### **5.JWT-HANDSON**

### **Securing RESTful Web Services with Spring Security Follow steps below to secure all web services using Spring Security:**

### **Open spring-learn project in Eclipse**

### **Include spring security related libraries by adding the below dependency in pom.xml**

### **<dependency>**

### **<groupId>org.springframework.boot</groupId>**

### **<artifactId>spring-boot-starter-security</artifactId>**

### **</dependency>**

### **Rebuild the project in command line using mvn clean package command (ensure to include proxy details in mvn command).**

### **To ensure the new libraries are enabled in Eclipse, right click the project and select Maven > Update Project**

### **Create a new package 'com.cognizant.spring-learn.security'**

### **Create a new class SecurityConfig in the new package created above which extends from WebSecurityConfigurerAdapter**

### **Include annotations @Configuration and @EnableWebSecurity at class level**

### **Import appropriate classes using Ctrl + Shift + O**

### **Start the application and check the logs and test the REST service. Refer command below:**

### **curl -s http://localhost:8090/countries**

### **The following error message is the expected response:**

### **{"timestamp":"2019-10-05T09:24:33.794+0000","status":401,"error":"Unauthorized","message":"Unauthorized","path":"/countries"}**

### **The inclusion of @EnableWebSecurity has restricted access to all the web services with a common password.**

### **Refer the logs to find out the password generated. Now execute the invocation of the service with password as specified below, which should get the list of countries. include the password from the log file after user:.**

### **curl -s -v -u user:d27321a9-0751-4f59-8fc6-f8633847a9b8 http://localhost:8090/countries**

### **Find below a sample response for the above command:**

### **[{"code":"US","name":"United States"},{"code":"DE","name":"Germany"},{"code":"IN","name":"India"},{"code":"JP","name":"Japan"}]\* timeout on name lookup is not supported**

### **\*   Trying ::1...**

### **\* TCP\_NODELAY set**

### **\* Connected to localhost (::1) port 8090 (#0)**

### **\* Server auth using Basic with user 'user'**

### **> GET /countries HTTP/1.1**

### **> Host: localhost:8090**

### **> Authorization: Basic dXNlcjpkMjczMjFhOS0wNzUxLTRmNTktOGZjNi1mODYzMzg0N2E5Yjg=**

### **> User-Agent: curl/7.55.0**

### **> Accept: \*/\***

### **>**

### **< HTTP/1.1 200**

### **< Set-Cookie: JSESSIONID=C0C907417A21BBCA9F30BEEA4B512AEE; Path=/; HttpOnly**

### **< X-Content-Type-Options: nosniff**

### **< X-XSS-Protection: 1; mode=block**

### **< Cache-Control: no-cache, no-store, max-age=0, must-revalidate**

### **< Pragma: no-cache**

### **< Expires: 0**

### **< X-Frame-Options: DENY**

### **< Content-Type: application/json;charset=UTF-8**

### **< Transfer-Encoding: chunked**

### **< Date: Sat, 05 Oct 2019 09:36:34 GMT**

### **<**

### **{ [133 bytes data]**

### **\* Connection #0 to host localhost left intact**

### **First line contains the country list responded successfully.**

### **The above result contains the request header and response header.**

### **The request lines starts with > and reponse lines starts with <**

### **Notice the Authorization header in the HTTP Request**

### **This denotes that it uses basic HTTP Authorisation. Whatever following Basic is Base64 encoding of the password that was supplied in the command line.**

### **Detailed Explanation of Securing RESTful Web Services with Spring Security**

Let's go through each step outlined in your question and explain them in detail to secure the RESTful web services using **Spring Security**.

#### **1. Add Spring Security Dependency in pom.xml**

The first step in securing the web services is to add Spring Security-related libraries to the project.

**In your pom.xml**, add the following dependency:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

This will pull in the necessary libraries for Spring Security. The spring-boot-starter-security starter includes:

* **Spring Security** itself.
* Default authentication mechanisms like HTTP Basic Authentication.
* Configuration of secure communication and access controls.

After adding this dependency, run:

mvn clean package

This will rebuild the project and ensure that all dependencies, including Spring Security, are downloaded and properly set up.

