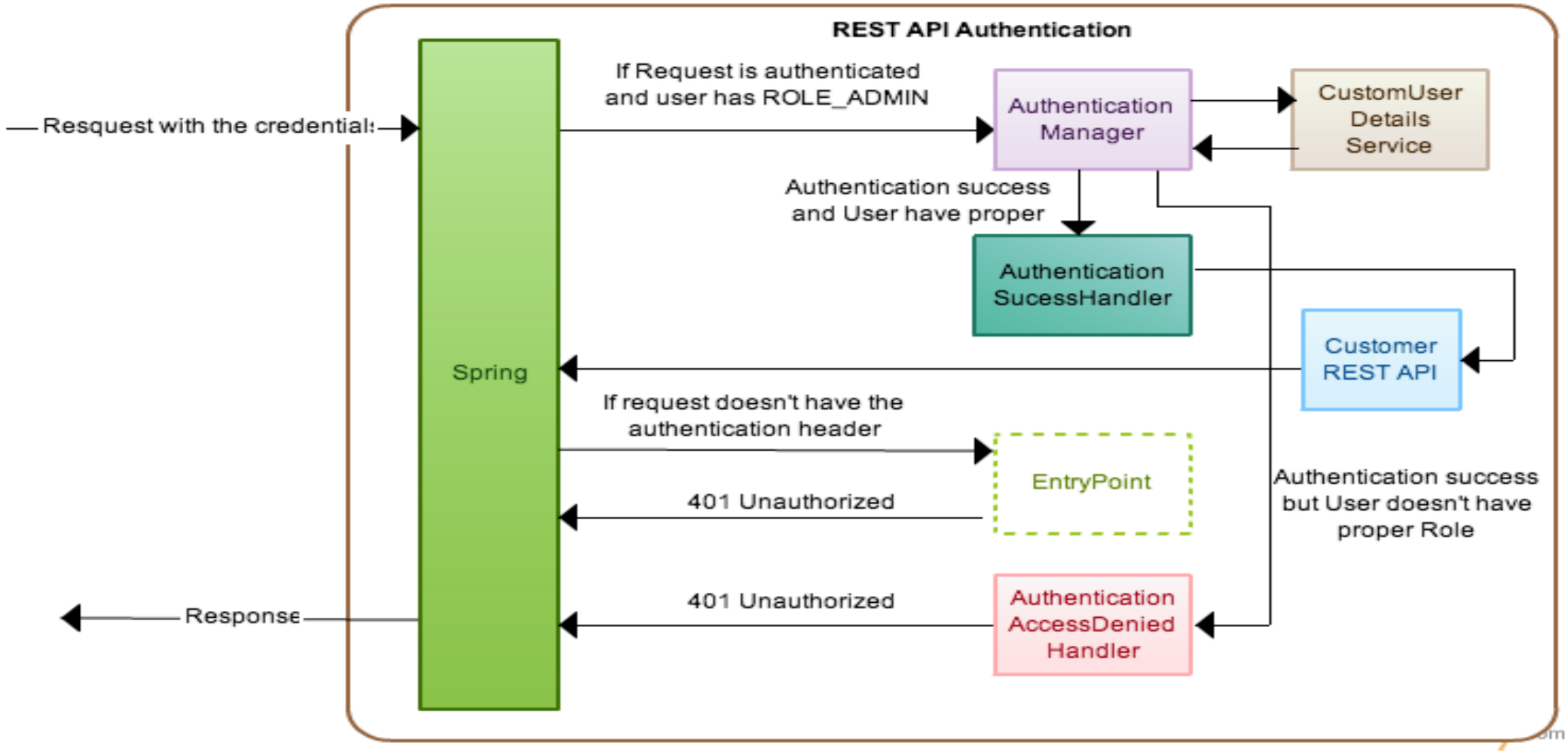
SPRING security

The two dimensions of System Security

- Authentication
  - Identifying the user as an authorized user
  - Limiting the time the user can operate in the system before re-identifying himself
  - Securing that all requests for the duration of the session come from the identified user
- Authorization
  - Constraining the operations the logged in user can perform in accordance with his assigned roles

The SPRING Security Model



In web.xml

```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>
org.springframework.web.filter.DelegatingFilterProxy
</filter-class>
</filter>


<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Java Configuration – Replaces web.xml

```
public class SecurityWebInitializer
        extends AbstractSecurityWebApplicationInitializer
{
// optionally override methods
}
```

Java Configuration – WebSecurityConfig

```
@Configuration
@EnableWebMvcSecurity
public class WebSecurityConfig
        extends WebSecurityConfigurerAdapter
{
...
