Spring Security

Spring Security provides a comprehensive security solution for J2EE-based enterprise software applications

Contents

Introduction

- The basic authentication using a username and a password to a complex one such as two-factor authentication using tokens and OTP’s. Also, we can use various databases – both relational and non-relational, use various password encoders, lock malicious users out of their accounts, and so on.

Overview

- Spring Security is a powerful and highly customizable authentication and access-control framework.
- It is the de-facto standard for securing Spring-based applications.
- Spring Security is a framework that focuses on providing both authentication and authorization to Java applications.
- The real power of Spring Security is found in how easily it can be extended to meet custom requirements

Spring Security 5.2.4.RELEASE

- Bean creation error for org.springframework.security.filterChains comes when you are using Spring version higher than 3.1 and have not added dependencies manually for spring-aop, spring-jdbc, spring-tx and spring-expressions in your pom.xml.
- https://repo1.maven.org/maven2/org/springframework/
- Add below entries in Spring context. We want to protect two REST endpoints (**helloworld** & **goodbye**). Adjust XSD version according to Spring version.

Installation or Setup

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>3.1.0.RELEASE</version>
</dependency>
<dependency>
     <groupId>org.springframework.security</groupId>
     <artifactId>spring-security-config</artifactId>
     <version>3.1.0.RELEASE</version>
<dependency>
<dependency>
   <groupId>org.glassfish.web</groupId>
   <artifactId>javax.servlet.jsp.jstl</artifactId>
    <version>1.2.1</version>
 </dependency>
```

```
<?xml version="1.0" encoding="UTF-8"?>
   <beans    xmlns="http://www.springframework.org/schema/beans"              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:security="http://www.springframework.org/schema/security"          xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
            http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.1.xsd
 http://www.springframework.org/schema/security                http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <security:http auto-config='true' create-session="never">
       <security:intercept-url pattern="/helloworld/**" access="ROLE_USER" />
       <security:intercept-url pattern="/goodbye/**" access="ROLE_ADMIN" />
       <security:intercept-url pattern="/**" access="IS_AUTHENTICATED_ANONYMOUSLY" />
       <security:http-basic />
    </security:http>
    <security:authentication-manager>
       <security:authentication-provider>
          <security:user-service>
             <security:user name="username1" password="password1"              authorities="ROLE_USER" />
             <security:user name="username2" password="password2"              authorities="ROLE_ADMIN" />
          </security:user-service>
       </security:authentication-provider>
    </security:authentication-manager>
 </beans>
```

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
 </filter>
 <filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
 </filter-mapping>
 <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
 <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:security-context.xml</param-value>
 </context-param>
```

Spring-Security using spring-boot and JDBC Authentication

- Suppose you want to prevent unauthorized users to access the page then you have to put barrier to them by authorizing access.
- We can do this by using spring-security which provides basic authentication by securing all HTTP end points.
- For that you need to add spring-security dependency to your project, in maven we can add the dependency as:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
http.csrf().disable().authorizeRequests().anyRequest().authenticated().and().httpBasic().and()
.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
 String password = passwordEncoder().encode("aaaAAA123");
 String adminpassword = passwordEncoder().encode("admin@123");
auth.inMemoryAuthentication().withUser("tarkesh").password(password).roles("USER").and()
                        .withUser("admin").password(adminpassword).roles("ADMIN")
                        //credentialsExpired(false)
                        //.accountExpired(true).accountLocked(true)
.authorities("WRITE_PRIVILEGES", "READ_PRIVILEGES").roles("ADMIN");
//auth.jdbcAuthentication().dataSource(datasource).passwordEncoder(passwordEncoder());
}
@Bean
public PasswordEncoder passwordEncoder() {
    PasswordEncoder encoder = new BCryptPasswordEncoder();
    return encoder;
 }
}
```

