Application Security

Amit Kumar

Application Security

Application security describes security measures at the application level that aim to prevent data or code within the app from being stolen or hijacked. It encompasses the security considerations that happen during application development and design, but it also involves systems and approaches to protect apps after they get deployed.

Application security may include hardware, software, and procedures that identify or minimize security vulnerabilities. A router that prevents anyone from viewing a computer's IP address from the Internet is a form of hardware application security. But security measures at the application level are also typically built into the software, such as an application firewall that strictly defines what activities are allowed and prohibited. Procedures can entail things like an application security routine that includes protocols such as regular testing.

Application security definition

Application security is the process of developing, adding, and testing security features within applications to prevent security vulnerabilities against threats such as unauthorized access and modification.

Why application security is important

Application security is important because today's applications are often available over various networks and connected to the cloud, increasing vulnerabilities to security threats and breaches. There is increasing pressure and incentive to not only ensure security at the network level but also within applications themselves. One reason for this is because hackers are going after apps with their attacks more today than in the past. Application security testing can reveal weaknesses at the application level, helping to prevent these attacks.

Application Security

Types of application security

Different types of application security features include authentication, authorization, encryption, logging, and application security testing. Developers can also code applications to reduce security vulnerabilities.

- **Authentication**: When software developers build procedures into an application to ensure that only authorized users gain access to it. Authentication procedures ensure that a user is who they say they are. This can be accomplished by requiring the user to provide a user name and password when logging in to an application. Multi-factor authentication requires more than one form of authentication—the factors might include something you know (a password), something you have (a mobile device), and something you are (a thumb print or facial recognition).
- **Authorization**: After a user has been authenticated, the user may be authorized to access and use the application. The system can validate that a user has permission to access the application by comparing the user's identity with a list of authorized users. Authentication must happen before authorization so that the application matches only validated user credentials to the authorized user list.
- **Encryption**: After a user has been authenticated and is using the application, other security measures can protect sensitive data from being seen or even used by a cybercriminal. In cloud-based applications, where traffic containing sensitive data travels between the end user and the cloud, that traffic can be encrypted to keep the data safe.
- **Logging**: If there is a security breach in an application, logging can help identify who got access to the data and how. Application log files provide a time-stamped record of which aspects of the application were accessed and by whom.
- **Application security testing**: A necessary process to ensure that all of these security controls work properly.

Web Application Security

Web application security is a branch of information security that deals specifically with security of websites, web applications and web services. At a high level, web application security draws on the principles of application security but applies them specifically to internet and web systems.

Web Application Security Tools are specialized tools for working with HTTP traffic, e.g., Web application firewalls.

Security threats

The Open Web Application Security Project (OWASP) provides free and open resources. It is led by a non-profit called The OWASP Foundation. The OWASP Top 10 - 2017 is the published result of recent research based on comprehensive data compiled from over 40 partner organizations. From this data, approximately 2.3 million vulnerabilities were discovered across over 50,000 applications.[4] According to the OWASP Top 10 - 2017, the ten most critical web application security risks include:

- Injection
- Broken authentication
- Sensitive data exposure
- XML external entities (XXE)
- Broken access control
- Security misconfiguration
- Cross-site scripting (XSS)
- Insecure deserialization
- Using components with known vulnerabilities
- Insufficient logging and monitoring

Spring Security

Amit Kumar

Spring Security

Spring is considered a trusted framework in the Java ecosystem and is widely used. It's no longer valid to refer to Spring as a framework, as it's more of an umbrella term that covers various frameworks. One of these frameworks is Spring Security, which is a powerful and customizable authentication and authorization framework. It is considered the de facto standard for securing Spring-based applications.

Spring Security

- Spring Security is a framework which provides various security features like: authentication, authorization to create secure Java Enterprise Applications.

- It is a sub-project of Spring framework which was started in 2003 by Ben Alex. Later on, in 2004, It was released under the Apache License as Spring Security 2.0.0.

- It overcomes all the problems that come during creating non spring security applications and manage new server environment for the application

Spring Security

This framework targets two major areas of application are authentication and authorization. Authentication is the process of knowing and identifying the user that wants to access.

Authorization is the process to allow authority to perform actions in the application.

We can apply authorization to authorize web request, methods and access to individual domain.

Technologies that support Spring Security Integration

Spring Security framework supports wide range of authentication models. These models either provided by third parties or framework itself. Spring Security supports integration with all of these technologies.

Spring Security

HTTP BASIC authentication headers
HTTP Digest authentication headers
HTTP X.509 client certificate exchange
LDAP (Lighweight Directory Access Protocol)
Form-based authentication
OpenID authentication
Automatic remember-me authentication
Kerberos
JOSSO (Java Open Source Single Sign-On)
AppFuse
AndroMDA
Mule ESB
DWR(Direct Web Request)

Spring Security Advantages

Spring Security has numerous advantages.

Some of that are given below:

- Comprehensive support for authentication and authorization.
- Protection against common tasks
- Servlet API integration
- Integration with Spring MVC
- Portability
- CSRF protection
- Java Configuration support

Spring Security Features

- LDAP (Lightweight Directory Access Protocol)
- Single sign-on
- JAAS (Java Authentication and Authorization Service) LoginModule
- Basic Access Authentication
- Digest Access Authentication
- Remember-me
- Web Form Authentication
- Authorization
- Software Localization
- HTTP Authorization

Spring Security Features

LDAP (Lightweight Directory Access Protocol)

It is an open application protocol for maintaining and accessing distributed directory information services over an Internet Protocol.

Single sign-on

This feature allows a user to access multiple applications with the help of single account(user name and password).

JAAS (Java Authentication and Authorization Service) LoginModule

It is a Pluggable Authentication Module implemented in Java. Spring Security supports it for its authentication process.

Spring Security Features

Basic Access Authentication

Spring Security supports Basic Access Authentication that is used to provide user name and password while making request over the network.

Digest Access Authentication

This feature allows us to make authentication process more secure than Basic Access Authentication. It asks to the browser to confirm the identity of the user before sending sensitive data over the network.

Remember-me

Spring Security supports this feature with the help of HTTP Cookies. It remember to the user and avoid login again from the same machine until the user logout.

Spring Security Features

Web Form Authentication

In this process, web form collect and authenticate user credentials from the web browser. Spring Security supports it while we want to implement web form authentication.

Authorization

Spring Security provides the this feature to authorize the user before accessing resources. It allows developers to define access policies against the resources.

Software Localization

This feature allows us to make application user interface in any language.

HTTP Authorization

Spring provides this feature for HTTP authorization of web request URLs using Apache Ant paths or regular expressions.

Spring Security

Terminology

- **Authentication** refers to the process of verifying the identity of a user, based on provided credentials. A common example is entering a username and a password when you log in to a website. You can think of it as an answer to the question *Who are you?*.
- **Authorization** refers to the process of determining if a user has proper permission to perform a particular action or read particular data, assuming that the user is successfully authenticated. You can think of it as an answer to the question *Can a user do/read this?*.
- **Principle** refers to the currently authenticated user.
- **Granted authority** refers to the permission of the authenticated user.
- **Role** refers to a group of permissions of the authenticated user.