}
```

Spring Security Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

        @Autowired
        public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
          auth.inMemoryAuthentication().withUser(“test").password("123456").roles("USER");
          auth.inMemoryAuthentication().withUser("admin").password("123456").roles("ADMIN");
          auth.inMemoryAuthentication().withUser("dba").password("123456").roles("DBA");
        }

        @Override
        protected void configure(HttpSecurity http) throws Exception {

          http.authorizeRequests()
                  .antMatchers("/admin/**").access("hasRole('ROLE_ADMIN')")
                  .antMatchers("/dba/**").access("hasRole('ROLE_ADMIN') or hasRole('ROLE_DBA')")
                  .and().formLogin();

        }
}
```

The equivalent of the Spring Security xml file :

```
<http auto-config="true">
        <intercept-url pattern="/admin**" access="ROLE_ADMIN" />
        <intercept-url pattern="/dba**" access="ROLE_ADMIN,ROLE_DBA" />
        <form-login />
</http>

<authentication-manager>
  <authentication-provider>
    <user-service>
        <user name=“test" password="123456" authorities="ROLE_USER" />
        <user name="admin" password="123456" authorities="ROLE_ADMIN" />
        <user name="dba" password="123456" authorities="ROLE_DBA" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

```
protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                        .anyRequest().authenticated()
                        .and()
                .formLogin();
                        }
```

The default configuration above:

- ❑ Ensures that any request to our application requires the user to be authenticated
- ❑ Allows users to authenticate with form based login

It is similar the XML Namespace configuration:

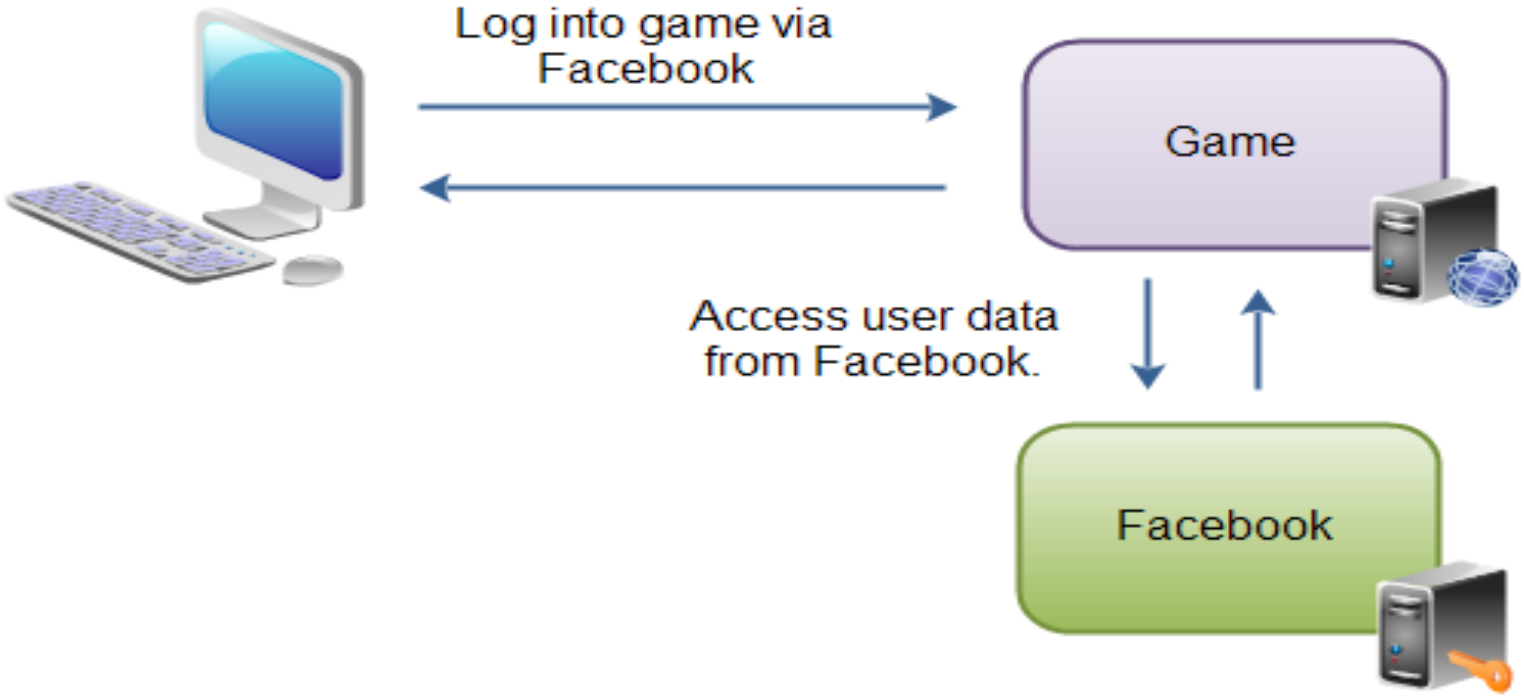
```
<http>
        <intercept-url pattern="/**" access="authenticated"/>
        <form-login />
</http>
```

OAUTH 2.0 OVERVIEW

OAuth 2.0 is an open authorization protocol specification defined by IETF OAuth WG (Working Group) which enables applications to access each other’s data.

The prime focus of this protocol is to define a standard where an application, say gaming site, can access the user’s data maintained by another application like facebook, google or other resource server.

OAuth 2.0 is a replacement for OAuth 1.0, which was more complicated. OAuth 1.0 involved certificates etc. OAuth 2.0 is more simple. It requires no certificates at all, just SSL / TLS.