#### **2. Rebuild the Project**

After adding the dependency, you should rebuild the project using Maven's **clean** and **package** commands. This ensures that Maven downloads all the necessary dependencies and packages your application. Make sure to include proxy settings in the command if you are behind a proxy:

mvn clean package

Then, go back to **Eclipse**:

* Right-click on the project.
* Select **Maven** > **Update Project** to ensure Eclipse is aware of the new libraries.

#### **3. Create a New Package com.cognizant.spring-learn.security**

Next, create a package to organize your security configurations. You will be placing all security-related classes here.

* In **Eclipse**, right-click on src/main/java.
* Select **New** > **Package**.
* Name the package: com.cognizant.spring-learn.security.

#### **4. Create the SecurityConfig Class**

Now, create the configuration class that will define your security settings. This class will extend WebSecurityConfigurerAdapter and will contain the rules for securing your RESTful services.

**Steps to create the SecurityConfig class**:

* Right-click on the security package.
* Select **New** > **Class**.
* Name it SecurityConfig.

Inside SecurityConfig.java, write the following code:

package com.cognizant.spring-learn.security;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.anyRequest().authenticated() // All requests require authentication

.and()

.httpBasic(); // Enable HTTP Basic Authentication

}

}

**Explanation**:

* @Configuration: Marks this class as a Spring configuration class.
* @EnableWebSecurity: Enables Spring Security in the application.
* WebSecurityConfigurerAdapter: This class is extended to override the configure(HttpSecurity http) method, which configures HTTP security.

In the configure(HttpSecurity http) method:

* .authorizeRequests() specifies the authorization rules for requests.
* .anyRequest().authenticated() ensures that all requests to the application require authentication.
* .httpBasic() enables basic HTTP authentication, which will prompt for a username and password.

#### **5. Run the Application**

After completing the above configurations, start the Spring Boot application. You can do this using **Eclipse** or the command line:

mvn spring-boot:run

#### **6. Test the RESTful Service**

Now that your application is running, open a terminal and test the REST service. You can use the curl command as follows:

curl -s http://localhost:8090/countries

**Expected Response**:

{

"timestamp": "2019-10-05T09:24:33.794+0000",

"status": 401,

"error": "Unauthorized",

"message": "Unauthorized",

"path": "/countries"

}

**Explanation**:

* The response status 401 Unauthorized is the expected outcome because the service is secured by Spring Security and requires authentication. The error message indicates that the service could not be accessed without providing the necessary credentials.

#### **7. Retrieve the Password from Logs**

After the application starts, check the logs for the password generated by Spring Security. In the application logs, you will find a log message that looks like this:

pgsql

Using generated security password: d27321a9-0751-4f59-8fc6-f8633847a9b8

This password is generated automatically by Spring Security when you don’t configure a specific password. The user that is created is user, and the generated password will be logged.

#### **8. Invoke the Service with Authentication**

Once you have the password, use it to authenticate and access the RESTful service. Use the following curl command, replacing the generated password from the logs:

curl -s -v -u user:d27321a9-0751-4f59-8fc6-f8633847a9b8 http://localhost:8090/countries

**Expected Response** (Sample):

[

{"code":"US","name":"United States"},

{"code":"DE","name":"Germany"},

{"code":"IN","name":"India"},

{"code":"JP","name":"Japan"}

]

**Explanation**:

* The -u flag is used to pass the username and password in the format username:password.
* The Authorization header in the HTTP request will contain the **Basic Authentication** information, where the password is Base64 encoded.
* The response now contains the list of countries, confirming that the service is successfully secured and accessible after authentication.

#### **9. Authorization Header and Basic Authentication**

In the request, the **Authorization** header is crucial. It will look like:

makefile

Authorization: Basic dXNlcjpkMjczMjFhOS0wNzUxLTRmNTktOGZjNi1mODYzMzg0N2E5Yjg=

This is the Base64-encoded string of the user:password pair, where:

* user is the username.
* The password is the one generated by Spring Security and Base64 encoded.

Spring Security uses **Basic Authentication**, where the server checks the encoded credentials sent in the request to authenticate the user.

**Creating users and roles in Spring Security**   
  
The earlier hands on demonstrated securing all URLs of the application with a common password. But it is not user and role specific.  
  
Let us create two new in memory users with names 'admin' and 'user'. The password for both the users will be 'pwd'.  
  
Let us define the rule that getting all countries can be accessed only 'user'.  
  