Filter

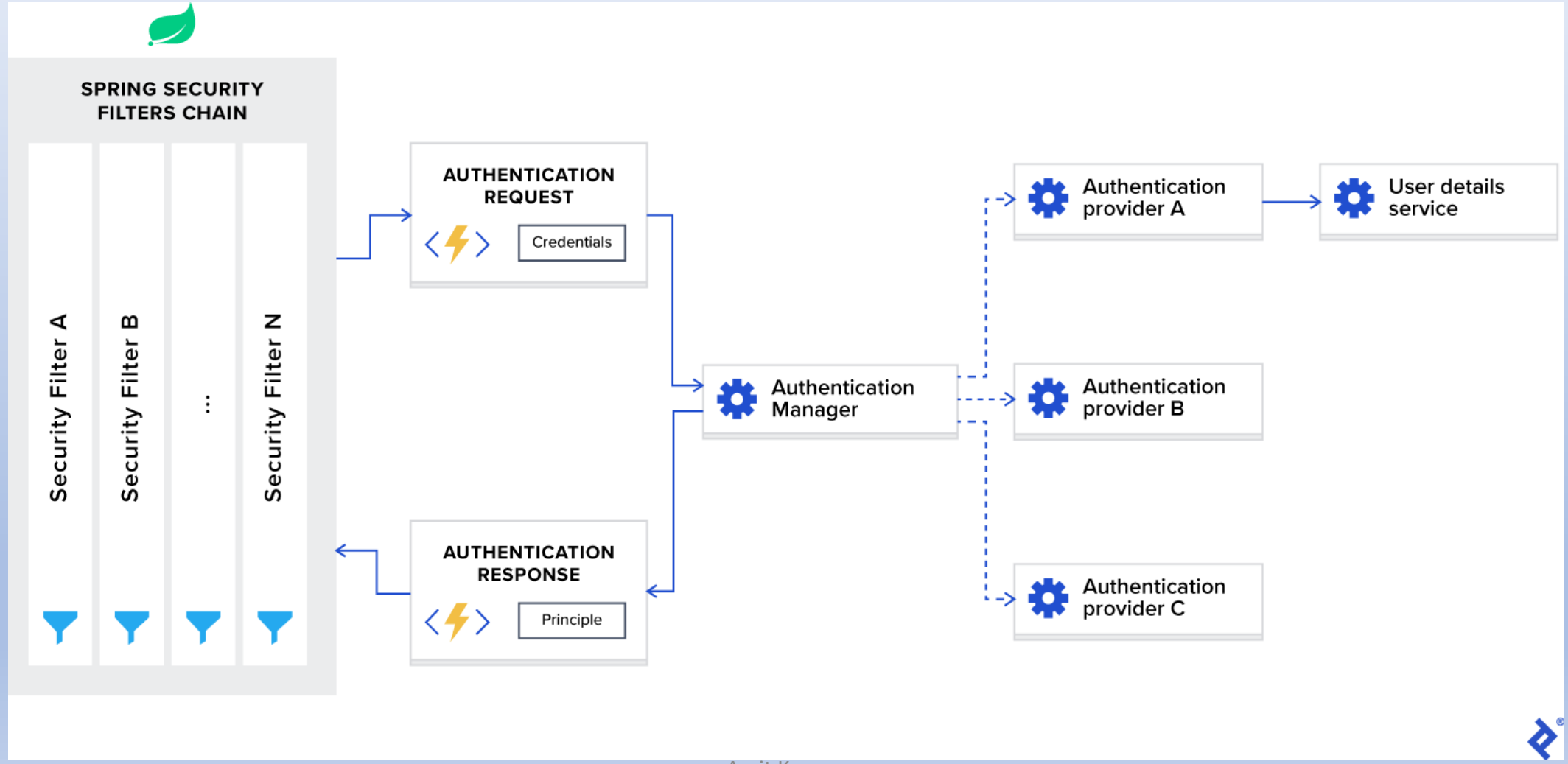
A filter is an object that performs filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both.

Filters perform filtering in the doFilter method. Every Filter has access to a FilterConfig object from which it can obtain its initialization parameters, a reference to the ServletContext which it can use, for example, to load resources needed for filtering tasks.
Filters are configured in the deployment descriptor of a web application

Examples that have been identified for this design are:

1) Authentication Filters
2) Logging and Auditing Filters
3) Image conversion Filters
4) Data compression Filters
5) Encryption Filters
6) Tokenizing Filters
7) Filters that trigger resource access events
8) XSL/T filters
9) Mime-type chain Filter

Spring Security Architecture



Spring Security Filters Chain

When you add the Spring Security framework to your application, it automatically registers a filters chain that intercepts all incoming requests. This chain consists of various filters, and each of them handles a particular use case.

For example:

- Check if the requested URL is publicly accessible, based on configuration.
- In case of session-based authentication, check if the user is already authenticated in the current session.
- Check if the user is authorized to perform the requested action, and so on.
- One important detail I want to mention is that Spring Security filters are registered with the lowest order and are the first filters invoked. For some use cases, if you want to put your custom filter in front of them, you will need to add padding to their order. This can be done with the following configuration:

spring.security.filter.order=10

Once we add this configuration to our application.properties file, we will have space for 10 custom filters in front of the Spring Security filters.

Spring Security Architecture

AuthenticationManager

You can think of AuthenticationManager as a coordinator where you can register multiple providers, and based on the request type, it will deliver an authentication request to the correct provider.

AuthenticationProvider

AuthenticationProvider processes specific types of authentication. Its interface exposes only two functions:

authenticate performs authentication with the request.

supports checks if this provider supports the indicated authentication type.

One important implementation of the interface that we are using in our sample project is DaoAuthenticationProvider, which retrieves user details from a UserDetailsService.

Spring Security Architecture

UserDetailsService

UserDetailsService is described as a core interface that loads user-specific data in the Spring documentation.

In most use cases, authentication providers extract user identity information based on credentials from a database and then perform validation. Because this use case is so common, Spring developers decided to extract it as a separate interface, which exposes the single function:

loadUserByUsername accepts username as a parameter and returns the user identity object.

Spring Security Architecture

Authentication Using JWT with Spring Security

After discussing the internals of the Spring Security framework, let's configure it for stateless authentication with a JWT token.

To customize Spring Security, we need a configuration class annotated with @EnableWebSecurity annotation in our classpath. Also, to simplify the customization process, the framework exposes a WebSecurityConfigurerAdapter class. We will extend this adapter and override both of its functions so as to:

- Configure the authentication manager with the correct provider
- Configure web security (public URLs, private URLs, authorization, etc.)

Spring Security Architecture

Authentication Using JWT with Spring Security

After discussing the internals of the Spring Security framework, let's configure it for stateless authentication with a JWT token.

To customize Spring Security, we need a configuration class annotated with @EnableWebSecurity annotation in our classpath. Also, to simplify the customization process, the framework exposes a WebSecurityConfigurerAdapter class. We will extend this adapter and override both of its functions so as to:

- Configure the authentication manager with the correct provider
- Configure web security (public URLs, private URLs, authorization, etc.)