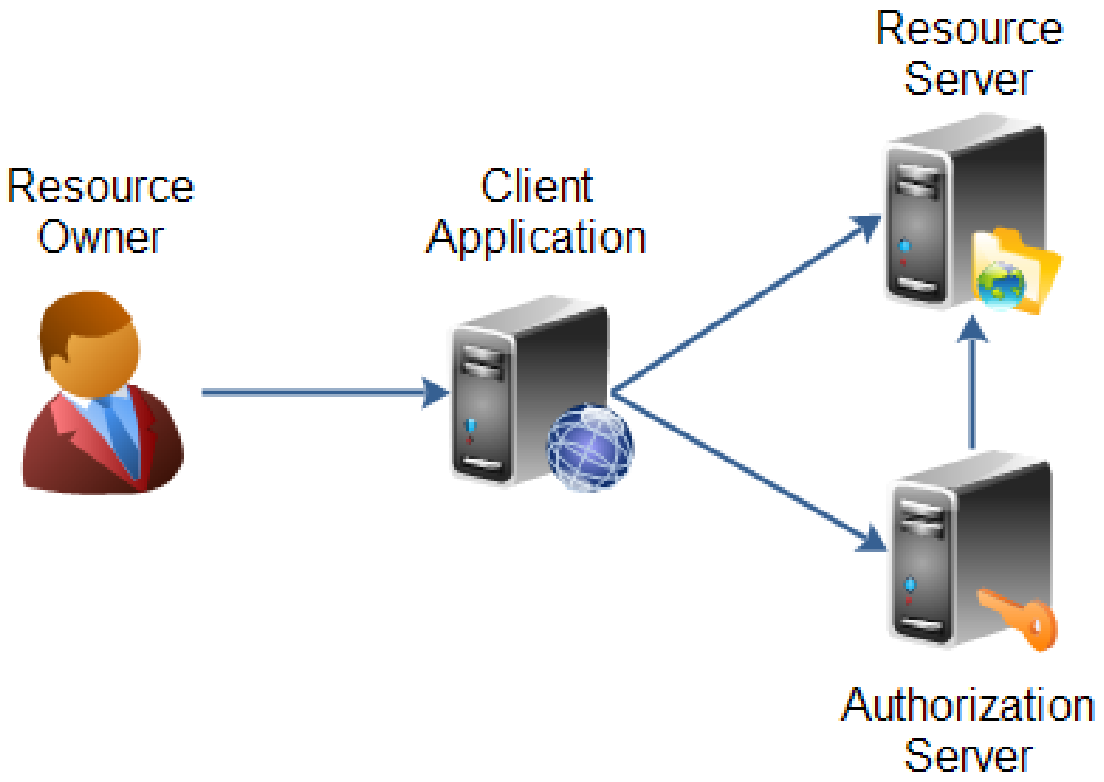
The user accesses the game web application.

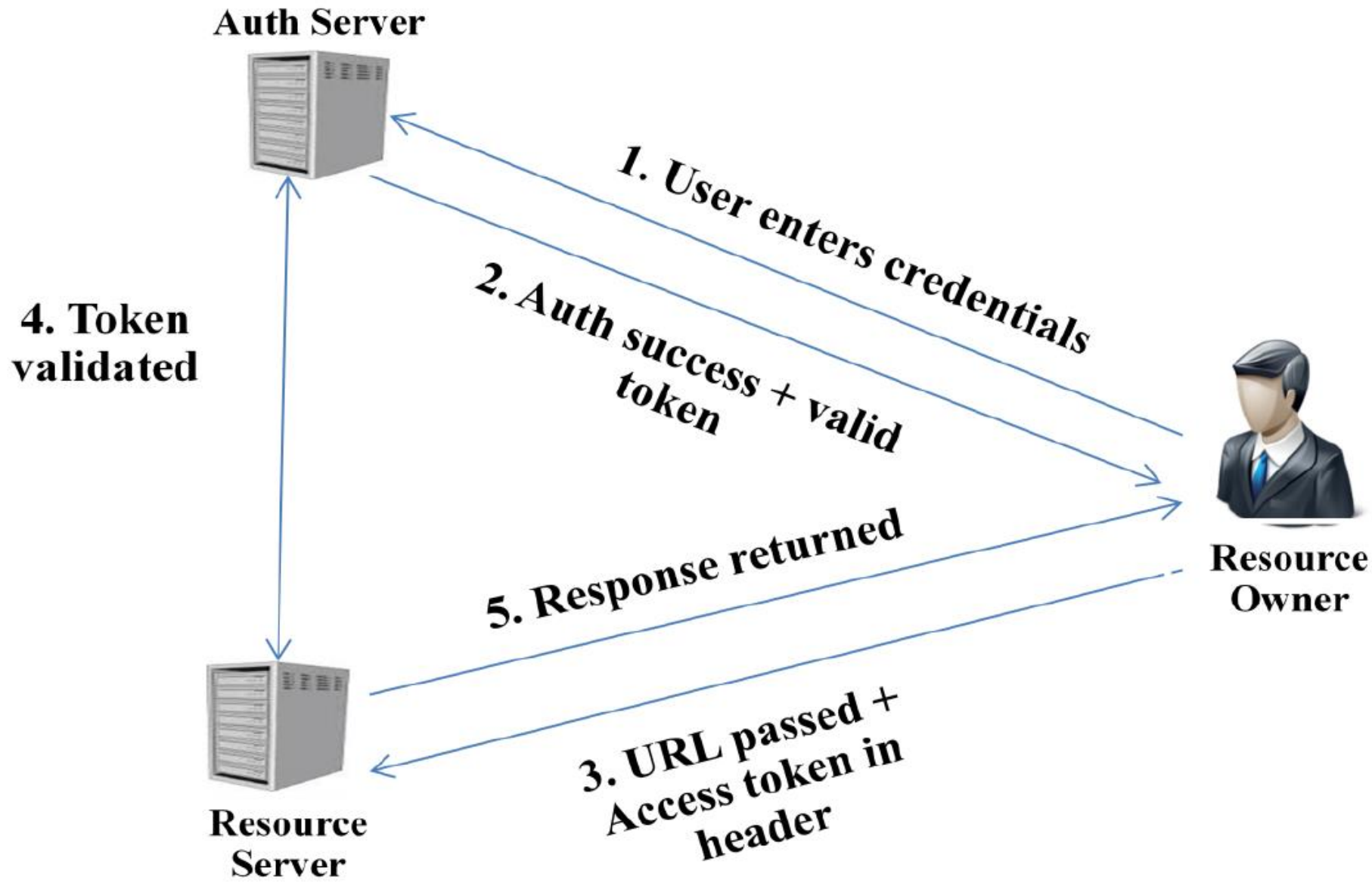
The game web application asks the user to login to the game via Facebook.

