SPRING SECURITY USING OAuth 2.0

Spring Security Introduction:

Spring security is the highly customizable authentication and access-control framework. This is the security module for securing spring applications. But this can also be used for non-spring based application with few extra configurations to enable the security features. The main focus of spring security is on Authentication and Authorization:

- Authentication: Process of checking the user, who they claim to be.
- Authorization: Process of deciding whether a user is allowed to perform an activity within the application.

Artifacts Download

The first step towards writing your spring security application is to get the required artifacts for spring security. Here is the dependency requirements for Maven and Gradle build scripts.

Pom.xml for Maven

```
 < dependencies >
 <! -- ... other dependency elements ... -->
< dependency >
          <groupId>org.springframework.security</groupId>
          <artifactId>spring-security-web</artifactId>
          <version>4.0.2.RELEASE</version>
 </dependency>
 <dependency>
          <groupId>org.springframework.security</groupId>
          <artifactId>spring-security-config</artifactId>
          <version>4.0.2.RELEASE</version>
 </dependency>
</dependencies>
```

build.gradle for Gradle

```
dependencies {

	compile 'org.springframework.security:spring-security-web:4.0.2.RELEASE'

	compile 'org.springframework.security:spring-security-config:4.0.2.RELEASE'
}
```

- If you are using the third party authentication models, then you have to include those dependencies in your build file.

OAuth 2.0 INTODUCTION

Introduction:

OAuth is an open authorization protocol, which allows accessing the resources of the resource owner by enabling the client applications on HTTP services such as Facebook, GitHub, etc. It allows sharing of resources stored on one site to another site without using their credentials. It uses username and password tokens instead.

Why we use OAuth 2

- You can use OAuth 2.0 to read data of a user from another application.
- It supplies the authorization workflow for web, desktop applications, and mobile devices.
- It is a server side web app that uses authorization code and does not interact with user credentials.

Features of OAuth 2.0

- OAuth 2.0 is a simple protocol that allows to access resources of the user without sharing passwords.
- It provides user agent flows for running clients application using a scripting language, such as JavaScript. Typically, a browser is a user agent.
- It accesses the data using tokens instead of using their credentials and stores data in online file system of the user such as Google Docs or Dropbox account.

OAuth2 Roles

OAuth defines four roles:

- Resource owner (the User) – an entity capable of granting access to a protected resource (for example end-user).

- Resource server (the API server) – the server hosting the protected resources, capable of accepting responding to protected resource requests using access tokens.

- Client – an application making protected resource requests on behalf of the resource owner and with its authorization.

- Authorization server – the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

OAuth2 Tokens

Tokens are implementation specific random strings, generated by the authorization server and are issued when the client requests them.

- Access Token: Sent with each request, usually valid for a very short life time [an hour e.g.]

- Refresh Token: Mainly used to get a new access token, not sent with each request, usually lives longer than access token.