Spring Security Architecture

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        // TODO configure authentication manager
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // TODO configure web security
    }
}
```

Spring Security Architecture

we store user identities in a MongoDB database, in the users collection. These identities are mapped by the User entity, and their CRUD operations are defined by the UserRepo Spring Data repository.

Now, when we accept the authentication request, we need to retrieve the correct identity from the database using the provided credentials and then verify it. For this, we need the implementation of the UserDetailsService interface, which is defined as follows:

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException;

}
```

Spring Security Architecture

Authorization with Spring Security

In the previous section, we set up an authentication process and configured public/private URLs. This may be enough for simple applications, but for most real-world use cases, we always need role-based access policies for our users. In this chapter, we will address this issue and set up a role-based authorization schema using the Spring Security framework.

In our sample application, we have defined the following three roles:

USER_ADMIN allows us to manage application users.

AUTHOR_ADMIN allows us to manage authors.

BOOK_ADMIN allows us to manage books.

Spring Security Architecture

Now, we need to apply them to the corresponding URLs:

api/public is publicly accessible.

api/admin/user can access users with the USER_ADMIN role.

api/author can access users with the AUTHOR_ADMIN role.

api/book can access users with the BOOK_ADMIN role.

The Spring Security framework provides us with two options to set up the authorization schema:

URL-based configuration

Annotation-based configuration

Spring Security Architecture

The Spring Security framework defines the following annotations for web security:

@PreAuthorize supports Spring Expression Language and is used to provide expression-based access control before executing the method.

@PostAuthorize supports Spring Expression Language and is used to provide expression-based access control after executing the method (provides the ability to access the method result).

@PreFilter supports Spring Expression Language and is used to filter the collection or arrays before executing the method based on custom security rules we define.

@PostFilter supports Spring Expression Language and is used to filter the returned collection or arrays after executing the method based on custom security rules we define (provides the ability to access the method result).

@Secured doesn't support Spring Expression Language and is used to specify a list of roles on a method.

@RolesAllowed doesn't support Spring Expression Language and is the JSR 250's equivalent annotation of the @Secured annotation.

Spring Security Architecture

These annotations are disabled by default and can be enabled in our application as follows:

```
@EnableWebSecurity
@EnableGlobalMethodSecurity(
    securedEnabled = true,
    jsr250Enabled = true,
    prePostEnabled = true
)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    // Details omitted for brevity
}
```

securedEnabled = true enables @Secured annotation.

jsr250Enabled = true enables @RolesAllowed annotation.

prePostEnabled = true enables @PreAuthorize, @PostAuthorize, @PreFilter, @PostFilter annotations.

Spring Security Architecture

Role Name Default Prefix

In this separate subsection, I want to emphasize one more subtle detail that confuses a lot of new users.

The Spring Security framework differentiates two terms:

- Authority represents an individual permission.
- Role represents a group of permissions.

Both can be represented with a single interface called GrantedAuthority and later checked with Spring Expression Language inside the Spring Security annotations as follows:

- *Authority: @PreAuthorize("hasAuthority('EDIT_BOOK')")*
- *Role: @PreAuthorize("hasRole('BOOK_ADMIN')")*

To make the difference between these two terms more explicit, the Spring Security framework adds a ROLE_ prefix to the role name by default. So, instead of checking for a role named BOOK_ADMIN, it will check for ROLE_BOOK_ADMIN.

Spring Security Remember Me

Remember me is a feature that allows a user to access into application without re-login. User's login session terminates after closing the browser and if user again access the application by opening browser, it prompts for login.

But we can avoid this re-login by using remember me feature. It stores user's identity into the Cookie or database and use to identity the user.

Spring Security at Method Level

Apart from authentication, spring security also check authorization of the logged in user. After login which user is authorize to access the resource is done on the bases of user's ROLE.

At the time of creating user in WebSecurityConfig class, we can specify user?s ROLE as well.

Security applied on a method restricts to unauthorized user and allow only authentic user.

Spring Security CSRF

Assume that your bank's website provides a form that allows transferring money from the currently logged in user to another bank account. For example, the HTTP request might look like:

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded
amount=100.00&routingNumber=1234&account=9876
```

Now pretend you authenticate to your bank's website and then, without logging out, visit an evil website. The evil website contains an HTML page with the following form:

```
<form action="https://bank.example.com/transfer" method="post">
<input type="hidden“ name="amount“ value="100.00"/>
<input type="hidden“ name="routingNumber“ value="evilsRoutingNumber"/>
<input type="hidden“ name="account“ value="evilsAccountNumber"/>
<input type="submit“ value="Win Money!"/>
</form>
```

You like to win money, so you click on the submit button. In the process, you have unintentionally transferred $100 to a malicious user. This happens because, while the evil website cannot see your cookies, the cookies associated with your bank are still sent along with the request.

Worst yet, this whole process could have been automated using JavaScript. This means you didn't even need to click on the button. So how do we protect ourselves from such attacks?

Security Config

This class extends **WebSecurityConfigurerAdapter** and provides usual spring security configuration. Here, we are using bcrypt encoder to encode our passwords. You can try this online Bcrypt Tool to encode and match bcrypt passwords. Following configuration basically bootstraps the authorization server and resource server.

@**EnableWebSecurity** : Enables spring security web security support.

@**EnableGlobalMethodSecurity**: Support to have method level access control such as @**PreAuthorize** @**PostAuthorize**

Spring Boot SSL

configure web application to run on SSL (HTTPS) with self-signed certificate:

Spring boot HTTPS Config:

server.port=8443

server.ssl.key-alias=selfsigned_localhost_sslserver

server.ssl.key-password=changeit

server.ssl.key-store=classpath:ssl-server.jks

server.ssl.key-store-provider=SUN

server.ssl.key-store-type=JKS

Spring Boot SSL

configure web application to run on SSL (HTTPS) with self-signed certificate:

Redirect from HTTP to HTTPS

```
private Connector redirectConnector() {
  Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");
  connector.setScheme("http");
  connector.setPort(8080);
  connector.setSecure(false);
  connector.setRedirectPort(8443);
  return connector;
}
```

Spring Boot SSL

Terminology:

SSL – stands for Secure Sockets Layer. It is the industry standard protocol for keeping an internet connection secure by safeguarding all sensitive data that is being sent between two systems, preventing hackers from reading and modifying any information transferred.

TLS – (Transport Layer Security) is an updated, more secure, version of SSL. It adds more features. Today, certificates provided by certificate authorities are based on TLS only. But regarding secured communication over network, the term SSL is still common as it is the old and just become popular among community.

Spring Boot SSL

Terminology:

HTTPS – (Hyper Text Transfer Protocol Secure) appears in the URL when a website is secured by an SSL certificate. It is the secured version of HTTP protocol.

Truststore and Keystore – Those are used to store SSL certificates in Java but there is little difference between them. truststore is used to store public certificates while keystore is used to store private certificates of client or server.

Spring Boot SSL

Create your own self signed SSL certificate:

To get SSL digital certificate for our application we have two options –

- to create a self-signed certificate
- to obtain SSL certificate from certification authority(CA) we call it CA certificate.

We will create self-signed certificate generated by java keytool command. We need to run the keytool -genkey command from command prompt.

keytool -genkey -alias selfsigned_localhost_sslserver -keyalg RSA -keysize 2048 -validity 700 -keypass changeit -storepass changeit -keystore ssl-server.jks

Spring Boot SSL

- -genkey – is the keytool command to generate the certificate, actually keytool is a multipurpose and robust tool which has several options
- -alias selfsigned_localhost_sslserver – indicates the alias of the certificate, which is used by SSL/TLS layer
- -keyalg RSA -keysize 2048 -validity 700 – are self descriptive parameters indicating the crypto algorithm, keysize and certificate validity.
- -keypass changeit -storepass changeit – are the passwords of our truststore and keystore
- -keystore ssl-server.jks – is the actual keystore where the certificate and public/private key will be stored. Here we are using JKS fromat – Java Key Store, there are other formats as well for keystore.

Spring Boot SSL