The user logs into Facebook, and is sent back to the game. The game can now access the users data in Facebook, and call functions in Facebook on behalf of the user (e.g. posting status updates).

OAuth 2.0 defines the following roles of users and applications:

- Resource Owner
- Resource Server
- Client Application
- Authorization Server


Resource
Server
Resource
Owner
Client
Application
Authorization
Server


Auth Server
1. User enters credentials
2. Auth success + valid token
4. Token validated
5. Response returned
3. URL passed + Access token in header
Resource Owner
Resource Server

- The resource owner is the person or application that owns the data that is to be shared.

  For instance, a user on Facebook or Google could be a resource owner. The resource they own is their data. The resource owner could also be an application. The OAuth 2.0 specification mentions both possibilities.

- The resource server is the server hosting the resources. For instance, Facebook or Google is a resource server (or has a resource server).

- The client application is the application requesting access to the resources stored on the resource server. The resources, which are owned by the resource owner. A client application could be a game requesting access to a users Facebook account.

- The authorization server is the server authorizing the client app to access the resources of the resource owner. The authorization server and the resource server can be the same server, but it doesn't have to.

Steps involved in User Authentication

1. User enters credentials which are passed over to Authorization Server in Http Authentication header in encrypted form. The communication channel is secured with SSL.

2. Authorization server authenticates the user with the credentials passed and generates a token for limited time and finally returns it in response.

3. The client application calls API to resource server, passing the token in http header or as a query string.

4. Resource server extracts the token and authorizes it with Authorization server.

5. Once the authorization is successful, a valid response is sent to the caller.

Client ID, Client Secret and Redirect URI

Before a client application can request access to resources on a resource server, the client application must first register with the authorization server associated with the resource server.

The registration is typically a one-time task. Once registered, the registration remains valid, unless the client app registration is revoked.

At registration the client application is assigned a client ID and a client secret (password) by the authorization server. The client ID and secret is unique to the client application on that authorization server.

Authorization Grant

The authorization grant is given to a client application by the resource owner, in cooperation with the authorization server associated with the resource server.

The OAuth 2.0 specification lists four different types of authorization grants. Each type has different security characteristics. The authorization grant types are:

- ❑ Authorization Code
- ❑ Implicit
- ❑ Resource Owner Password Credentials
- ❑ Client Credentials

Authorization Code

An authorization grant using an authorization code works like this

1) The resource owner (user) accesses the client application.

2) The client application tells the user to login to the client application via an authorization server (e.g. Facebook, Twitter, Google etc.).

3) To login via the authorization server, the user is redirected to the authorization server by the client application. The client application sends its client ID along to the authorization server, so the authorization server knows which application is trying to access the protected resources.