```
@Configuration @Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
 @Autowired
 DataSource datasource;
 @Override
protected void configure(HttpSecurity http) throws Exception {
      http.authorizeRequests().anyRequest().fullyAuthenticated().and().formLogin().loginPage("/login")
          .failureUrl("/login?error").permitAll().and().logout().logoutUrl("/logout")
          .logoutSuccessUrl("/login?logout").permitAll().and().csrf();
 }
 @Override protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(datasource).passwordEncoder(passwordEncoder());
 }
 @Bean public PasswordEncoder passwordEncoder() {
    PasswordEncoder encoder = new BCryptPasswordEncoder();
    return encoder;
 }}
```

Configuration	Description
@Configuration	Indicates that the class can be used by the Spring IoC container as a source of bean definitions
@Order (SecurityProperties.ACCESS_OVERRIDE_ORDER)	Override the access rules without changing any other autoconfigured features. Lower values have higher priority
WebSecurityConfigurerAdapter	The SecurityConfig class extends and overrides a couple of its methods to set some specifics of the security configuration.
@Autowired of DataSource	Provide factory for connections to the physical data source.
configure(HttpSecurity)	Overridden method defines which URL paths should be secured and which should not.
.authorizeRequests().anyRequest() .fullyAuthenticated()	Indicates to spring that all request to our application requires to be authenticated.
.formLogin()	Configures a form based login
.loginPage("/login").failureUrl("/login?error").permitAll()	Specifies the location of the log in page and all users should be permitted to access the page.
.logout().logoutUrl("/logout") .logoutSuccessUrl("/login?logout").permitAll()	The URL to redirect to after logout has occurred. The default is /login?logout
.csrf()	Used to prevent Cross Site Request Forgery, CSR Fprotection is enabled (default
configure(AuthenticationManagerBuilder){}	Overridden method to define how the users are authenticated.
.jdbcAuthentication().dataSource(datasource)	Indicates to spring that we are using JDBC authentication
.passwordEncoder(passwordEncoder())	Indicates to spring that we are using a password encoder to encode our passwords. (A bean is created to return the choice of password Encoder, we are using BCrypt in this case)

spring security by default looks

```
create table users (
   username varchar(50) not null primary key,
   password varchar(255) not null,
   enabled boolean not null) ;
create table authorities (
   username varchar(50) not null,
   authority varchar(50) not null,
   foreign key (username) references users (username),
   unique index authorities_idx_1 (username, authority));
```

Queries

```
INSERT INTO authorities(username,authority) VALUES ('user', 'ROLE_ADMIN');

INSERT INTO users(username,password,enabled) VALUES('user',
'$2a$10$JvqOtJaDys0yoXPX9w47YOqu9wZr/PkN1dJqjG9HHAzMyu9EV1R4m',
'1');
```

The username in our case is user and the password is also user encrypted with BCrypt algorithm Finally, Configure a datasource in the ***application.properties*** for spring boot to use:

```
spring.datasource.url = jdbc:mysql://localhost:3306/spring
spring.datasource.username = root

spring.datasource.password = aaaAAA123
```

WEB-INF/spring/security.xml

```
<b:beans   xmlns="http://www.springframework.org/schema/security"
xmlns:b="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security.xsd ">
   <http />
   <user-service>
     <user name="tarkeshwar" password="aaaAAA123" authorities="ROLE_USER" />
   </user-service>
 </b:beans>
```

WEB-INF/web.xml

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
 </filter>
 <filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
 </filter-mapping>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>     /WEB-INF/spring/*.xml   </param-value>
 </context-param>
 <listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
 </listener>
```

Displaying user name

Adding jstl tag after the "Hello", that print the username

index.jsp

```
<h1>Hello <c:out value="${
pageContext.request.remoteUser}" />!!
</h1>
```

Logging out

*Adding form, input tags after "**Hello user name**", that submitting generated logging out url **/logout** from spring security.*

```
<h1>Hello <c:out value="${pageContext.request.remoteUser}" />!!</h1>
<form action="/logout" method="post">
   <input type="submit" value="Log out" />
 <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}" />
 </form>
```

When you successfully log out, you see the auto generated login page again. Because of you are not authenticated now

Spring Security config with java (not XML)

Typical database backed, annotation base spring security setup.