```
C:\Program Files\Java\jdk-11.0.11\bin>keytool -genkey -alias selfsigned_localhost_sslserver -keyalg RSA -keysize 2048 -validity 700 -keypass changeit -storepass cha
eystore ssl-server.jks
What is your first and last name?
  [Unknown]:  Amit Kumar
What is the name of your organizational unit?
  [Unknown]:  amit.co.sample
What is the name of your organization?
  [Unknown]:  amit.co
What is the name of your City or Locality?
  [Unknown]:  pune
What is the name of your State or Province?
  [Unknown]:  maharastra
What is the two-letter country code for this unit?
  [Unknown]:  IN
Is CN=Amit Kumar, OU=amit.co.sample, O=amit.co, L=pune, ST=maharastra, C=IN correct?
  [no]:  yes


C:\Program Files\Java\jdk-11.0.11\bin>keytool -list -keystore ssl-server.jks
Enter keystore password:
Keystore type: PKCS12
Keystore provider: SUN

Your keystore contains 1 entry

selfsigned_localhost_sslserver, 26-Apr-2021, PrivateKeyEntry,
Certificate fingerprint (SHA-256): D2:3B:41:E4:1F:15:99:A0:18:79:9A:E9:23:28:A4:78:81:C6:62:2C:9C:1C:CC:C0:5E:7F:5A:DA:C7:49:B4:69