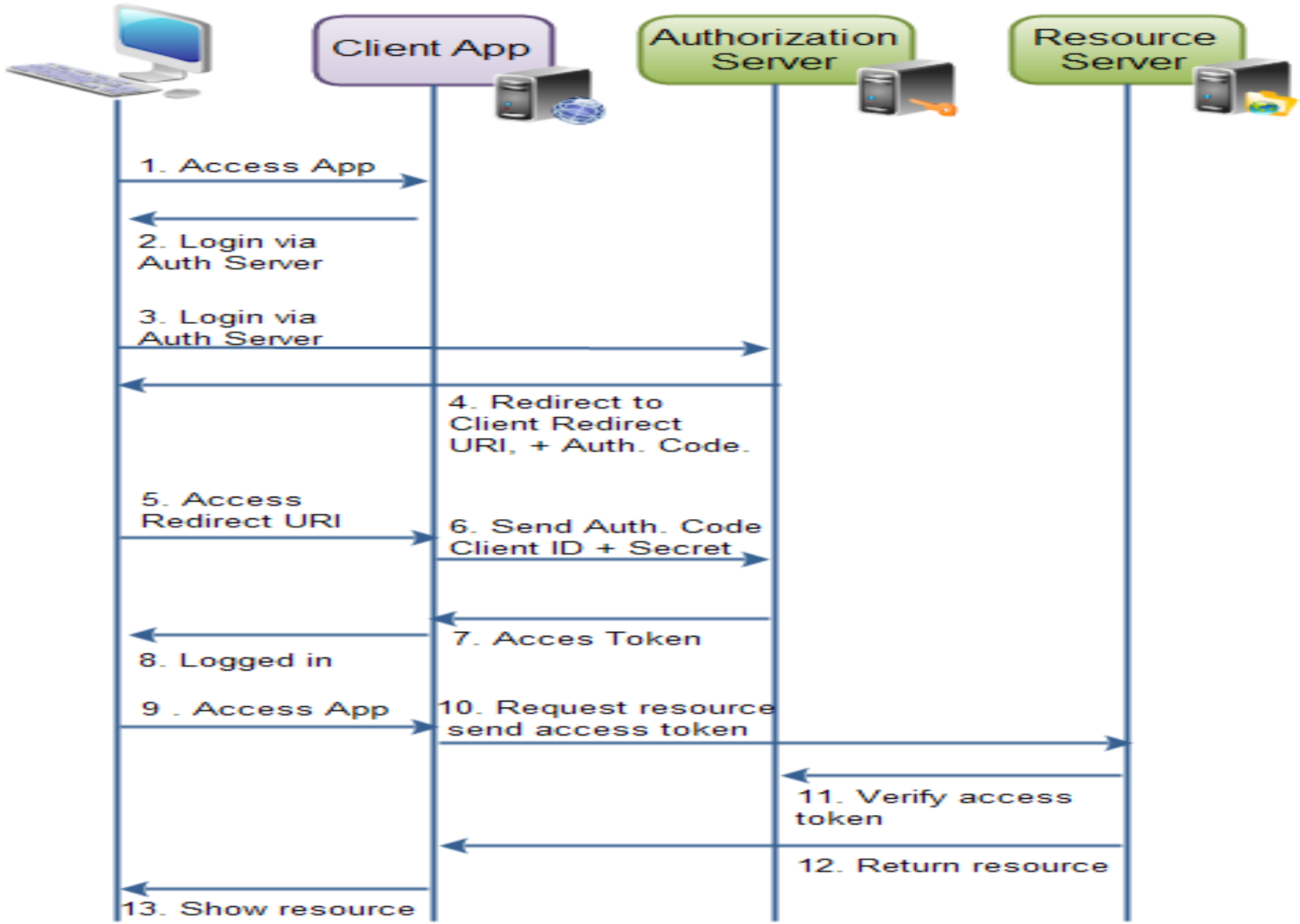
4) The user logs in via the authorization server. After successful login the user is asked if he wants to grant access to his resources to the client application. If the user accepts, the user is redirected back to the client application.

5) When redirected back to the client application, the authorization server sends the user to a specific redirect URI, which the client application has registered with the authorization server ahead of time. Along with the redirection, the authorization server sends an authorization code, representing the authorization.

6) When the redirect URI in the client application is accessed, the client application connects directly to the authorization server. The client application sends the authorization code along with its own client ID and and client secret.

7) If the authorization server can accept these values, the authorization server sends back an access token.

10) The client application can now use the access token to request resources from the resource server. The access token serves as both authentication of the client, resource owner (user) and authorization to access the resources.


Client App
Authorization Server
Resource Server
1. Access App
2. Login via Auth Server
3. Login via Auth Server
4. Redirect to Client Redirect URI, + Auth. Code.
5. Access Redirect URI
6. Send Auth. Code Client ID + Secret
7. Acces Token
8. Logged in
9 . Access App
10. Request resource send access token
11. Verify access token
12. Return resource
13. Show resource

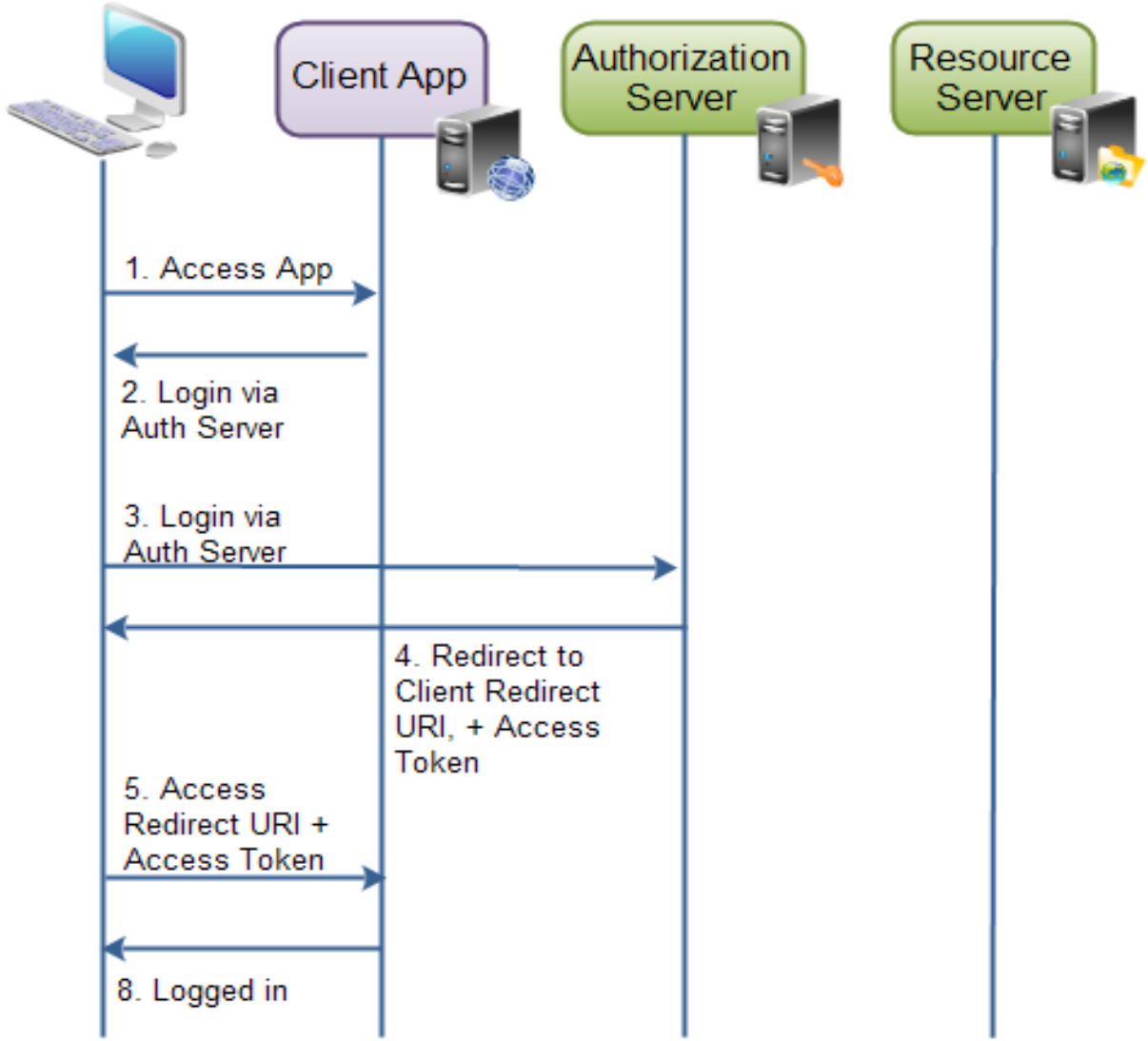
Implicit

An implicit authorization grant is similar to an authorization code grant, except the access token is returned to the client application already after the user has finished the authorization. The access token is thus returned when the user agent is redirected to the redirect URI.

This of course means that the access token is accessible in the user agent, or native application participating in the implicit authorization grant. The access token is not stored securely on a web server.

Furthermore, the client application can only send its client ID to the authorization server. If the client were to send its client secret too, the client secret would have to be stored in the user agent or native application too. That would make it vulnerable to hacking.

Implicit authorization grant is mostly used in a user agent or native client application. The user agent or native application would receive the access token from the authorization server.


Client App
Authorization Server
Resource Server
1. Access App
2. Login via Auth Server
3. Login via Auth Server
4. Redirect to Client Redirect URI, + Access Token
5. Access Redirect URI + Access Token
8. Logged in

Resource Owner Password Credentials