Refer steps below to incorporate the above aspects:

* Include the below methods in the SecurityConfig class

    @Override

    protected void configure(AuthenticationManagerBuilderauth) throws Exception {

        auth.inMemoryAuthentication()

            .withUser("admin").password(passwordEncoder().encode("pwd")).roles("ADMIN")

            .and()

            .withUser("user").password(passwordEncoder().encode("pwd")).roles("USER");

    }

    @Bean

    publicPasswordEncoderpasswordEncoder() {

        LOGGER.info("Start");

        return new BCryptPasswordEncoder();

    }

    @Override

    protected void configure(HttpSecurityhttpSecurity) throws Exception {

        httpSecurity.csrf().disable().httpBasic().and()

            .authorizeRequests().antMatchers("/countries").hasRole("USER");

    }

* The first configure() method defines two users admin and user with password as pwd. It also includes the specification of respective roles.

**IMPORTANT NOTE:** For learning purpose we are hard coding user details. When working on Spring Data JPA module, the credentials will be validate from the database. 

* The password encoder is required to encrypt the password.
* The second configure() method defines that /countries services is accessible only to users of role "USER"
* For testing the service with right credentials:

curl -s -u user:pwd http://localhost:8090/countries

* For testing the service with incorrect credentials and response:

curl -s -u user:pwd1 http://localhost:8090/countries

{"timestamp":"2019-10-05T10:19:08.237+0000","status":401,"error":"Unauthorized","message":"Unauthorized","path":"/countries"}

* For testing the service with correct credentials but a different role

curl -s -u admin:pwd http://localhost:8090/countries

{"timestamp":"2019-10-05T10:22:38.015+0000","status":403,"error":"Forbidden","message":"Forbidden","path":"/countries"}

**Limitations of this security approach**

* RESTful Web Service is a stateless protocol, hence each request needs to be attached the with user id and password credentials.
* The credentials passed on the HTTP request is not secure. Refer steps below to understand this better:
* Execute the below command to display the request and response headers:

curl -s -v -u admin:pwd http://localhost:8090/countries

* In the result display, in the request section, refer the Authorization

> Authorization: Basic YWRtaW46cHdk

* If "admin:pwd" is encoded with Base64 it results in  "YWRtaW46cHdk"
* Search using google and find out a online website that can decode Base64. (Example website, https://www.base64decode.net/)
* Try decoding YWRtaW46cHdk using the web site and one can obtain "admin:pwd"

These liminations can be overcome by incorporating security using JWT. Subsequent hands on will address this issue.

### **Detailed Explanation of Creating Users and Roles in Spring Security**

Let's break down the steps required to create **in-memory users** and **roles** in Spring Security and define access rules for specific roles. Additionally, we'll also explore the limitations of this approach and how it can be improved with **JWT**.

#### **1. In-memory Authentication with Users and Roles**

In this step, we’ll define two in-memory users: admin and user, both with the same password pwd. Each user will have a different role: ADMIN for admin and USER for user. We also need to configure a **password encoder** to encode passwords securely.

##### **Code for SecurityConfig.java:**

package com.cognizant.spring-learn.security;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

// In-memory authentication with two users and roles

auth.inMemoryAuthentication()

.withUser("admin").password(passwordEncoder().encode("pwd")).roles("ADMIN")

.and()

.withUser("user").password(passwordEncoder().encode("pwd")).roles("USER");

}

// Password Encoder Bean (BCryptPasswordEncoder)

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder(); // BCrypt is a strong password hashing algorithm

}

@Override

protected void configure(HttpSecurity httpSecurity) throws Exception {

httpSecurity.csrf().disable()

.httpBasic().and() // Enable HTTP Basic Authentication

.authorizeRequests()

.antMatchers("/countries").hasRole("USER"); // Only allow users with the "USER" role to access "/countries"

}

}

#### **Explanation of Key Sections:**

* **AuthenticationManagerBuilder**: In this method, we define two in-memory users: admin and user. Both users have the same password (pwd), but different roles:  
  + admin has the role ADMIN.
  + user has the role USER.
* **passwordEncoder()**: We use the BCryptPasswordEncoder to encode the passwords. BCrypt is a secure password hashing algorithm that ensures passwords are stored in a hashed format, making them difficult to reverse.
* **configure(HttpSecurity httpSecurity)**: Here, we configure access rules. Specifically, we restrict access to the /countries endpoint to users with the USER role using the .hasRole("USER") rule.

