Spring Oauth2 to secure REST

How to create from scratch a REST service with Spring Boot? Secure it using the Oauth2 protocol, using JSON Web Tokens, or JWT

Dig deep on how to secure a REST service using Oauth2 protocol

Creating a new Spring Boot project with Initializr

Fill up the basic information and select the following dependencies: Web, Cloud Oauth2, Security, JPA and H2

Take a look at your dependencies in the `pom.xml` file.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Saying ‘Hello World’ with the RestController

Finally, let’s get our hands dirty. First, we need to provide a public URL that will print “Hello World” when a Get Requests is received. It is extremely easy to create this kind of resource on Spring, all you’ll have to do is annotate a class with `@RestController` and set the URL using `@RequestMapping("/")` annotations. You can even specify the exact type of request that the resource should receive, using `@PostRequest`, `@PutRequest`, `@DeleteRequest` and so on. It is also possible to define the headers, parameters, what kind of data the request consumes and produces. You can take a look at the documentation to get a better understanding.

```
@RestController
public class GeneralController {
   @RequestMapping("/")
   public String home() {
      return "Hello World";
   }
}
```

If you run the application now and call the command `curl localhost:8080/`, you’ll receive a **401 Status, “Full authentication is required to access this resource”**. Even though the resource is already available, it is still closed to external access. Out of the box, the `spring-boot-starter-security` dependency will turn “on” some security configuration:

- It automatically creates a user, called “user” and with a random defined password.
- Ignored (insecure) paths for common static resource locations (`/css/**`, `/js/**`, `/images/**`, `/webjars/**` and `**/favicon.ico`)
- HTTP Basic security for all other endpoints
- Common low-level features (HSTS, XSS, CSRF, caching) provided by Spring Security are on by default

All those options can be turned “off” using the `@EnableWebSecurity` annotation and providing an `@Bean` of type `WebSecurityConfigurerAdapter` that will provide many customization options. So, to access our `home()` endpoint, we’ll need to authenticate a valid user. Firlet’slets define a static password for the user. On the `application.properties` file add:

```
security.user.password=password
```

Now run the application again and call the endpoint with the username and password:

```
curl user:password@localhost:8080/
```

Finally, we’ll receive the classic “Hello World”!

The WebSecurityConfigurerAdapter

Now it is time to take control of the security configuration. We'll turn off the default configuration using the `@EnableWebSecurity` annotation and we'll also provide a `WebSecurityConfigurerAdapter` `@Bean` that will give us the base to configure the authentication process. The most important methods that can be overridden are:

- `protected void configure(HttpSecurity http)`: It allows configuring web-based security for specific http requests.
- `protected void configure(AuthenticationManagerBuilder auth)`: The place to configure the default `authenticationManager` @Bean.
- `public void configure(WebSecurity web)`: Override this method to expose the default `AuthenticationManager` as a Bean.
- `protected UserDetailsService userDetailsService()`: Allows modifying and accessing the `UserDetailsService`, a core interface which loads user-specific data.

It is also important to pay attention to the adapter filtering order. On Spring, the filter with the greatest value takes precedence over the lowest ones. On Spring Boot 1.5.+, the default order of the adapter is 100, but you can change it using the @Order annotation. Try to keep that information in mind, it will be relevant when your security configuration becomes more complex. You could take a look at the source code to get a deep understanding of how it works. Basically, the adapter provides some configuration methods that can be customized through overriding.