C:\Program Files\Java\jdk-11.0.11\bin>
```

OAuth

What is OAuth

OAuth is simply a secure authorization protocol that deals with the authorization of third party application to access the user data without exposing their password. eg. (Login with fb, gPlus, twitter in many websites..) all work under this protocol.The Protocol becomes easier when you know the involved parties. Basically there are three parties involved: oAuth Provider, oAuth Client and Owner.Here, oAuth Provider provides the auth token such as Facebook, twitter. Similarly, oAuth Client are the the applications which want access of the credentials on behalf of owner and owner is the user which has account on oAuth providers such as facebook and twitter.

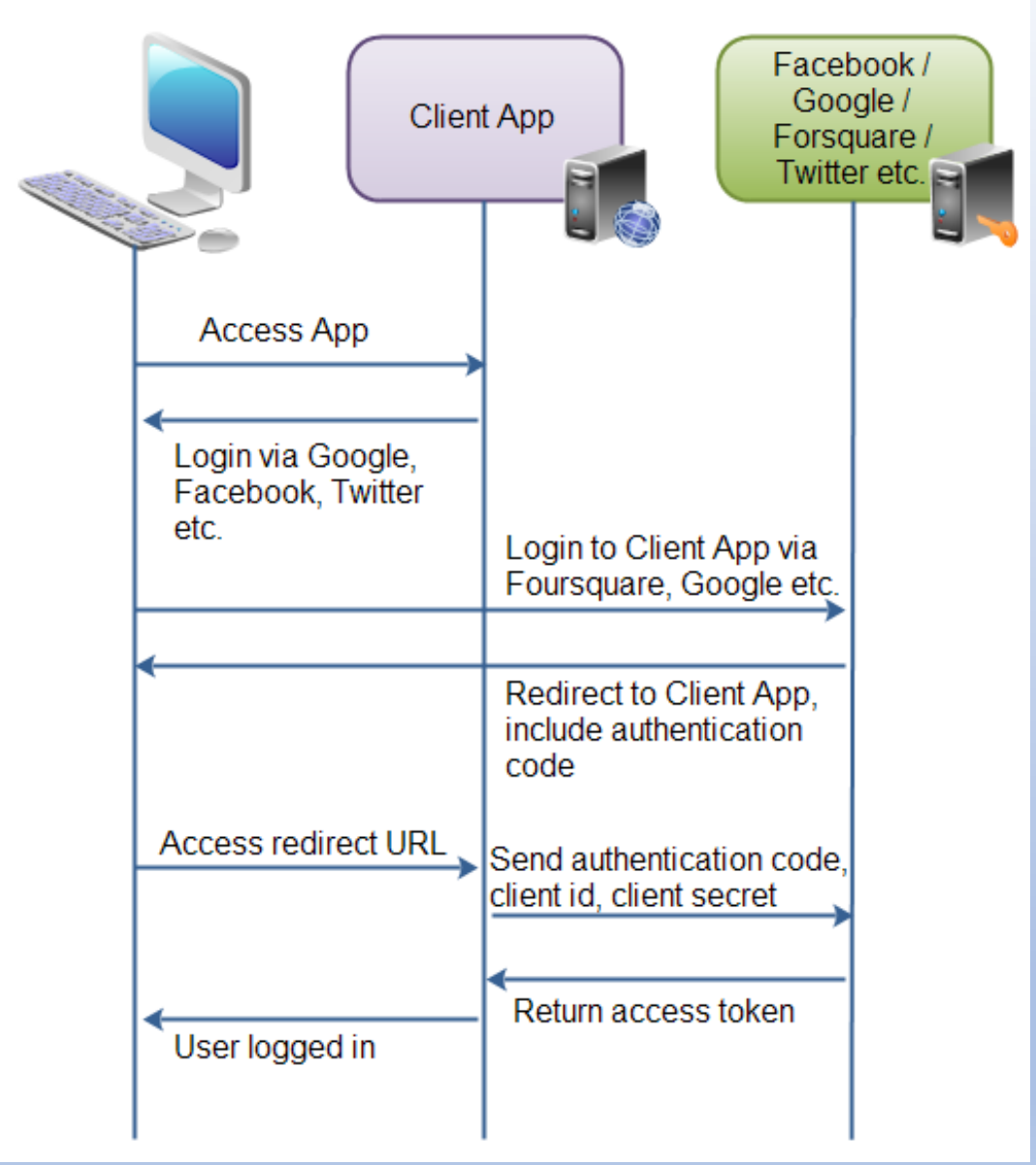
OAuth

What is OAuth

OAuth 2 is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service, such as Facebook, GitHub, and DigitalOcean. It works by delegating user authentication to the service that hosts the user account, and authorizing third-party applications to access the user account. OAuth 2 provides authorization flows for web and desktop applications, and mobile devices.

OAuth

OAuth 2.0 covers different ways a client application can obtain authorization to access the resources stored on the resource server. Here I will show you the most common, and most secure use case: A client web application requesting access to resources in another web application.



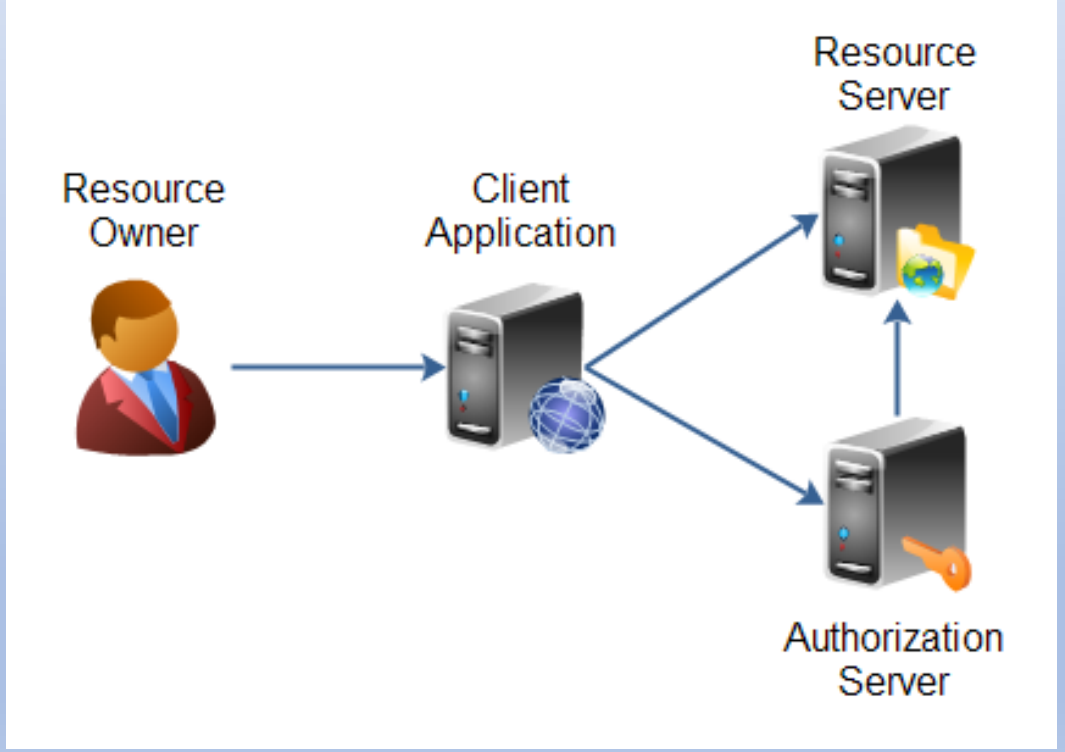
OAuth

- First the user accesses the client web application. In this web app is button saying "Login via Facebook" (or some other system like Google or Twitter).
- Second, when the user clicks the login button, the user is redirected to the authenticating application (e.g. Facebook). The user then logs into the authenticating application, and is asked if she wants to grant access to her data in the authenticating application, to the client application. The user accepts.
- Third, the authenticating application redirects the user to a redirect URI, which the client app has provided to the authenticating app. Providing this redirect URI is normally done by registering the client application with the authenticating application. During this registration the owner of the client application registers the redirect URI. It is also during this registration that the authenticating application gives the client application a client id and a client password. To the URI is appended an authentication code. This code represents the authentication.
- Fourth, the user accesses the page located at the redirect URI in the client application. In the background the client application contacts the authenticating application and sends client id, client password and the authentication code received in the redirect request parameters. The authenticating application sends back an access token.
- Once the client application has obtained an access token, this access token can be sent to the Facebook, Google, Twitter etc. to access resources in these systems, related to the user who logged in.

OAuth

OAuth 2.0 defines the following roles of users and applications:

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



OAuth

- 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 is depicted in the diagram as a person, which is probably the most common situation. 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. The OAuth 2.0 specification does not say anything about how these two servers should communicate, if they are separate. This is an internal design decision to be made by the resource server + authorization server developers.

OAuth

The OAuth 2.0 client role is subdivided into a set of client types and profiles. This text will explain these types and profiles.

The OAuth 2.0 specification defines two types of clients:

- Confidential
- Public

A confidential client is an application that is capable of keeping a client password confidential to the world. This client password is assigned to the client app by the authorization server. This password is used to identify the client to the authorization server, to avoid fraud. An example of a confidential client could be a web app, where no one but the administrator can get access to the server, and see the client password.