#### **2. Test the Service with Correct Credentials**

Now that the authentication and role-based access control is set up, let's test the REST service.

* **Test with correct credentials** (for user):

curl -s -u user:pwd http://localhost:8090/countries

**Expected Response**:

[{"code":"US","name":"United States"},{"code":"DE","name":"Germany"},{"code":"IN","name":"India"},{"code":"JP","name":"Japan"}]

* **Test with incorrect credentials** (wrong password for user):

curl -s -u user:pwd1 http://localhost:8090/countries

**Expected Response** (401 Unauthorized):

{

"timestamp": "2019-10-05T10:19:08.237+0000",

"status": 401,

"error": "Unauthorized",

"message": "Unauthorized",

"path": "/countries"

}

* **Test with correct credentials but different role (admin instead of user)**:

curl -s -u admin:pwd http://localhost:8090/countries

**Expected Response** (403 Forbidden):

{

"timestamp": "2019-10-05T10:22:38.015+0000",

"status": 403,

"error": "Forbidden",

"message": "Forbidden",

"path": "/countries"

}

**Explanation**:

* The first test (user:pwd) works because the user has the USER role, which grants access to /countries.
* The second test fails because the password is incorrect.
* The third test fails because even though the credentials are correct (admin:pwd), the admin role does not have access to /countries as per the configured role-based access.

#### **3. Limitations of This Security Approach**

While the approach of securing RESTful web services with **Basic Authentication** works, there are some significant limitations:

1. **Stateless Protocol**:  
   * REST is a stateless protocol, meaning that every HTTP request is independent and must carry all necessary credentials for the server to authenticate the user.
   * Each request requires sending the user credentials (username and password), which can be insecure.
2. **Insecure Credentials Transmission**:  
   * The HTTP Basic Authentication sends the username and password in each request. These credentials are encoded using **Base64**, but Base64 is **not encryption**. It is easily decodable.
   * For example, if you encode admin:pwd in Base64, you get YWRtaW46cHdk. This can be decoded back into the plain text admin:pwd using any Base64 decoding tool, making it insecure.

##### **Example of Base64 Decoding:**

To demonstrate the insecurity, here’s how you can decode the Base64-encoded string:

1. Take the Base64-encoded string from the Authorization header:

makefile

Authorization: Basic YWRtaW46cHdk

1. Use an online tool to decode the Base64 string, such as Base64 Decode.  
     
    When decoded, you will get:

bash

admin:pwd

This clearly shows that Base64 encoding is not a secure way to protect passwords.

#### **4. Security Improvements Using JWT (JSON Web Tokens)**

* The **Basic Authentication** mechanism, although simple, is not secure enough for production applications. It exposes the credentials in each request and relies on Base64 encoding, which can be easily decoded.
* A better approach is to use **JWT** (JSON Web Tokens) for stateless authentication.

With JWT:

* Users authenticate once, and a token is issued.
* The token is sent in the request header for subsequent requests.
* The server can validate the token without needing to authenticate the user each time.

In future hands-on sessions, JWT will be implemented to overcome these limitations, ensuring more secure and scalable authentication.

**Understanding JWT**   
  
**What is JWT?**

* JWT stands for JSON Web Token
* Internet standard ([**IETF 7519 Link**](https://tools.ietf.org/html/rfc7519)) for creating [JSON](https://en.wikipedia.org/wiki/JSON)-based [access tokens](https://en.wikipedia.org/wiki/Access_token)
* JWT can be typically used to pass identity of authenticated users and [service provider](https://en.wikipedia.org/wiki/Service_provider),

**JWT Process Flow (**[**diagram**](https://miro.medium.com/max/1600/0*13yKRyewaI1sLFSz.)**link)**

* Client sends username and password to server
* Servers validates credentials, creates token (JWT) and reponds it back
* Client attaches the token in the subsequent requests to server
* Server validates the token (JWT) on each client request

**Structure of JSON Web Token**

* Reference: https://en.wikipedia.org/wiki/JSON\_Web\_Token#Structure
* Header: Contains the encryption algorithm
* Payload: Contains application specific data. Usually this contains the user id and role.
* Signature: Computed based on the formula defined using header and payload