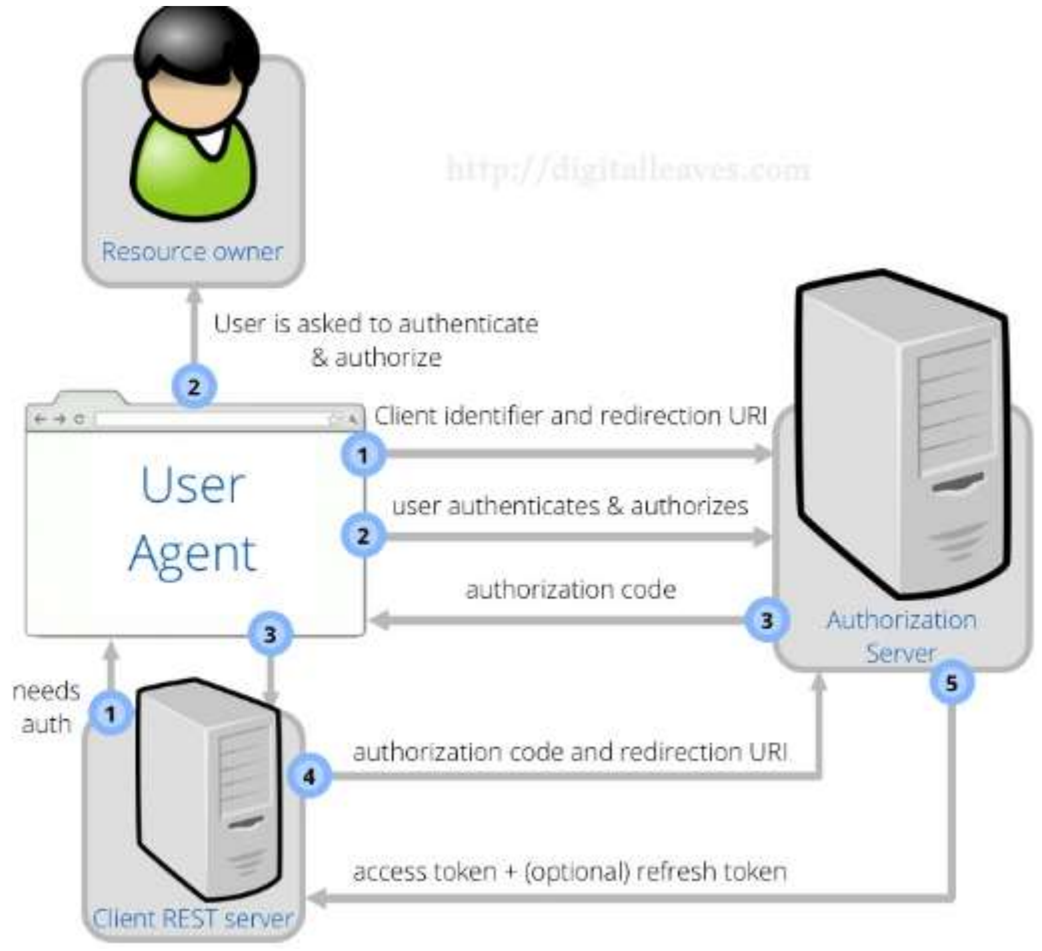
OAuth2 Grant Types

There are different types of scenario were OAuth can be deployed, with different types of clients and different protocol flows for obtaining authorization. OAuth has defined four "grant types" to handle four authorization flows, depending on the kind of client accessing the information and the needs of the information schema.

The grant types defined are:

i) Authorization Code
ii) Implicit
iii) Resource Owner Password Credential
iv) Client Credentials

i) Authorization Code (web apps)



Resource owner
User is asked to authenticate & authorize
2
User Agent
Client identifier and redirection URI
1
user authenticates & authorizes
2
authorization code
3
3
Authorization Server
5
needs auth
1
4
authorization code and redirection URI
access token + (optional) refresh token
Client REST server
http://digitalleaves.com

1.The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint.
The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

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

3. Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.