The resource owner password credentials authorization grant method works by giving the client application access to the resource owners credentials. For instance, a user could type his Twitter user name and password (credentials) into the client application. The client application could then use the user name and password to access resources in Twitter.

Using the resource owner password credentials requires a lot of trust in the client application. You do not want to type your credentials into an application you suspect might abuse it.

The resource owner password credentials would normally be used by user agent client applications, or native client applications.

Client Credentials

Client credential authorization is for the situations where the client application needs to access resources or call functions in the resource server, which are not related to a specific resource owner (e.g. user).

For instance, obtaining a list of venues from Foursquare. This does not necessary have anything to do with a specific Foursquare user.

OAuth 2.0 Authorization Code Requests and Responses

The authorization code grant consists of 2 requests and 2 responses in total.

An authorization request + response, and a token request + response.

Authorization Request

The authorization request is sent to the authorization endpoint to obtain an authorization code. Here are the parameters used in the request

response_type	Required. Must be set to code
client_id	Required. The client identifier as assigned by the authorization server, when the client was registered.
redirect_uri	Optional. The redirect URI registered by the client.
scope	Optional. The possible scope of the request.
state	Optional (recommended). Any client state that needs to be passed on to the client request URI.

Authorization Response

The authorization response contains the authorization code needed to obtain an access token.

code	Required. The authorization code.
state	Required, if present in request. The same value as sent by the client in the state parameter, if any.

Authorization Error Response

If an error occurs during authorization, two situations can occur.

The first is, that the client is not authenticated or recognized. For instance, a wrong redirect URI was sent in the request. In that case the authorization server must not redirect the resource owner to the redirect URI. Instead it should inform the resource owner of the error.

The second situation is that client is authenticated correctly, but that something else failed.

error	Required. Must be one of a set of predefined error codes. See the specification for the codes and their meaning.
error_description	Optional. A human-readable UTF-8 encoded text describing the error. Intended for a developer, not an end user.
error_uri	Optional. A URI pointing to a human-readable web page with information about the error.
state	Required, if present in authorization request. The same value as sent in the state parameter in the request.