**Exercise to check how JWT token is created**

* Open link https://en.wikipedia.org/wiki/JSON\_Web\_Token#Structure in browser
* Open link https://jwt.io/ in another browser tab and scroll down to the Encoded, Decoded section
* Copy and paste the header content from wikipedia article and paste it in header section of https://jwt.io
* Copy and paste the payload content from wikipedia article and paste it in payload section of https://jwt.io
* Type "secretkey" in the textbox within Verify Signature section
* Check if the token generated in the Encoded section of https://jwt.io matches with the generated token displayed in the Structure section of wikipedia article

### **Detailed Explanation of JWT (JSON Web Token) and Its Creation**

#### **1. What is JWT (JSON Web Token)?**

**JWT** is a compact, URL-safe means of representing claims to be transferred between two parties. It is often used for securely transmitting information about the user between client and server. JWTs are most commonly used for **authentication** and **authorization** in modern web applications.

##### **Key Features of JWT:**

* **Compact**: The token is small in size and can be easily sent via URL or HTTP headers.
* **Self-contained**: Contains all the necessary information about the user and can be verified and trusted by the server.
* **Stateless Authentication**: The server does not need to maintain any session data for the user as all information is encoded within the token itself.
* **Secure**: JWTs can be signed and optionally encrypted to ensure confidentiality and integrity.

##### **Typical Use Case:**

JWT is often used to pass the **identity** of an authenticated user along with their **roles** and **permissions** to a service provider. For example, once the user logs in, a JWT can be issued to them, which is sent with every request thereafter to access protected resources.

#### **2. JWT Process Flow**

Here's how the JWT authentication process typically works:

1. **Client sends username and password to the server**: The client authenticates by sending the username and password (usually over HTTPS) to the server.
2. **Server validates credentials and creates the JWT**: Upon successful authentication, the server creates a JWT. This JWT includes claims such as user information and roles, and it is signed with a secret key to prevent tampering.
3. **Client stores and sends the token**: The client stores the JWT (typically in local storage or a cookie) and sends it as part of the authorization header in subsequent requests to the server.
4. **Server validates the token**: Every time the client sends a request with the JWT, the server validates the token using the secret key. If the token is valid, the request is processed; otherwise, the client is unauthorized.

#### **3. Structure of a JSON Web Token**

A JWT consists of three parts:

* **Header**
* **Payload**
* **Signature**

##### **Header:**

* The header typically consists of two parts:  
  + The **type** of the token, which is JWT.
  + The **signing algorithm**, such as HMAC SHA256 or RSA.

Example of a JWT header:

{

"alg": "HS256",

"typ": "JWT"

}

##### **Payload:**

* The payload contains the claims, which are statements about an entity (usually the user) and additional metadata.  
  + **Registered claims** (e.g., sub for subject, iat for issued at, exp for expiration time).
  + **Public claims** (custom claims, like user roles, user ID, etc.).
  + **Private claims** (custom claims shared between parties).

Example of a JWT payload:

{

"sub": "user",

"iat": 1570379474,

"exp": 1570383074

}

##### **Signature:**

* The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't tampered with.
* It is computed using the header, payload, and a secret key.

The signature is created with this formula:

scss

HMACSHA256(

base64UrlEncode(header) + "." + base64UrlEncode(payload),

secret)

#### **4. Exercise to Check How JWT Token is Created**

To understand how JWT tokens are created, follow these steps:

1. Open the link: JSON Web Token Structure in one browser tab.
2. Open the link: JWT.io in another tab.
3. Copy and paste the header content from the Wikipedia article into the header section of JWT.io.
4. Copy and paste the payload content from the Wikipedia article into the payload section of JWT.io.
5. Type secretkey in the **Verify Signature** textbox.
6. Check if the **Encoded** token generated in JWT.io matches the token displayed in the Wikipedia article.

This exercise will help you visualize how the different parts of a JWT (header, payload, and signature) work together.

#### **5. Create Authentication Service that Returns JWT**

Once you have a clear understanding of JWT, let's implement a simple authentication service in a Spring Boot application that generates and returns a JWT. The process includes the following major steps:

1. **Create an Authentication Controller**: This controller will handle the /authenticate endpoint where users send their credentials.
2. **Read Authorization Header**: The server will decode the username and password from the Authorization header.
3. **Generate JWT**: If the credentials are valid, the server will generate a JWT token and return it.