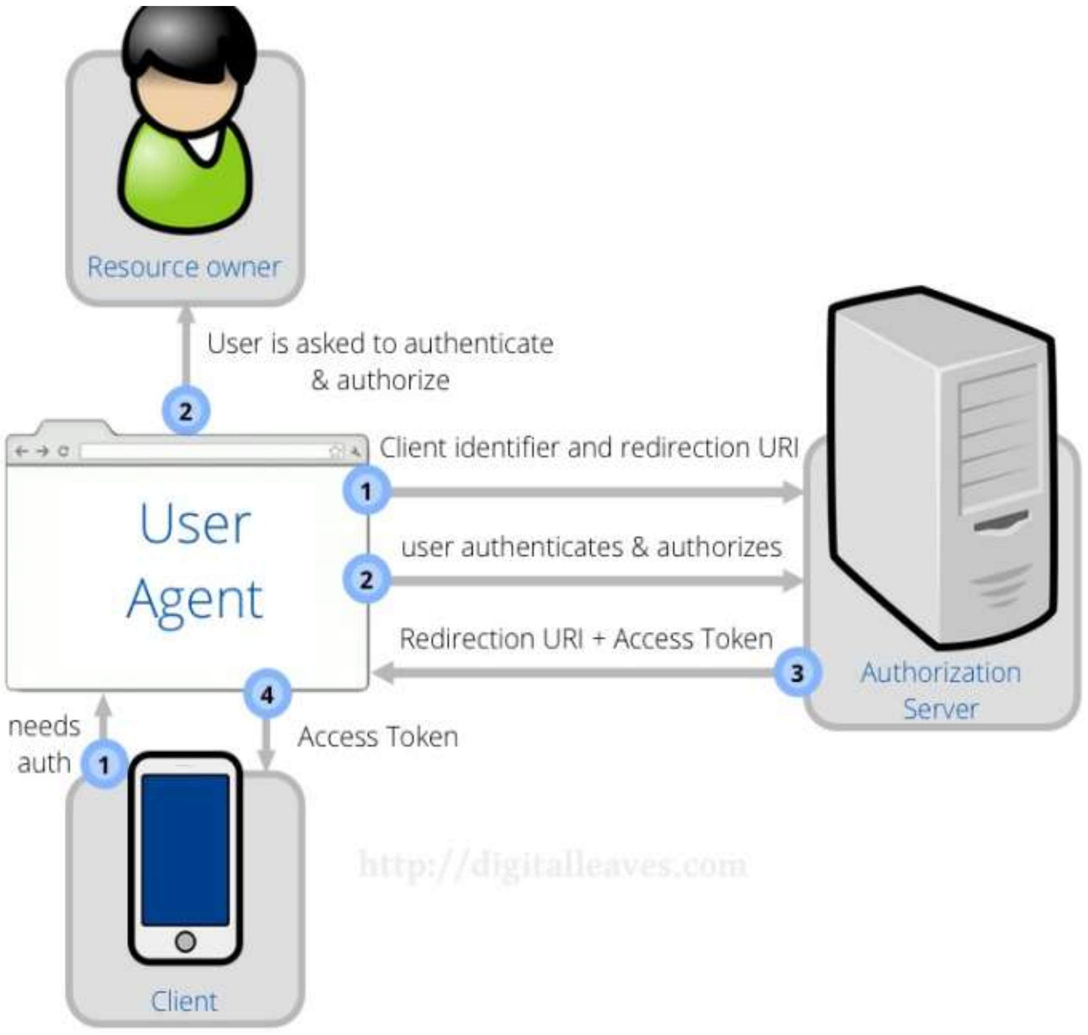
4. The client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.

5. The authorization server authenticates the client, validates the authorization code, and ensures that the redirection URI received matches the URI used to redirect the client in step (3). If valid, the authorization server responds back with an access token and, optionally, a refresh token.

Request:

- grant_type: REQUIRED, Value MUST be set to "authorization_code".
- Code: REQUIRED, The authorization code received from the authorization server.
- redirect_uri: REQUIRED, if the "redirect_uri" parameter was included in the authorization request and their values MUST be identical.
- client_id: REQUIRED, if the client is not authenticating with the authorization server.

ii) Implicit (Client-side web application flow)


Resource owner
User is asked to authenticate & authorize
2
Client identifier and redirection URI
1
User Agent
user authenticates & authorizes
2
Redirection URI + Access Token
3
Authorization Server
4
Access Token
needs auth
1
Client
http://digitalleaves.com

- this grant type is aimed at situations where the OAuth client is the browser. This flow can only provide short-lived tokens, not refresh tokens, and does not require an intermediate authorization code like the Server-side web application flow.

- In this scenario, when the client needs to get authorization from the resource owner, it will redirect the user to the Authorization server, where he is asked to authenticate (if he or she is not already logged in) and then authorize the requested permissions. The request will include a redirect_uri where the client is going to be redirected when/if he or she grants access. For example, for Google services, this request would be:

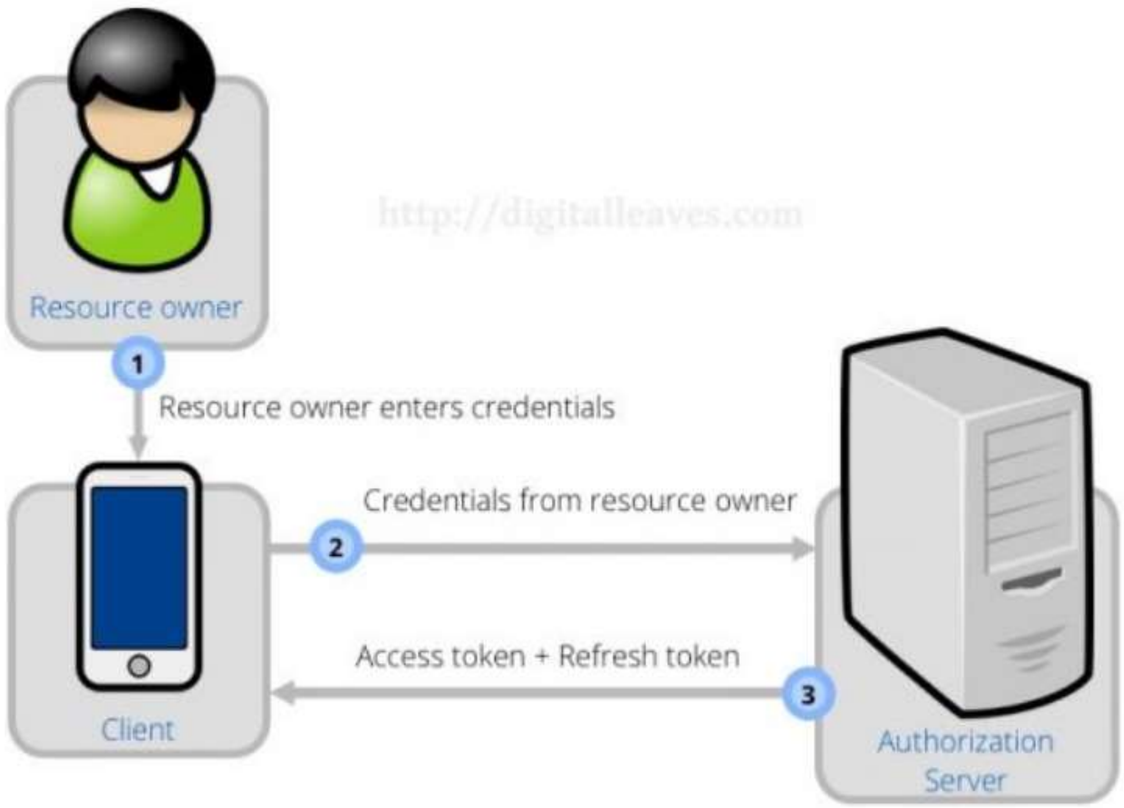
```
https://accounts.google.com/o/oauth2/auth? client_id={app-id}& redirect_uri=http://www.exa
mple.com/oauth_token& scope={permissions}& response_type=token
```

- In this case, we are indicating the client-side web application flow type with the “response_type=token”. After successfully being granted access, the client is redirected to the redirect_uri address, including an access_token that can be used directly by the client to request information or perform operations on behalf of the user:

```
http://www.example.com/oauth_token#access_token=abcde&expires_in=3600
```

- Notice how in this case the access token is separated from the server’s base URL by a ‘#’ sign. There does exist differences in the way in which different services implement the OAuth protocols, so you must be sure to read the API for each service you want to include in your application.