On our project, we'll use our `WebSecurityConfigurerAdapter` to turn "off" the default security options and to configure users and an authentication process. At first, we'll also close all accesses to our endpoints. We'll correctly configure the accesses later on.

```
@Configuration
@EnableWebSecurity( debug = true )
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http
            .formLogin().disable() // disable form authentication
            .anonymous().disable() // disable anonymous user
            .authorizeRequests().anyRequest().denyAll(); // denying all access
   }

   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws Exception {
      auth.inMemoryAuthentication() // creating user in memory
            .withUser("user")
               .password("password").roles("USER")
            .and().withUser("admin")
               .password("password").authorities("ROLE_ADMIN");
   }
```

```
    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        // provides the default AuthenticationManager as a Bean
        return super.authenticationManagerBean();
    }
}
```

Now, if you call `curl localhost:8080/`, even with the correct credentials, you'll receive a 403 Status, "Access Denied". Our next step will be to give access to certain resources for specific users through an `@EnableResourceServer` annotation on a `ResourceServerConfigurerAdapter` bean and also provide an authentication endpoint, using JWT tokens through a `@EnableAuthorizationServer` and a `AuthorizationServerConfigurerAdapter` bean.

Authorizing and granting Oauth2 Tokens

If we annotate a configuration class with `@EnableAuthorizationServer`, and provide the properties `security.oauth2.client.client-id` and `security.oauth2.client.client-secret`, Spring will give us an authentication server, providing standard Oauth2 tokens at the endpoint `/oauth/token`. However, since we had turned off the default configuration, we must provide an `AuthorizationServerConfigurerAdapter` bean to correctly setup the authorization server. Like on the `WebSecurityConfigurerAdapter` , the `AuthorizationServerConfigurerAdapter` relies on some methods that can be overridden to setup the configuration. The main methods are:

- `void configure(AuthorizationServerSecurityConfigurer security)`: Configure the security of the Authorization Server, which means in practical terms the /oauth/token endpoint.
- `void configure(ClientDetailsServiceConfigurer clients)`: Configure the `ClientDetailsService`, declaring individual clients and their properties.
- `void configure(AuthorizationServerEndpointsConfigurer endpoints)`: Configure the non-security features of the Authorization Server endpoints, like token store, token customizations, user approvals and grant types.

However, before we start with the adapter, let's change some things on `application.properties`. Open the file and remove everything, then and add the following line:

```
security.oauth2.resource.id=oauth2_id
```

Now we can create the AuthorizationServerConfigurerAdapter.

```
@Configuration
@EnableAuthorizationServer
public class AuthorizationConfig extends AuthorizationServerConfigurerAdapter {

    private int accessTokenValiditySeconds = 10000;
    private int refreshTokenValiditySeconds = 30000;

    @Value("${security.oauth2.resource.id}")
```

```
    private String resourceId;

    @Autowired
    private AuthenticationManager authenticationManager;

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

    @Override
    public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception {
        oauthServer
                // we're allowing access to the token only for clients with 'ROLE_TRUSTED_CLIENT' authority
                .tokenKeyAccess("hasAuthority('ROLE_TRUSTED_CLIENT')")
                .checkTokenAccess("hasAuthority('ROLE_TRUSTED_CLIENT')");
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
                .withClient("trusted-app")
                    .authorizedGrantTypes("client_credentials", "password", "refresh_token")
                    .authorities("ROLE_TRUSTED_CLIENT")
                    .scopes("read", "write")
                    .resourceIds(resourceId)
                    .accessTokenValiditySeconds(accessTokenValiditySeconds)
                    .refreshTokenValiditySeconds(refreshTokenValiditySeconds)
                    .secret("secret");
    }
}
```

Notice that we're using the authenticationManager defined as a @Bean on the SecurityConfig and also that, for now, we're relying on only one client. Our authorization server is already working, however, we can improve it a lot. But first, let's test it as it is. Run the application and call the following curl command:

```
curl trusted-app:secret@localhost:8080/oauth/token -d "grant_type=password&username=user&password=password"
```

The authorization server will provide us with a token, a refresh token, and some information:

```
{
  "access_token": "dcd1afeb-e378-423e-8ccb-7660f4ec8b82",
```

```
  "token_type": "bearer",
  "refresh_token": "a73cd360-c28f-4e3c-93bf-0c54493cc1ef",
  "expires_in": 9092,
  "scope": "read write"
}
```

Our token is working, however, one of our objectives is that the server must provide JSON Web Tokens, which contains some advantages over the standard one. The JWT contains all information necessary to validate the user on the token itself, meaning that it doesn't depend on a back-end storage to conclude the authentication process. Nonetheless, you must pay special attention to the revocation process of the JWT, considering that it won't be saved in a database.