##### **Step 1: Create Authentication Controller and Configure in SecurityConfig**

**AuthenticationController.java**:

package com.cognizant.springlearn.controller;

import com.cognizant.springlearn.security.JwtTokenUtil;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.Authentication;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.web.bind.annotation.GetMapping;

@RestController

public class AuthenticationController {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private JwtTokenUtil jwtTokenUtil;

@RequestMapping("/authenticate")

public ResponseEntity<?> createAuthenticationToken(@RequestParam String username, @RequestParam String password) throws Exception {

Authentication authentication = authenticationManager.authenticate(

new UsernamePasswordAuthenticationToken(username, password)

);

SecurityContextHolder.getContext().setAuthentication(authentication);

// Generate the JWT token

String token = jwtTokenUtil.generateToken(authentication);

return ResponseEntity.ok(new JwtResponse(token));

}

}

In this controller:

* **/authenticate** endpoint is used to authenticate users.
* The credentials (username and password) are passed via request parameters, authenticated by AuthenticationManager.
* A JWT token is generated if authentication is successful and returned to the client.

##### **Step 2: Decode the Username and Password**

The username and password are automatically decoded from the Authorization header as part of the authentication process in Spring Security. We use **Basic Authentication** for this purpose, and Spring Security provides built-in support for handling it.

##### **Step 3: Generate JWT**

We use a utility class **JwtTokenUtil** to generate the token. It signs the token with a secret key, as shown below:

**JwtTokenUtil.java**:

package com.cognizant.springlearn.security;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import org.springframework.security.core.Authentication;

import org.springframework.stereotype.Component;

import java.util.Date;

@Component

public class JwtTokenUtil {

private String secretKey = "secretkey";

public String generateToken(Authentication authentication) {

// Get the username from the Authentication object

String username = authentication.getName();

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + 86400000)) // Token expiration in 1 day

.signWith(SignatureAlgorithm.HS256, secretKey)

.compact();

}

}

In this utility:

* **generateToken()**: This method generates a JWT based on the provided authentication object. It includes the username, issue date, and expiration date.

##### **Step 4: Testing the Service**

Run the Spring Boot application and test the authentication service using **cURL**:

**Request**:

curl -s -u user:pwd http://localhost:8090/authenticate

**Expected Response**:

{

"token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIiwiaWF0IjoxNTcwMzc5NDc0LCJleHAiOjE1NzAzODA2NzR9.t3LRvlCV-hwKfoqZYlaVQqEUiBloWcWn0ft3tgv0dL0"

}

The response contains the JWT token, which the client can use for subsequent requests by including it in the Authorization header.

#### **Conclusion**

This exercise walks you through the process of creating an authentication service that generates and returns a JWT. You learned how to:

* Use **Spring Security** to authenticate users.
* Generate a JWT using the **JWT.io** library.
* Return the generated JWT to the client for further use in API requests.

In a real-world application, this JWT can then be used for **stateless authentication**, allowing clients to interact with the backend without needing to store session data.

### **Create authentication service that returns JWT  As part of first step of JWT process, the user credentials needs to be sent to authentication service request that generates and returns the JWT. Ideally when the below curl command is executed that calls the new authentication service, the token should be responded. Kindly note that the credentials are passed using -u option. Request**

### **curl -s -u user:pwd http://localhost:8090/authenticate**

### **Response**

### **{"token":"eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIiwiaWF0IjoxNTcwMzc5NDc0LCJleHAiOjE1NzAzODA2NzR9.t3LRvlCV-hwKfoqZYlaVQqEUiBloWcWn0ft3tgv0dL0"}**

### **This can be incorporated as three major steps:**

### **Create authentication controller and configure it in SecurityConfig**

### **Read Authorization header and decode the username and password**

### **Generate token based on the user retrieved in the previous step**

### **Let incorporate the above as separate hands on exercises.**

### **Create authentication controller and configure it in SecurityConfig  AuthenticationController.java**

### **Create new rest controller named AuthenticationController in controller package**

### **Include method authenticate with "/authenticate" as the URL with @GetMapping.**

### **To read the Authorization value from HTTP Header, include a parameter for authenticate method as specified below. Spring takes care of reading the Authorization value from HTTP Header and pass it as parameter.**