Token Request

Once an authorization code is obtained, the client can use that code to obtain an access token. Here is the access token request parameters:

client_id	Required. The client application's id.
client_secret	Required. The client application's client secret .
grant_type	Required. Must be set to authorization_code .
code	Required. The authorization code received by the authorization server.
redirect_uri	Required, if the request URI was included in the authorization request. Must be identical then.

Token Response

The response to the access token request is a JSON string containing the access token plus some more information:

```
{ "access_token" : "...", "token_type" : "...", "expires_in" : "...",
"refresh_token" : "...", }
```

The access_token property is the access token as assigned by the authorization server.

The token_type property is a type of token assigned by the authorization server.

The expires_in property is a number of seconds after which the access token expires, and is no longer valid. Expiration of access tokens is optional.

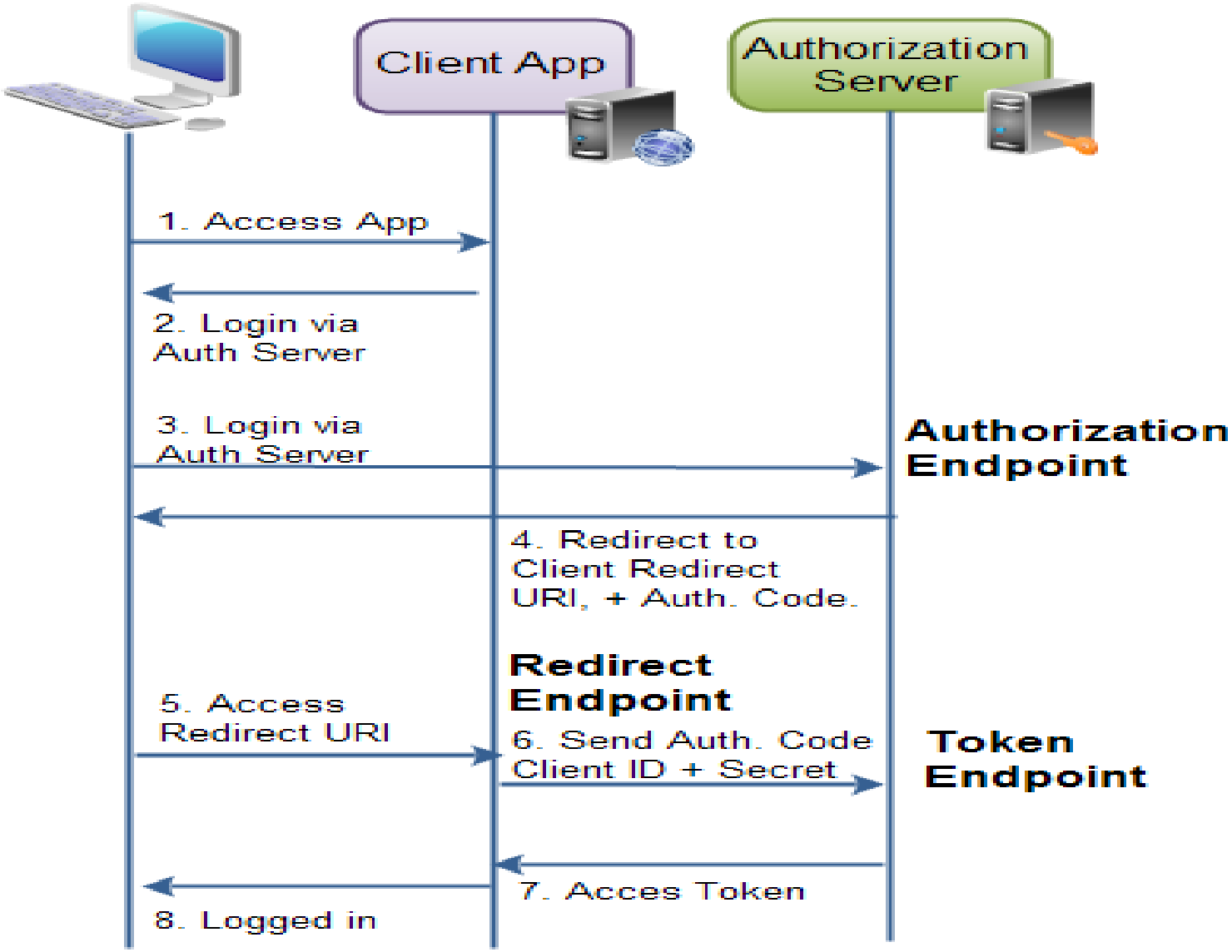
The refresh_token property contains a refresh token in case the access token can expire. The refresh token is used to obtain a new access token once the one returned in this response is no longer valid.

OAuth 2.0 Endpoints

OAuth 2.0 defines a set of endpoints. An endpoint is typically a URI on a web server. For instance, the address of a Java servlet, JSP page, PHP page, ASP.NET page etc.