To provide the JWT, we'll need to create a `JwtTokenStore`, a `JwtAccessTokenConverter` and a `DefaultTokenServices` bean, and wire all that to the `AuthorizationServerEndpointsConfigurer`. So, add the following lines in our `AuthorizationConfig` class:

```
@Bean
public TokenStore tokenStore() {
    return new JwtTokenStore(accessTokenConverter());
}

@Bean
public JwtAccessTokenConverter accessTokenConverter() {
    JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
    converter.setSigningKey("abcd");
    return converter;
}

@Bean
@Primary
public DefaultTokenServices tokenServices() {
    DefaultTokenServices defaultTokenServices = new DefaultTokenServices();
    defaultTokenServices.setTokenStore(tokenStore());
    defaultTokenServices.setSupportRefreshToken(true);
    defaultTokenServices.setTokenEnhancer(accessTokenConverter());
    return defaultTokenServices;
}
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
    endpoints
            .authenticationManager(this.authenticationManager)
            .tokenServices(tokenServices())
            .tokenStore(tokenStore())
            .accessTokenConverter(accessTokenConverter());
```

```
}
```

If you run the same curl command you'll receive a JWT token, as we wanted from the start.

```
{
 "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsib2F1dGgyX2lkIl0sInVzZXJfbmFtZSI6InVzZXIiLCJzY29wZSI6WyJyZWFkIiwid3JpdGUiXSwiZXhwIjoxNDk4MjA3NDc3LCJhdXRob3JpdGllcyI6WyJST0xFX1VTRVIiXSwianRpIjoiMWY2NGRmMDQtZTk1Ni00OWU1LWJiYWMtMjI1MmVjYWU1NTE5IiwiY2xpZW50X2lkIjoidHJ1c3RlZC1hcHAifQ.CZNwij0IzYkPC67u1RcBHDUQL7r6Teb-tS_hqlc4lkE",
 "token_type": "bearer",
 "refresh_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsib2F1dGgyX2lkIl0sInVzZXJfbmFtZSI6InVzZXIiLCJzY29wZSI6WyJyZWFkIiwid3JpdGUiXSwiYXRpIjoiMWY2NGRmMDQtZTk1Ni00OWU1LWJiYWMtMjI1MmVjYWU1NTE5IiwiZXhwIjoxNTAwNzU2Mjc3LCJhdXRob3JpdGllcyI6WyJST0xFX1VTRVIiXSwianRpIjoiMDU3NTEzNTItZDkxNi00NzcyLTliYTktZTFmYmVhNGJkMzhkIiwiY2xpZW50X2lkIjoidHJ1c3RlZC1hcHAifQ.NzuwJyyAbr3Md6_d9M2kmhzgtjD5dSgezT_NOLm0z-E",
 "expires_in": 43199,
 "scope": "read write",
 "jti": "1f64df04-e956-49e5-bbac-2252ecae5519"
}
```

It is possible to renew the token using a refresh token, as long as the client and the authorization support it. To do it, a refresh_token's grant_type and the refresh_token itself must be sent to the server. It is also important to authenticate the client. You can test it with the command:

```
curl trusted-app:secret@localhost:8080/oauth/token -d "grant_type=refresh_token&refresh_token=[REFRESH-TOKEN]"
```

Using an asymmetric key

Note that, for the sake of simplicity, we used a symmetric key in the JwtAccessTokenConverter. This isn't the best approach for a production environment. We should create an asymmetric key and use it to sign the converter. You can use Keytool to produce a key pair, running the following command on the `/src/main/resources/` directory:

```
keytool -genkeypair -alias mykeys -keyalg RSA -keypass mypass -keystore mykeys.jks -storepass mypass
```