### **@RequestHeader("Authorization") String authHeader**

### **The return type of this method should be Map<String, String>**

### **Include start and end logger in this method**

### **Include a debug log for displaying the authHeader parameter**

### **Create a new HashMap<String, String> and assign it to a map.**

### **Put a new item into the map with key as "token" and value as empty string.**

### **SecurityConfig.java**

### **In the second configure method, include authenticate URL just after the countries URL defined earlier. Refer code below:**

### **.antMatchers("/countries").hasRole("USER")**

### **.antMatchers("/authenticate").hasAnyRole("USER", "ADMIN")**

### **The above configuration sets that users of both roles can access /authenticate URL.**

### **Testing *curl command:***

### **curl -s -u user:pwd http://localhost:8090/authenticate**

### ***Expected Response:***

### **{"token":""}**

### ***Log verification:* Check if Authorization header value is displayed with "Basic" prefix and Base64 encoding of "user:pwd"**

### **Detailed Explanation of Creating Authentication Controller and Configuring Security**

Let's go step by step to implement the **AuthenticationController** and update the **SecurityConfig** to allow users with the roles USER and ADMIN to access the /authenticate URL.

#### **1. Create AuthenticationController**

We will create a new REST controller called AuthenticationController to handle the /authenticate endpoint. The purpose of this controller is to read the **Authorization** header from the request, authenticate the user, and return a JWT (or an empty string, for now).

##### **AuthenticationController.java**

package com.cognizant.springlearn.controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RestController;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.util.HashMap;

import java.util.Map;

@RestController

public class AuthenticationController {

private static final Logger logger = LoggerFactory.getLogger(AuthenticationController.class);

@GetMapping("/authenticate")

public Map<String, String> authenticate(@RequestHeader("Authorization") String authHeader) {

// Start logger

logger.info("Start Authentication");

// Log the Authorization header value at the debug level

logger.debug("Authorization Header: {}", authHeader);

// Create a map to return the response

Map<String, String> response = new HashMap<>();

// Put an empty token into the map

response.put("token", "");

// End logger

logger.info("End Authentication");

return response;

}

}

#### **Explanation of the Code:**

* **@RestController**: Marks this class as a Spring MVC controller where each method will return a response body.
* **@GetMapping("/authenticate")**: This method is mapped to the /authenticate URL, which handles GET requests.
* **@RequestHeader("Authorization")**: This annotation allows Spring to read the **Authorization** header from the HTTP request and pass it to the authHeader parameter.
* **Logger**:  
  + We use **SLF4J** with **Logback** to log information.
  + logger.info("Start Authentication") logs the start of the authentication process.
  + logger.debug("Authorization Header: {}", authHeader) logs the value of the **Authorization** header at the debug level.
  + logger.info("End Authentication") logs the end of the authentication process.
* **Returning a Map**:  
  + A HashMap<String, String> is used to create the response.
  + The token value is set to an empty string for now.

#### **2. Update SecurityConfig**

In **SecurityConfig.java**, we need to update the security configuration to allow users with both USER and ADMIN roles to access the /authenticate URL. This can be done by modifying the **HttpSecurity** configuration.

##### **SecurityConfig.java**

package com.cognizant.springlearn.security;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity httpSecurity) throws Exception {

httpSecurity.csrf().disable()

.httpBasic().and() // Enable HTTP Basic Authentication

.authorizeRequests()

.antMatchers("/countries").hasRole("USER") // Only users with "USER" role can access "/countries"

.antMatchers("/authenticate").hasAnyRole("USER", "ADMIN") // Users with either "USER" or "ADMIN" can access "/authenticate"

.anyRequest().authenticated(); // Other requests must be authenticated

}

}

#### **Explanation of Changes:**

* **antMatchers("/authenticate").hasAnyRole("USER", "ADMIN")**: This line configures the /authenticate URL to be accessible by both **users with the "USER"** role and **users with the "ADMIN"** role.

#### **3. Testing the Implementation**

We can test the /authenticate endpoint using **cURL**.

* Use the following command to call the /authenticate service with correct credentials:

curl -s -u user:pwd http://localhost:8090/authenticate

**Expected Response**:

{

"token": ""

}

#### **Log Verification:**

1. **Check Logs**: When the request is made, the following should appear in the logs:  
   * Start of the authentication process: Start Authentication.