1) configureGlobal() configure the auth object.
2) The later two SQLs may be optional.
3) configure() method tells spring mvc how to authenticate request
4) some url we do not need to authenticate
5) others will redirect to /login if not yet authenticated.

```
@Configuration
public class AppSecurityConfig extends WebSecurityConfigurerAdapter {
 @Autowired
DataSource dataSource;
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
auth.jdbcAuthentication().dataSource(dataSource) .passwordEncoder(new BcryptPasswordEncoder()).
usersByUsernameQuery("select username,password, enabled from users where
username=?") .authoritiesByUsernameQuery("select username, role from user_roles where username=?");
 }
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable();
                                                    http.authorizeRequests().antMatchers(".resources/**",
"/public/**") .permitAll().anyRequest().authenticated().and().formLogin()
.loginPage("/login").permitAll().and().logout().permitAll();
    } }
```

Demo home.html

```
<!DOCTYPE html>
<html xmlns = "http://www.w3.org/1999/xhtml" xmlns:th = "http://www.thymeleaf.org"
   xmlns:sec = "http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
   <head>
      <title>Spring Security Example</title>
   </head>
   <body>
      <h1>Welcome!</h1>
      <p>Click <a th:href = "@{/hello}">here</a> to see a greeting.</p>
   </body>
</html>
```

hello.html

```
<!DOCTYPE html>
<html xmlns = "http://www.w3.org/1999/xhtml" xmlns:th = "http://www.thymeleaf.org"
   xmlns:sec = "http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
   <head>
      <title>Hello World!</title>
   </head>
   <body>
      <h1>Hello world!</h1>
   </body>
</html>
```

mvc configuration

```
@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {
  @Override
  public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/home").setViewName("home");
    registry.addViewController("/").setViewName("home");
    registry.addViewController("/hello").setViewName("hello");
    registry.addViewController("/login").setViewName("login");
  }
}
```

mvc configuration

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .authorizeRequests()
        .antMatchers("/", "/home").permitAll()
        .anyRequest().authenticated()
        .and()
      .formLogin()
        .loginPage("/login")
        .permitAll()
        .and()
        .logout()
        .permitAll();
  }
  @Autowired
  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth
      .inMemoryAuthentication()
      .withUser("user").password("password").roles("USER");
  }
}
```

login.html

```
<!DOCTYPE html>
<html xmlns = "http://www.w3.org/1999/xhtml" xmlns:th = "http://www.thymeleaf.org"
  xmlns:sec = "http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
  <head>
    <title>Spring Security Example </title>
  </head>
  <body>
    <div th:if = "${param.error}">
       Invalid username and password.
    </div>
    <div th:if = "${param.logout}">
       You have been logged out.
    </div>
    <form th:action = "@{/login}" method = "post">
       <div>
          <label> User Name : <input type = "text" name = "username"/> </label>
       </div>
       <div>
          <label> Password: <input type = "password" name = "password"/> </label>
       </div>
       <div>
          <input type = "submit" value = "Sign In"/>
       </div>
    </form>
  </body>
</html>
```

hello.html

```
<!DOCTYPE html>
<html xmlns = "http://www.w3.org/1999/xhtml" xmlns:th =
"http://www.thymeleaf.org"
   xmlns:sec = "http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
   <head>
      <title>Hello World!</title>
   </head>
   <body>
      <h1 th:inline = "text">Hello [[${#httpServletRequest.remoteUser}]]!</h1>
      <form th:action = "@{/logout}" method = "post">
         <input type = "submit" value = "Sign Out"/>
      </form>
   </body>
</html>
```

Login Error snapshot

OAuth2 with JWT

- Authorization Server is a supreme architectural component for Web API Security. The Authorization Server acts a centralization authorization point that allows your apps and HTTP endpoints to identify the features of your application.
- Resource Server is an application that provides the access token to the clients to access the Resource Server HTTP Endpoints. It is collection of libraries which contains the HTTP Endpoints, static resources, and Dynamic web pages.
- OAuth2 is an authorization framework that enables the application Web Security to access the resources from the client. To build an OAuth2 application, we need to focus on the Grant Type (Authorization code), Client ID and Client secret.

JWT Token

- JWT Token is a JSON Web Token, used to represent the claims secured between two parties. **www.jwt.io/**.
- An OAuth2 application that enables the use of Authorization Server, Resource Server with the help of a JWT Token.
- we need to add the following dependencies in our build configuration file.
- Maven users can add the following dependencies in your **pom.xml** file

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-jwt</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-test</artifactId>
  <scope>test</scope>
</dependency>
```