Finally, the `accessTokenConverter` must be edited:

```
@Bean
public JwtAccessTokenConverter accessTokenConverter() {
   JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
   KeyStoreKeyFactory keyStoreKeyFactory =
```

```
            new KeyStoreKeyFactory(
                    new ClassPathResource("mykeys.jks"),
                    "mypass".toCharArray());
    converter.setKeyPair(keyStoreKeyFactory.getKeyPair("mykeys"));
    return converter;
}
```

Authorizing Access to Rest Resources

We've already got our token provider and now it is time to use it. Enable Oauth2 authentication on Spring is pretty straight forward, all you need to do is annotate a configuration class with `@EnableResourceServer`. The annotation enables a Spring Security filter that authenticates requests via an incoming OAuth2 token. To customize the security setting we'll need to provide a `ResourceServerConfigurerAdapter` bean. So, let's get to it:

```
@Configuration
@EnableResourceServer
public class ResourceConfig extends ResourceServerConfigurerAdapter {
    @Value("${security.oauth2.resource.id}")
    private String resourceId;

    // The DefaultTokenServices bean provided at the AuthorizationConfig
    @Autowired
    private DefaultTokenServices tokenServices;

    // The TokenStore bean provided at the AuthorizationConfig
    @Autowired
    private TokenStore tokenStore;

    // To allow the rResourceServerConfigurerAdapter to understand the token,
    // it must share the same characteristics with AuthorizationServerConfigurerAdapter.
    // So, we must wire it up the beans in the ResourceServerSecurityConfigurer.
    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources
                .resourceId(resourceId)
                .tokenServices(tokenServices)
                .tokenStore(tokenStore);
    }
}
```

Observe that we had wired the `TokenStore` and the `DefaultTokenServices`, beans that were defined back on the `AuthorizationServerConfigurerAdapter`. Both adapters must share the same logic on how to create and extract the token in order for the authentication process to work.

Initially, we'll work with two resource endpoints, that weren't declared yet:`/api/hello` and`/api/admin`. Now, let's configure the `HttpSecurity` to control the access to specific resources.

```
@Override
public void configure(HttpSecurity http) throws Exception {
   http
        .requestMatcher(new OAuthRequestedMatcher())
        .csrf().disable()
        .anonymous().disable()
        .authorizeRequests()
        .antMatchers(HttpMethod.OPTIONS).permitAll()
        // when restricting access to 'Roles' you must remove the "ROLE_" part role
        // for "ROLE_USER" use only "USER"
        .antMatchers("/api/hello").access("hasAnyRole('USER')")
        .antMatchers("/api/admin").hasRole("ADMIN")
        // restricting all access to /api/** to authenticated users
        .antMatchers("/api/**").authenticated();
}

private static class OAuthRequestedMatcher implements RequestMatcher {
   public boolean matches(HttpServletRequest request) {
      // Determine if the resource called is "/api/**"
      String path = request.getServletPath();
      if ( path.length() >= 5 ) {
        path = path.substring(0, 5);
        boolean isApi = path.equals("/api/");
        return isApi;
      } else return false;
   }
}
```