iii) Resource Owner Password Credential



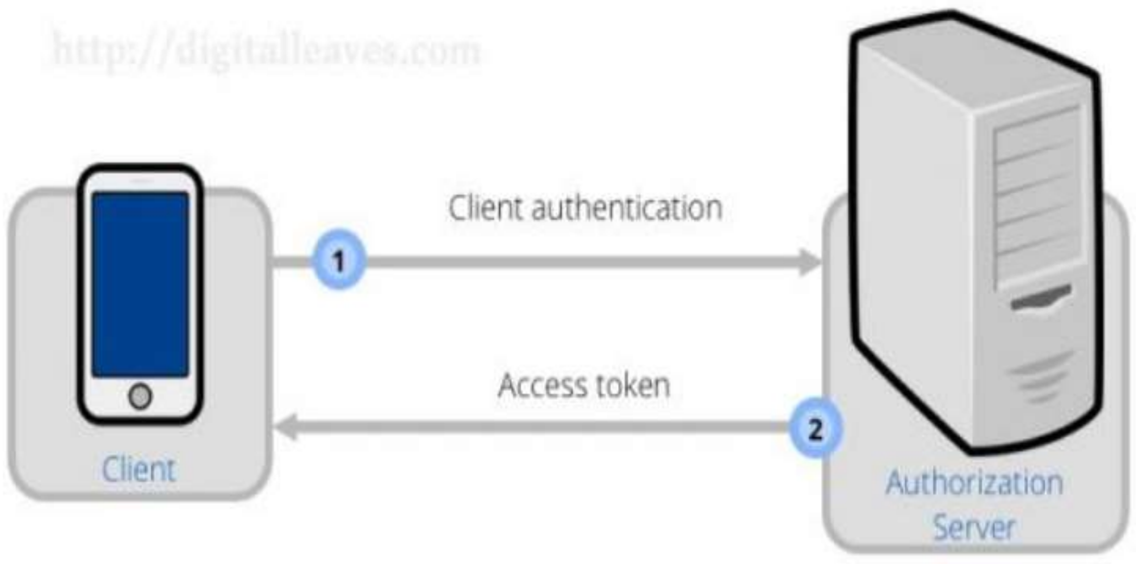
http://digitalleaves.com
Resource owner
1
Resource owner enters credentials
Credentials from resource owner
2
Access token + Refresh token
3
Client
Authorization Server

1. The resource owner provides the client with its username and password.

2. The client requests an access token from the authorization server's token endpoint by including the credentials received from the resource owner. When making the request, the client authenticates with the authorization server.

3. The authorization server authenticates the client and validates the resource owner credentials, and if valid, issues an access token.

IV)Client Credentials



1. The client authenticates with the authorization server and requests an access token from the token endpoint.

2. The authorization server authenticates the client, and if valid, issues an access token.

Spring Security using OAuth2

Project Structure:

Following is the project structure of our Spring Boot Security OAuth2 implementation.

Maven Dependencies:

Pom.xml

Following are the required dependencies.

```
<parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>1.5.8.RELEASE</version>
</parent>
 <dependencies>
             <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-starter-security</artifactId>
                </dependency>
 <dependency>
             <groupId>org.springframework.boot</groupId>
             <artifactId>spring-security-oauth2</artifactId>
         </dependency>
 <dependency>
              <groupId>mysql</groupId>
              <artifactId>mysql-connector-java</artifactId>
                </dependency>
 <dependency>
               <groupId>commons-dbcp</groupId>
               <artifactId>commons-dbcp</artifactId>
                        </dependency>

   </dependencies>
```

OAuth2 Authorization Server Config

Authorization server is the one responsible for verifying credentials and if credentials are OK, providing the tokens[refresh-token as well as access-token].

It also contains information about registered clients and possible access scopes and grant types.

The token store is used to store the token. We will be using an in-memory token store.

@EnableAuthorizationServer enables an Authorization Server (i.e. an AuthorizationEndpoint and a TokenEndpoint) in the current application context. Class AuthorizationServerConfigurerAdapter implements AuthorizationServerConfigurer .which provides all the necessary methods to configure an Authorization server

Eg: if a user wants to login to Amazon.com via facebook then facebook auth server will be generating tokens for Amazon. In this case, Amazon becomes the client which will be requesting for authorization code on behalf of user from facebook

Following is a similar implementation that facebook will be using.

```
@Configuration
@EnableAuthorizationServer
public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {
            static final String CLIEN_ID = "amazon-client";
               static final String CLIENT_SECRET = "amazon-secret";
               static final String GRANT_TYPE = "password";
               static final String AUTHORIZATION_CODE = "authorization_code";
               static final String REFRESH_TOKEN = "refresh_token";
               static final String IMPLICIT = "implicit";
               static final String SCOPE_READ = "read";
               static final String SCOPE_WRITE = "write";
               static final String TRUST = "trust";
               static final int ACCESS_TOKEN_VALIDITY_SECONDS = 1*60*60;
   static final int FREFRESH_TOKEN_VALIDITY_SECONDS = 6*60*60;
               @Autowired
               private TokenStore tokenStore;
             @Autowired
               private AuthenticationManager authenticationManager;
  @Override
               public void configure(ClientDetailsServiceConfigurer configurer) throws Exception {
                           configurer
                                                          .inMemory()
                                                          .withClient(CLIEN_ID)
                                                          .secret(CLIENT_SECRET)
                                                          .authorizedGrantTypes(GRANT_TYPE_PASSWORD, AUTHORIZA
TION_CODE, REFRESH_TOKEN, IMPLICIT )
                                                          .scopes(SCOPE_READ, SCOPE_WRITE, TRUST)
                                                          .accessTokenValiditySeconds(ACCESS_TOKEN_VALIDITY_SECON
DS).
                                                          refreshTokenValiditySeconds(FREFRESH_TOKEN_VALIDITY_SEC
ONDS);
               }
@Override
               public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
                            endpoints.tokenStore(tokenStore)
                                                          .authenticationManager(authenticationManager);
               }
}
```