The endpoints defined are:
•Authorization Endpoint
•Token Endpoint
•Redirection Endpoint

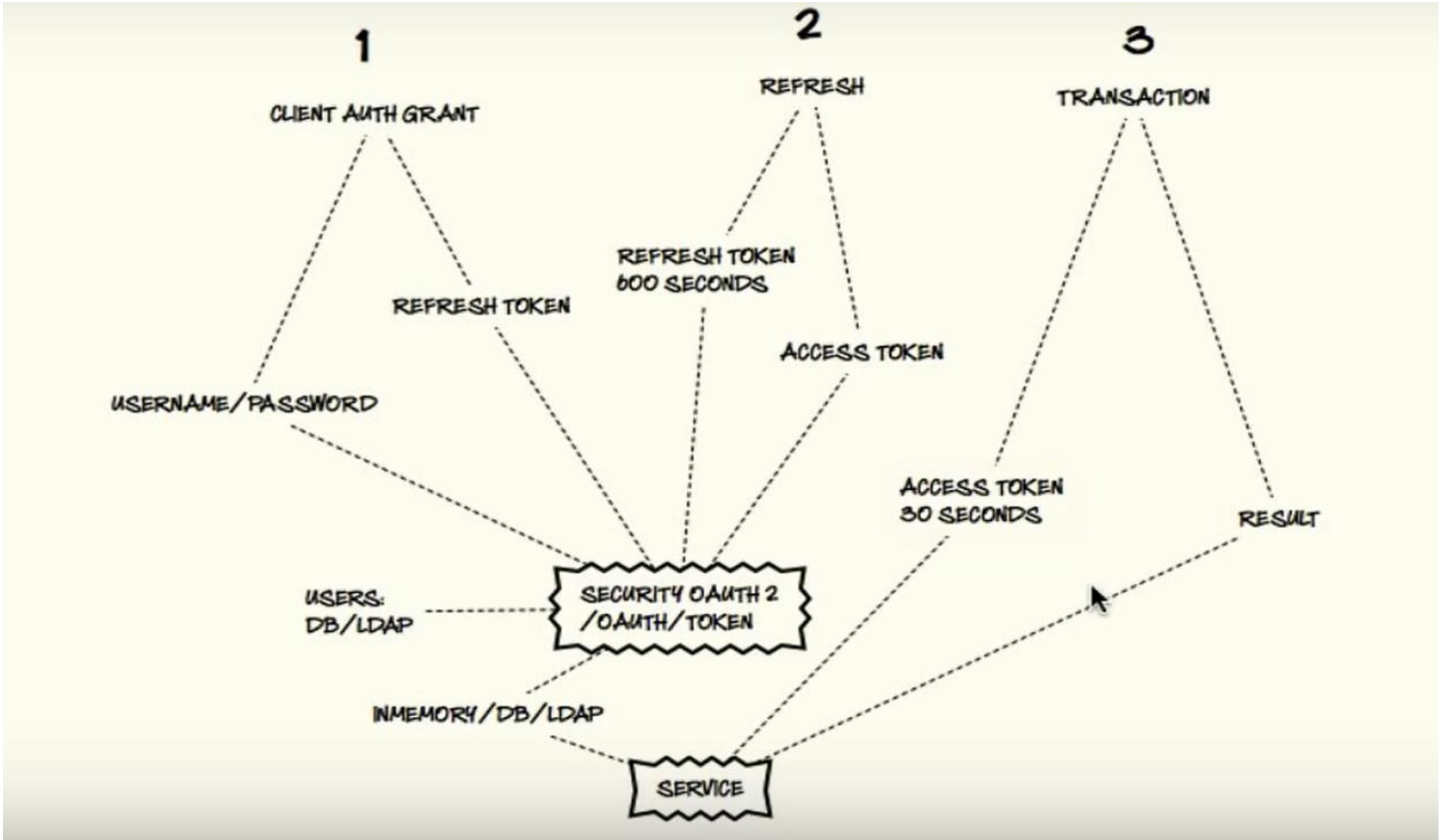
The authorization endpoint and token endpoint are both located on the authorization server. The redirection endpoint is located in the client application.


Client App
Authorization Server
1. Access App
2. Login via Auth Server
3. Login via Auth Server
Authorization Endpoint
4. Redirect to Client Redirect URI, + Auth. Code.
Redirect Endpoint
5. Access Redirect URI
6. Send Auth. Code Client ID + Secret
Token Endpoint
7. Acces Token
8. Logged in

Authorization Endpoint
The authorization endpoint is the endpoint on the authorization server where the resource owner logs in, and grants authorization to the client application.

Token Endpoint
The token endpoint is the endpoint on the authorization server where the client application exchanges the authorization code, client ID and client secret, for an access token.

Redirect Endpoint
The redirect endpoint is the endpoint in the client application where the resource owner is redirected to, after having granted authorization at the authorization endpoint.


1
2
3
CLIENT AUTH GRANT
REFRESH
TRANSACTION
REFRESH TOKEN
600 SECONDS
REFRESH TOKEN
ACCESS TOKEN
USERNAME/PASSWORD
ACCESS TOKEN
30 SECONDS
RESULT
USERS:
DB/LDAP
SECURITY OAUTH 2
/OAUTH/TOKEN
INMEMORY/DB/LDAP
SERVICE

@EnableResourceServer

The @EnableResourceServer annotation adds a filter of type OAuth2AuthenticationProcessingFilter automatically to the Spring Security filter chain. These incoming requests have to include OAuth2 token.