Notice that the `/api/hello` endpoint can only be accessed by users with role `"USER"`, while the `/api/admin` can only be accessed by users with `"ADMIN"` role. We're also restricting the access to all resources derived from `/api/**` only to authenticated users. Another cool configuration that we're using is the `.requestMatcher(new OAuthRequestedMatcher())`, which restricts all options defined on this `HttpSecurity` only to requests directed at URIs with the path `/api/`, guaranteeing that the `HttpSecurity` defined at `ResourceConfig` will prevail over any other endpoint remaining.
Now, the resource endpoints must be added in the `GeneralController` class:.

```
@RestController
public class GeneralController {
   @GetMapping("/")
      public RestMsg hello(){
      return new RestMsg("Hello World!");
   }
```

```
    @GetMapping("/api/test")
        public RestMsg apitest(){
        return new RestMsg("Hello apiTest!");
    }

    @GetMapping(value = "/api/hello", produces = "application/json")
    public RestMsg helloUser(){
        // The authenticated user can be fetched using the SecurityContextHolder
        String username = SecurityContextHolder.getContext().getAuthentication().getName();
        return new RestMsg(String.format("Hello '%s'!", username));
    }

    @GetMapping("/api/admin")
    // If a controller request asks for the Principal user in
    // the method declaration Spring security will provide it.
    public RestMsg helloAdmin(Principal principal){
        return new RestMsg(String.format("Welcome '%s'!", principal.getName()));
    }

    // A helper class to make our controller output look nice
    public static class RestMsg {
        private String msg;
        public RestMsg(String msg) {
            this.msg = msg;
        }
        public String getMsg() {
            return msg;
        }
        public void setMsg(String msg) {
            this.msg = msg;
        }
    }
}
```

We also need to edit some things in the `SecurityConfig` class. First of all, its filtering order must be changed to a lower one, allowing that the configurations defined on `ResourceConfig` class take precedence over it. We should also change the `HttpSecurity` configuration defined in the `SecurityConfig`. Let's add an `httpBasic` authentication and restrict all access only to authenticated users.

```
@Configuration
@EnableWebSecurity( debug = true )
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
```

```
        http
                .formLogin().disable() // disable form authentication
                .anonymous().disable() // disable anonymous user
                .httpBasic().and()
                // restricting access to authenticated users
                .authorizeRequests().anyRequest().authenticated();
    }
    // .....
}
```

Since the `HttpSecurity` defined on `ResourceConfig` takes precedence over the one defined on `SecurityConfig`, and also considering that the configuration on the `ResourceConfig` is restricted to requests directed at `/api/**`, the `HttpSecurity`defined on the `SecurityConfig` class won't be a problem for our resources security.

Accessing the Protected Resources

To access any endpoint defined on our server, the user needs to be authenticated. However, we're using two different types of authentication: Oauth2 authentication using JSON Web Token for all requests on `/api/**` and the classic `httpBasic`, with user and password for any other request. Let's test our protection level with some curl commands (the following commands will depend on the installation of the ./JQ).

To access `/api/hello`:

```
curl -H "Authorization: Bearer $(curl trusted-app:secret@localhost:8080/oauth/token -d "grant_type=password&username=user&password=password" | jq --raw-output .'access_token')" localhost:8080/api/hello | jq
```

To access `/api/admin`:

```
curl -H "Authorization: Bearer $(curl trusted-app:secret@localhost:8080/oauth/token -d "grant_type=password&username=admin&password=password" | jq --raw-output .'access_token')" localhost:8080/api/admin | jq
```

If you try to access those endpoints with an authentication using an `httpBasic` authentication, the process will fail:

```
curl user:password@localhost:8080/api/hello | jq
```

However, if you use this authentication process on the `/` endpoint, it will work:

```
curl user:password@localhost:8080/ | jq
```

By the same logic, calling this path using a token authentication will also fail:

Summing-Up

This was way longer than I expected and if you've managed to get till here you are a code warrior! I tried my best to provide as much information as I could, however, some interesting points were left behind. If you want to know even more, you can read the extensive Spring Security Guide, the Oauth2 Developer Guide and the Security Session on Spring Boot. Be advised, it will take some time for you to read all those guides combined, but you'll have an awesome understanding of the security process by the end.

```
curl -H "Authorization: Bearer $(curl trusted-app:secret@localhost:8080/oauth/token -d "grant_type=password&username=user&password=password" | jq --raw-output .'access_token')" localhost:8080/ | jq
```