pom.xml

Gradle Dependency

- Gradle users can add the following dependencies in the build.gradle file.

```
compile('org.springframework.boot:spring-boot-starter-security')
compile('org.springframework.boot:spring-boot-starter-web')
testCompile('org.springframework.boot:spring-boot-starter-test')
testCompile('org.springframework.security:spring-security-test')
compile("org.springframework.security.oauth:spring-security-oauth2")
compile('org.springframework.security:spring-security-jwt')
compile("org.springframework.boot:spring-boot-starter-jdbc")
compile("com.h2database:h2:1.4.191")
```

Description

- **Spring Boot Starter Security** − Implements the Spring Security
- **Spring Security OAuth2** − Implements the OAUTH2 structure to enable the Authorization Server and Resource Server.
- **Spring Security JWT** − Generates the JWT Token for Web security
- **Spring Boot Starter JDBC** − Accesses the database to ensure the user is available or not.
- **Spring Boot Starter Web** − Writes HTTP endpoints.
- **H2 Database** − Stores the user information for authentication and authorization.

add the @EnableAuthorizationServer and @EnableResourceServer annotation to act as an Auth server and Resource Server in the same application.

Websecurityapp Application.java

```
@SpringBootApplication
@EnableAuthorizationServer
@EnableResourceServer
@RestController
public class WebsecurityappApplication {
   public static void main(String[] args) {
      SpringApplication.run(WebsecurityappApplication.class, args);
   }
   @RequestMapping(value = "/products")
   public String getProductName() {
      return "Honey";
   }
}
```

POJO class

```
public class UserEntity {
   private String username;
   private String password;
   private Collection<GrantedAuthority> grantedAuthoritiesList =
new ArrayList<>();
}
public Record UserEntity<T>(String username, String password,
Collection<GrantedAuthority> grantedAuthoritiesList){
}
```

CustomUser.java

- The **CustomUser** class that extends the **org.springframework.security.core.userdetails.User** class for Spring Boot authentication.

```
public class CustomUser extends User {
   private static final long serialVersionUID = 1L;
   public CustomUser(UserEntity user) {
      super(user.getUsername(), user.getPassword(), user.getGrantedAuthoritiesList());
   }
}
```

You can create the **@Repository** class to read the User information from the database and send it to the Custom user service and also add the granted authority **“ROLE_SYSTEMADMIN”**.

OauthDao.java

```
@Repository
public class OAuthDao {
  @Autowired
  private JdbcTemplate jdbcTemplate;
  public UserEntity getUserDetails(String username) {
    Collection<GrantedAuthority> grantedAuthoritiesList = new ArrayList<>();
    String userSQLQuery = "SELECT * FROM USERS WHERE USERNAME=?";
    List<UserEntity> list = jdbcTemplate.query(userSQLQuery, new String[] { username },
      (ResultSet rs, int rowNum) -> {
      UserEntity user = new UserEntity();
      user.setUsername(username);
      user.setPassword(rs.getString("PASSWORD"));
      return user;
    });
    if (list.size() > 0) {
      GrantedAuthority grantedAuthority = new SimpleGrantedAuthority("ROLE_SYSTEMADMIN");
      grantedAuthoritiesList.add(grantedAuthority);
      list.get(0).setGrantedAuthoritiesList(grantedAuthoritiesList);
      return list.get(0);
    }
    return null;
  }
}
```