OAuth2 Resource Server Config

Resource in our context is the REST API which we have exposed for the crud operation. Resources are located on /users/.

To access these resources, client must be authenticated.

Class ResourceServerConfigurerAdapter implements ResourceServerConfigurer pr oviding methods to adjust the access rules and paths that are protected by OAuth2 security.

In real-time scenarios, whenever a 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

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {

        private static final String RESOURCE_ID = "resource_id";

        @Override
        public void configure(ResourceServerSecurityConfigurer resources) {
                resources.resourceId(RESOURCE_ID).stateless(false);
        }

        @Override
        public void configure(HttpSecurity http) throws Exception {
    http.
        anonymous().disable()
        .authorizeRequests()
        .antMatchers("/users/**").authenticated()
        .and().exceptionHandling().accessDeniedHandler(new OAuth2AccessDeniedHandler());
        }
}
```

OAuth2 Security Configuration

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

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

   @Resource(name = "userService")
   private UserDetailsService;

   @Override
   @Bean
   public AuthenticationManager authenticationManagerBean() throws Exception {
      return super.authenticationManagerBean();
   }

   @Autowired
   public void globalUserDetails(AuthenticationManagerBuilder auth) throws Exception {
      auth.userDetailsService(userDetailsService)
            .passwordEncoder(encoder());
   }

   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http
            .csrf().disable()
            .anonymous().disable()
            .authorizeRequests()
            .antMatchers("/api-docs/**").permitAll();
   }

   @Bean
   public TokenStore tokenStore() {
      return new InMemoryTokenStore();
   }

   @Bean
   public BCryptPasswordEncoder encoder(){
      return new BCryptPasswordEncoder();
   }

   @Bean
   public FilterRegistrationBean corsFilter() {
      UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
      CorsConfiguration config = new CorsConfiguration();
      config.setAllowCredentials(true);
      config.addAllowedOrigin("*");
      config.addAllowedHeader("*");
      config.addAllowedMethod("*");
      source.registerCorsConfiguration("/**", config);
      FilterRegistrationBean bean = new FilterRegistrationBean(new CorsFilter(source));
      bean.setOrder(0);
      return bean;
   }
}
```

Default Scripts

Following are the insert statements that are inserted when application starts.

```
INSERT INTO User (id, username, password, salary, age) VALUES (1, 'Alex123',
'$2a$04$I9Q2sDc4QGGg5WNTLmsz0.fvGv3OjoZyj81PrSFyGOqMphqfS2qKu', 3456, 33);
INSERT INTO User (id, username, password, salary, age) VALUES (2, 'Tom234', '
$2a$04$PCIX2hYrve38M7eOcqAbCO9UqjYg7gfFNpKsinAxh99nms9e.8HwK', 7823, 23);
INSERT INTO User (id, username, password, salary, age) VALUES (3, 'Adam', '$2
a$04$I9Q2sDc4QGGg5WNTLmsz0.fvGv3OjoZyj81PrSFyGOqMphqfS2qKu', 4234, 45);
```

References

- https://aaronparecki.com/oauth-2-simplified/#web-server-apps
- https://dzone.com/articles/secure-spring-rest-with-spring-security-and-oauth2
- https://oauth.net/2/ (OATH 2.0 specification)
- http://www.baeldung.com/rest-api-spring-oauth2-angularjs (Spring boot example)