A public client is an application that is not capable of keeping a client password confidential. For instance, a mobile phone application or a desktop application that has the client password embedded inside it. Such an application could get cracked, and this could reveal the password. The same is true for a JavaScript application running in the users browser. The user could use a JavaScript debugger to look into the application, and see the client password.

OAuth

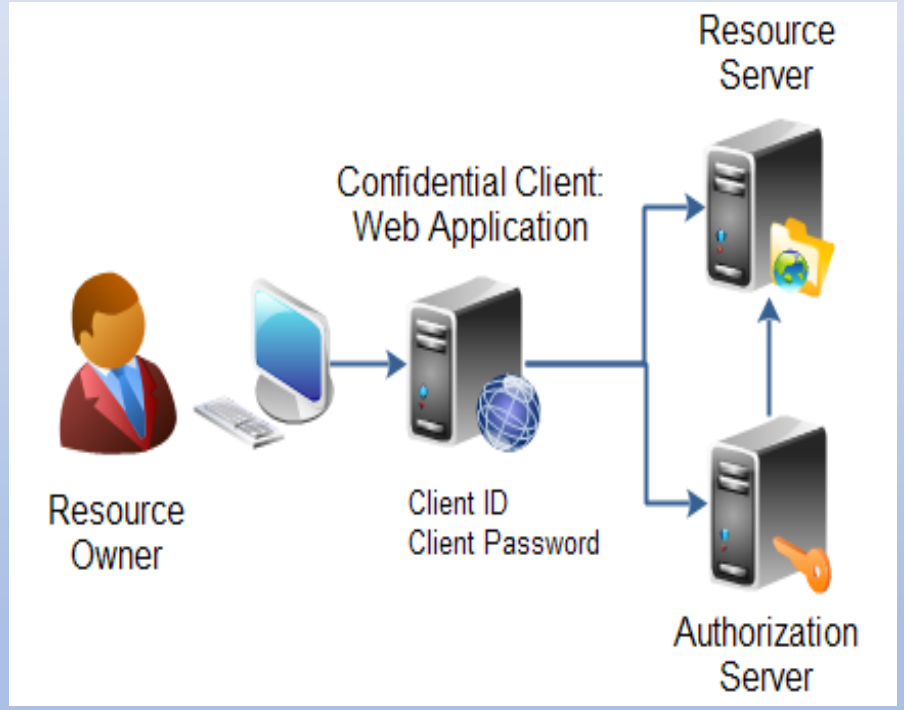
Client Profiles

The OAuth 2.0 specification also mentions a set of client profiles. These profiles are concrete types of applications, that can be either confidential or public. The profiles are:

- Web Application
- User Agent
- Native

Web Application

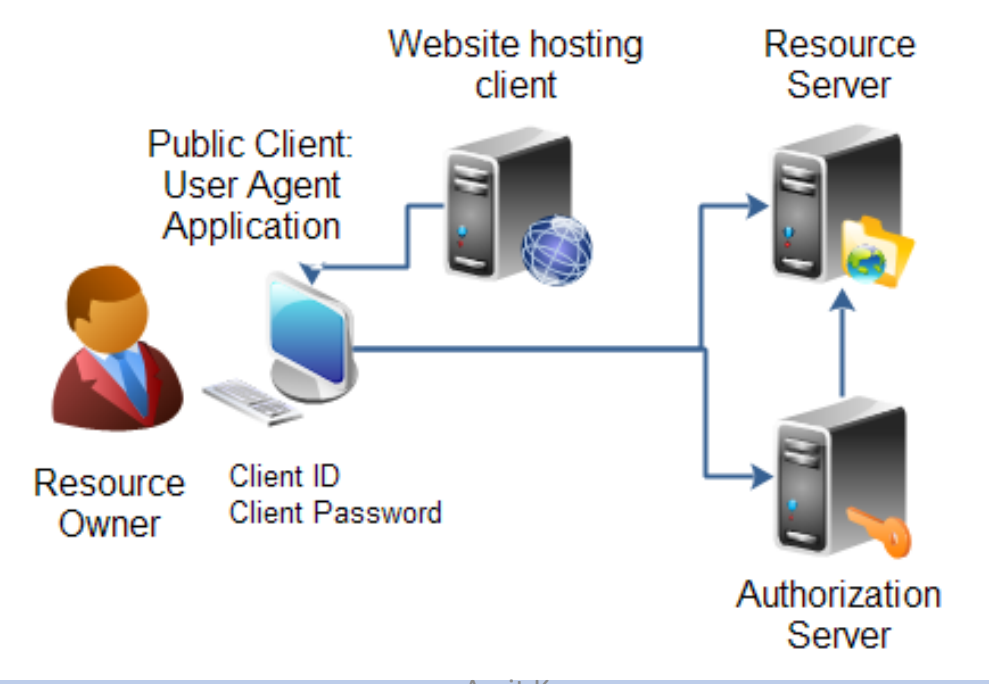
- A web application is an application running on a web server. In reality, a web application typically consists of both a browser part and a server part. If a web application needs access to a resource server (e.g. to Facebook user accounts), then the client password could be stored on the server. The password would thus be confidential.



OAuth

User Agent Application

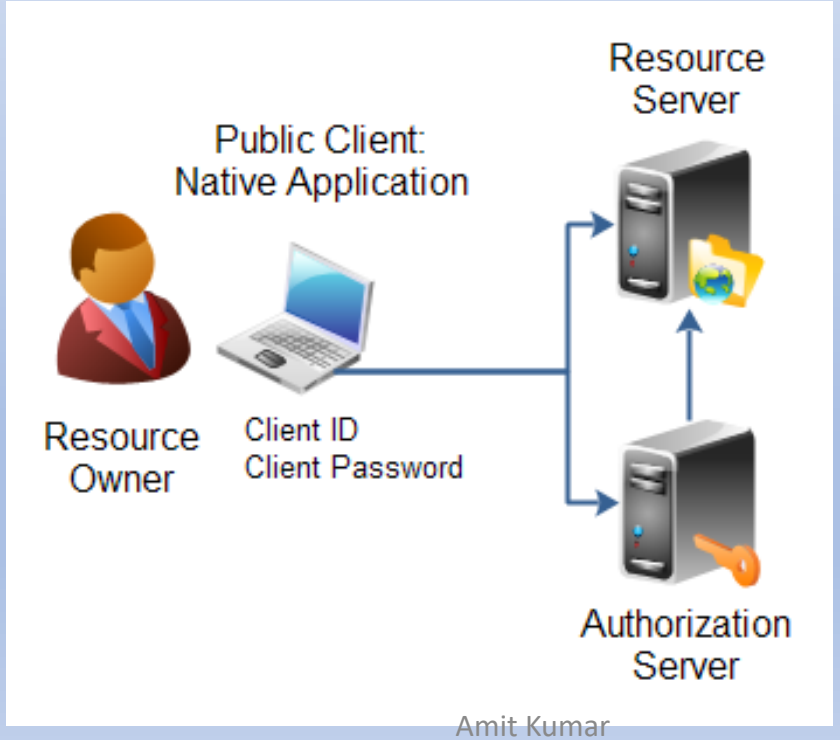
A user agent application is for instance a JavaScript application running in a browser. The browser is the user age. A user agent application may be stored on a web server, but the application is only running in the user agent once downloaded. An example could be a little JavaScript game that only runs in the browsernt.



OAuth

Native Application

A native application is for instance a desktop application or a mobile phone application. Native applications are typically installed on the users computer or device (phone, tablet etc.). Thus, the client password will be stored on the users computer or device too.



OAuth

OAuth 2.0 Authorization

- Client ID, Client Secret and Redirect URI
- Authorization Grant
- Authorization Code
- Implicit
- Resource Owner Password Credentials
- Client Credentials

- When a client applications wants access to the resources of a resource owner, hosted on a resource server, the client application must first obtain an authorization grant. This text explains how such an application grant is obtained.

OAuth

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. If a client application registers with multiple authorization servers (e.g. both Facebook, Twitter and Google), each authorization server will issue its own unique client ID to the client application.

Whenever the client application requests access to resources stored on that same resource server, the client application needs to authenticate itself by sending along the client ID and the client secret to the authorization server.

During the registration the client also registers a redirect URI. This redirect URI is used when a resource owner grants authorization to the client application. When a resource owner has successfully authorized the client application via the authorization server, the resource owner is redirected back to the client application, to the redirect URI.

OAuth

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

Each of these authorization grant types is covered in the following sections.

OAuth

Authorization Code

An authorization grant using an authorization code works like this (the numbers correspond to the steps shown in the diagram below the description):

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 she wants to grant access to her resources to the client application. If the user accepts, the user is redirected back to the client application.

OAuth

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.

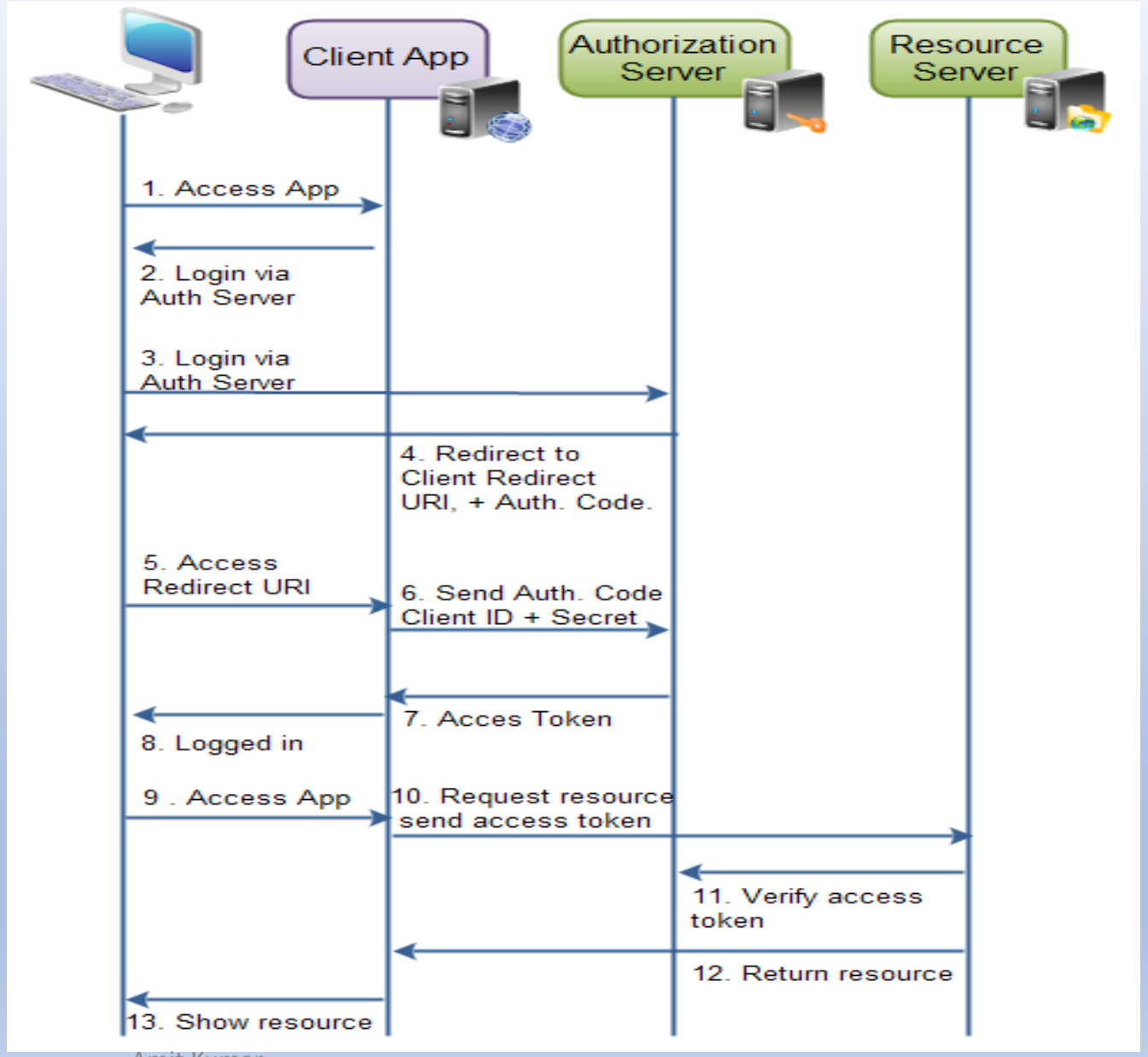
6) When the redirect URI in the client application is accessed, the client application connects directly to the authorization server. The client application sends the authorization code along with its own client ID and and client secret.

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.

OAuth

Authorization grant via authorization code.



OAuth

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.

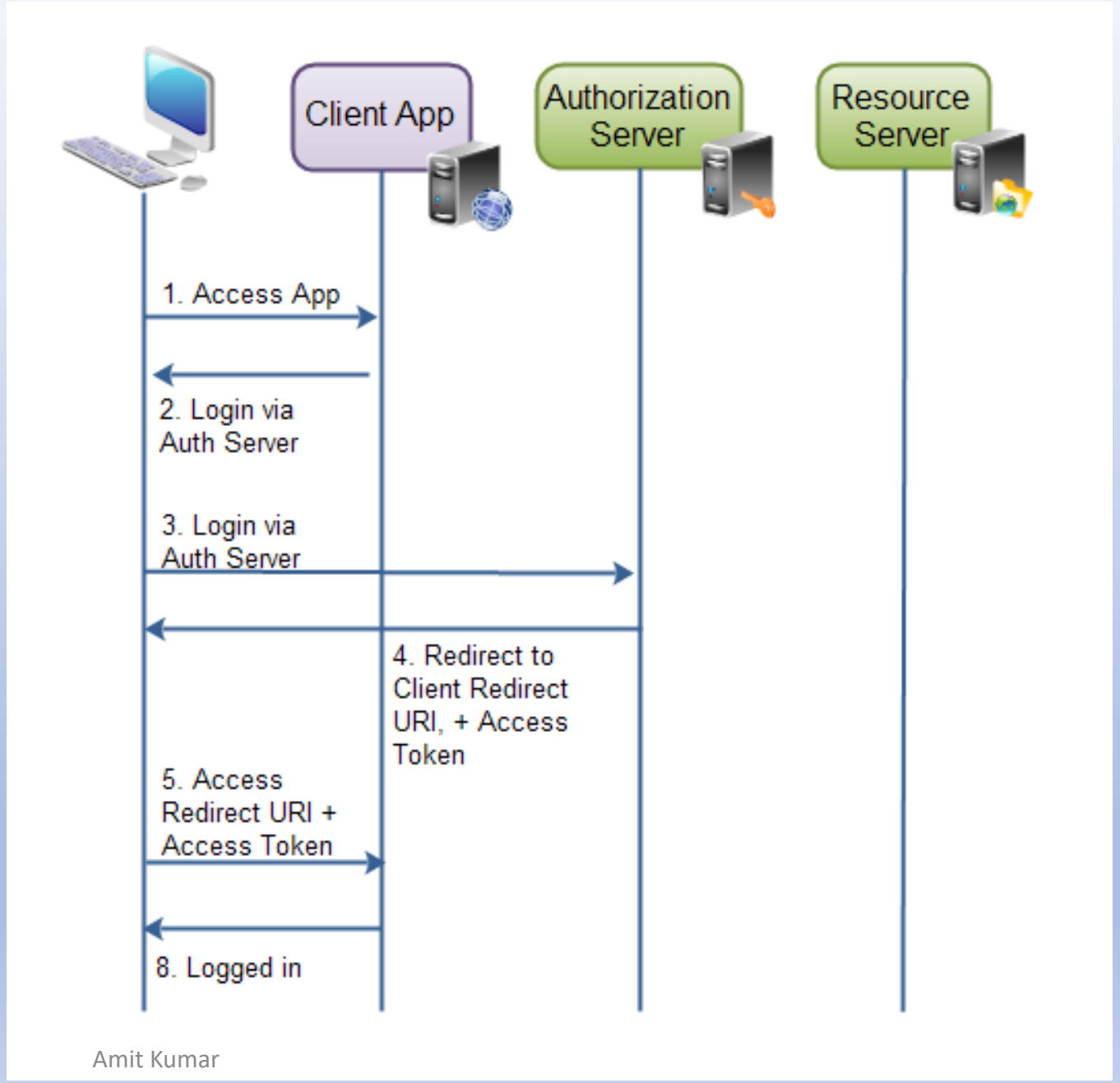