CustomDetailsService.java

```
@Service
public class CustomDetailsService implements UserDetailsService {
  @Autowired
  OAuthDao oauthDao;
  @Override
  public CustomUser loadUserByUsername(final String username) throws
UsernameNotFoundException {
      UserEntity userEntity = null;
      try {
        userEntity = oauthDao.getUserDetails(username);
        CustomUser customUser = new CustomUser(userEntity);
        return customUser;
      } catch (Exception e) {
        e.printStackTrace();
        throw new UsernameNotFoundException("User " + username + " was not found in the database");
      }
  }
}
```

SecurityConfiguration.java

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
  @Autowired
  private CustomDetailsService customDetailsService;
  @Bean
  public PasswordEncoder encoder() {
    return new BCryptPasswordEncoder();
  }
  @Override
  @Autowired
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(customDetailsService).passwordEncoder(encoder());
  }
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated().and().sessionManagement()
      .sessionCreationPolicy(SessionCreationPolicy.NEVER);
  }
  @Override
  public void configure(WebSecurity web) throws Exception {
    web.ignoring();
  }
  @Override
  @Bean
  public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
  }
}
```

OAuth2Config.java

```
@Configuration
public class OAuth2Config extends AuthorizationServerConfigurerAdapter {
  private String clientid = "tarkeshwar";
  private String clientSecret = "my-secret-key";
  private String privateKey = "private key";
  private String publicKey = "public key";
  @Autowired
  @Qualifier("authenticationManagerBean")
  private AuthenticationManager authenticationManager;
  @Bean
  public JwtAccessTokenConverter tokenEnhancer() {
    JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
    converter.setSigningKey(privateKey);
    converter.setVerifierKey(publicKey);
    return converter;
  }
  @Bean
  public JwtTokenStore tokenStore() {
    return new JwtTokenStore(tokenEnhancer());
  }
  @Override
  public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
    endpoints.authenticationManager(authenticationManager).tokenStore(tokenStore())
    .accessTokenConverter(tokenEnhancer());
  }
  @Override
  public void configure(AuthorizationServerSecurityConfigurer security) throws Exception {
    security.tokenKeyAccess("permitAll()").checkTokenAccess("isAuthenticated()");
  }
  @Override
  public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
    clients.inMemory().withClient(clientid).secret(clientSecret).scopes("read", "write")
      .authorizedGrantTypes("password", "refresh_token").accessTokenValiditySeconds(20000)
      .refreshTokenValiditySeconds(20000);
  }
}
```

create a Private key and public key by using openssl. You can use the following commands for generating private key.

openssl genrsa -out jwt.pem 2048

openssl rsa -in jwt.pem

OAuth2Config.java

You can use For public key generation use the below commands.

openssl rsa -in jwt.pem -pubout

For the version of Spring Boot latter than 1.5 release, add the below property in your **application.properties** file to define OAuth2 Resource filter order.

security.oauth2.resource.filter-order=3

YAML file users can add the below property in YAML file.

```
security:
  oauth2:
    resource:
      filter-order: 3
```

Now, create schema.sql and data.sql file under the classpath resources src/main/resources/directory to connect the application to H2 database.

CREATE TABLE USERS (ID INT PRIMARY KEY, USERNAME VARCHAR(45), PASSWORD VARCHAR(60));

```
INSERT INTO USERS (ID, USERNAME,PASSWORD) VALUES (
   1,
'tbarua1@gmail.com','$2a$08$fL7u5xcvsZl78su29x1ti.dxI.9rYO8t0q5wk2ROJ.1cdR53
bmaVG');
INSERT INTO USERS (ID, USERNAME,PASSWORD) VALUES (
   2, 'myemail@gmail.com
','$2a$08$fL7u5xcvsZl78su29x1ti.dxI.9rYO8t0q5wk2ROJ.1cdR53bmaVG');
```

OAuth2Config.java

Password should be stored in the format of Bcrypt Encoder in the database table.

- **Authorization** − Basic Auth with your Client Id and Client secret.
- **Content Type** − application/x-www-form-urlencoded

add the Request Parameters as follows −

- **grant_type** = password
- **username** = your username
- **password** = your password

Now, hit the API and get the access_token