This of course means that the access token is accessible in the user agent, or native application participating in the implicit authorization grant. The access token is not stored securely on a web server.

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.

OAuth

Implicit authorization grant



OAuth 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.

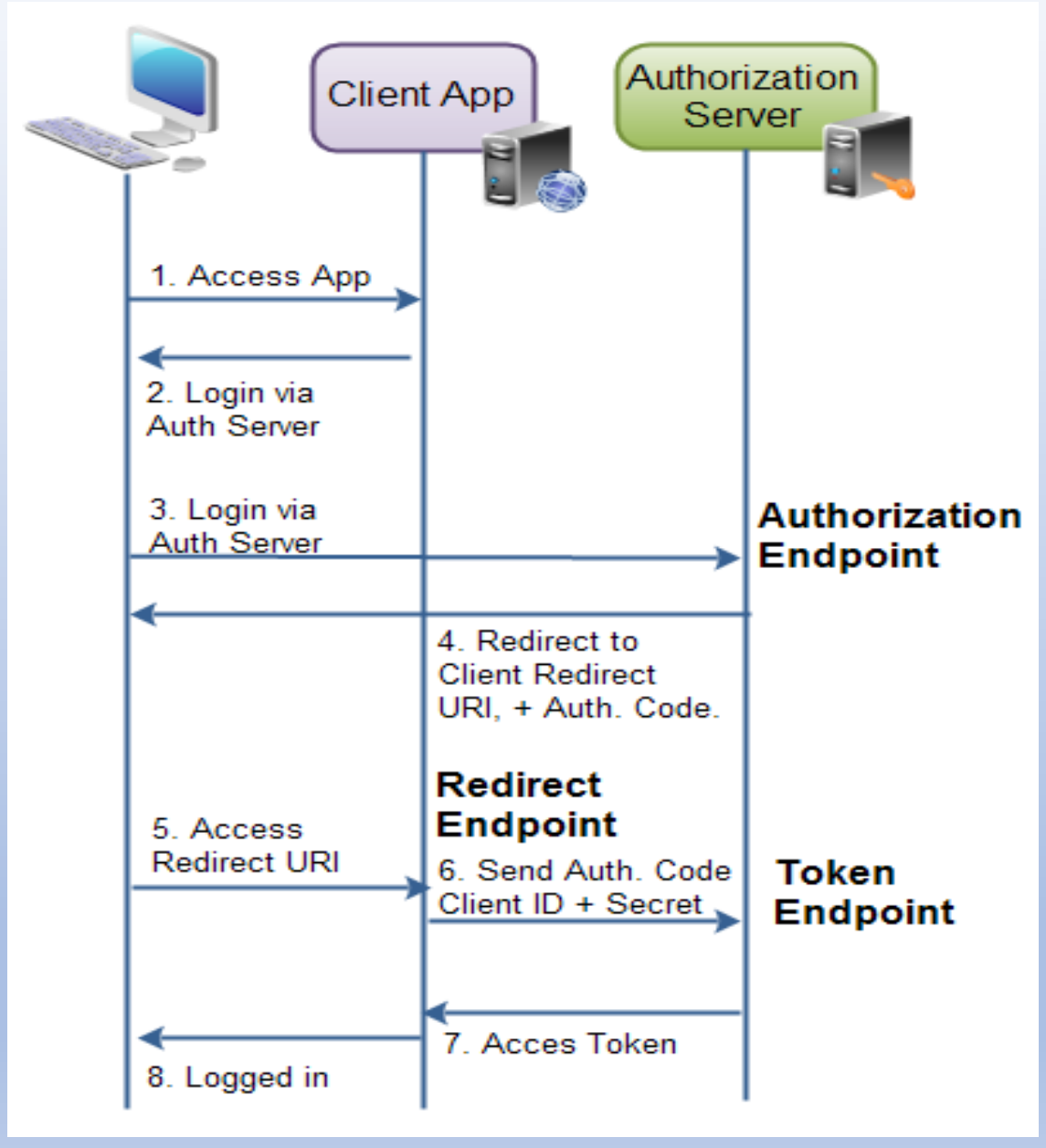
The endpoints defined are:

- 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. Each of these endpoints are covered below.

OAuth Endpoints

OAuth 2.0 Endpoints



OAuth

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.

OAuth

OAuth 2.0 Requests and Responses:

When the client application requests authorization and access tokens it sends HTTP requests to the authorization server, to its authorization and token endpoints. What request and response is sent forth and back depends on the authorization grant type. Remember, the four grant types are:

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

The request and response of each of these authorization grant types is explained in more detail in the following, separate texts.

The information presented in the following texts is, however, mostly a summary. In order to get all the details of what it all means, you may have to consult the OAuth 2.0 specification, or the documentation of the system (Facebook, Twitter, Foursquare etc.) you are trying to integrate with.

OAuth

OAuth 2.0 Authorization Code Requests and Responses:

- Authorization Request
- Authorization Response
- Authorization Error Response
- Token Request
- Token Response

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:

OAuth

Authorization 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.

OAuth

Authorization Response

The authorization response contains the authorization code needed to obtain an access token. Here are the parameters included in the response:

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.

OAuth

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. In that case the following error response is sent to the client, included in the redirect URI:

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.

OAuth

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.

OAuth

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

OAuth 2.0 Implicit Requests and Responses

- Implicit Grant Request
- Implicit Grant Response
- Implicit Grant Error Response

The implicit grant consists of only 1 request and 1 response.

Implicit Grant Request

The implicit grant request contains the following parameters:

response_type	Required. Must be set to token .
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.

OAuth

- **Implicit Grant Response**

The implicit grant response contains the following parameters. Note, that the implicit grant response is not JSON.

access_token	Required. The access token assigned by the authorization server.
token_type	Required. The type of the token
expires_in	Recommended. A number of seconds after which the access token expires.
scope	Optional. The scope of the access token.
state	Required, if present in the autorization request. Must be same value as state parameter in request.

OAuth

Implicit Grant 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 okay, but that something else happened. In that case the following error response is sent to the client, included in the redirect URI:

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.

OAuth

OAuth 2.0 Resource Owner Password Credentials Grant - Requests and Response:

- Resource Owner Password Credentials Grant Request
- Resource Owner Password Credentials Grant Response

The resource owner password credentials authorization contains a single request + response.

Resource Owner Password Credentials Grant Request:

The request contains the following parameters:

grant_type	Required. Must be set to password
username	Required. The username of the resource owner, UTF-8 encoded.
password	Required. The password of the resource owner, UTF-8 encoded.
scope	Optional. The scope of the authorization.

OAuth

Resource Owner Password Credentials Grant Response:

The response is a JSON structure containing the access token. The JSON structure looks like this:

```
{ "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

OAuth 2.0 Client Credentials Grant - Requests and Response

- **Client Credentials Grant Request**

The client credentials grant request contains the following parameters:

grant_type	Required. Must be set to client_credentials .
scope	Optional. The scope of the authorization.

- **Client Credentials Grant Response**

The client credentials response contains the following parameters:

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

OAuth

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.

A refresh token should not be included for this type of authorization request.

OAuth in Spring Boot

OAuth2 Roles

- Resource Owner: User
- Client: Application
- Resource Server: API
- Authorization Server: API

OAuth2 Grant Types

- Authorization Code: used with server-side Applications
- Implicit: used with Mobile Apps or Web Applications (applications that run on the user's device)
- Resource Owner Password Credentials: used with trusted Applications, such as those owned by the service itself
- Client Credentials: used with Applications API access

OAuth2 Authorization Server Config

AuthorizationServerConfigurerAdapter and is responsible for generating tokens specific to a client. Suppose, if a user wants to login to amit.com via facebook then facebook auth server will be generating tokens for Amit. In this case, Amit becomes the client which will be requesting for authorization code on behalf of user from facebook - the authorization server. Following is a similar implementation that facebook will be using.

Here, we are using in-memory credentials with client_id as amit-client and CLIENT_SECRET as amit-secret. But you are free to use JDBC implementation too.

@**EnableAuthorizationServer**: Enables an authorization server.AuthorizationServerEndpointsConfigurer defines the authorization and token endpoints and the token services.

OAuth2 Resource Server Config

Resource in our context is the REST API which we have exposed for the crud operation. To access these resources, client must be authenticated. In real-time scenarios, whenever an user tries to access these resources, the user will be asked to provide his authenticity and once the user is authorized then he will be allowed to access these protected resources.

@**EnableResourceServer**: Enables a resource server