The **Authorization** header should be logged with the **Base64-encoded credentials**:  
  
 Authorization Header: Basic dXNlcjpwd2Q=

* + This is the Base64 encoding of user:pwd.
  + End of the authentication process: End Authentication.

1. **Decoding Authorization Header**:  
   * If you take the value dXNlcjpwd2Q=, you can decode it to get the **username and password** (i.e., user:pwd).
   * Use an online tool like Base64 Decode to decode the value.

#### **4. Additional Notes**

* The current implementation returns an empty string for the **token**. This is just for testing purposes.
* In a real-world application, the **JWT token** would be generated here and sent back to the client.

#### **Conclusion**

You’ve successfully created an **AuthenticationController** that reads the **Authorization header** and logs the credentials. You also updated the **SecurityConfig** to restrict access to /authenticate for users with the roles USER and ADMIN. The next step would be to actually generate and return a **JWT** token upon successful authentication.

**Read Authorization header and decode the username and password**   
  
Steps to read and decode header:

* Create a new private method in AuthenticationController with below method signature

private String getUser(String authHeader)

* Get the Base64 encoded text after "Basic "
* Decode it using the library available in Java 8 API. Refer code below.

Base64.getDecoder().decode(encodedCredentials)

* The above call returns a byte array, which can be passed as parameter to string constructor to convert to string.
* Get the text until colon on the string created in previous step to get the user
* Return the user obtained in previous step
* Include appropriate debug logs within this method
* Invoke the getUser() method from authenticate method
* Execute the curl command used in the previous step and check the logs if the user information is obtained successfully.

### **Detailed Explanation of Reading Authorization Header and Generating JWT**

To complete the task of reading the **Authorization** header and generating a **JWT** token, we will break it down into two parts:

1. **Reading and decoding the Authorization header** to extract the username and password.
2. **Generating a JWT token** based on the username.

Let's go through the steps in detail:

### **Step 1: Read and Decode the Authorization Header**

We will create a private method in AuthenticationController to read and decode the **Authorization** header to extract the username.

#### **Code for getUser Method in AuthenticationController.java**

package com.cognizant.springlearn.controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RestController;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.util.Base64;

import java.util.HashMap;

import java.util.Map;

@RestController

public class AuthenticationController {

private static final Logger logger = LoggerFactory.getLogger(AuthenticationController.class);

@GetMapping("/authenticate")

public Map<String, String> authenticate(@RequestHeader("Authorization") String authHeader) {

// Start logger

logger.info("Start Authentication");

// Log the Authorization header value at the debug level

logger.debug("Authorization Header: {}", authHeader);

// Get the username using the getUser method

String user = getUser(authHeader);

logger.debug("Extracted Username: {}", user);

// Generate the token for the user

String token = generateJwt(user);

// Create a map to return the response with the token

Map<String, String> response = new HashMap<>();

response.put("token", token);

// End logger

logger.info("End Authentication");

return response;

}

// Private method to decode the Authorization header and extract the username

private String getUser(String authHeader) {

// Log that we are starting the decoding process

logger.debug("Decoding Authorization header");

// Remove the "Basic " prefix and decode the remaining base64 encoded string

String encodedCredentials = authHeader.substring(6); // "Basic " is 6 characters

byte[] decodedBytes = Base64.getDecoder().decode(encodedCredentials);

// Convert the byte array to a string

String decodedString = new String(decodedBytes);

// Log the decoded string (username:password)

logger.debug("Decoded Credentials: {}", decodedString);

// Extract the username (before the colon)

String user = decodedString.split(":")[0];

return user;

}

}

#### **Explanation of getUser Method:**

* **authHeader.substring(6)**: We remove the first 6 characters, "Basic ", from the header value to get the Base64-encoded credentials.
* **Base64.getDecoder().decode(encodedCredentials)**: Decodes the Base64 string into a byte array.
* **new String(decodedBytes)**: Converts the byte array into a plain string (username:password).
* **split(":")[0]**: Splits the string at the colon (:) and returns the username part.

#### **Logging:**

* **logger.debug("Decoded Credentials: {}", decodedString)**: Logs the decoded string.
* **logger.debug("Extracted Username: {}", user)**: Logs the extracted username.

### **Step 2: Generate JWT Based on the User**

Now that we have the username, we need to generate a **JWT** token for the user.

#### **Maven Dependency for JWT Library**

Add the following Maven dependency for the **JWT** library to your pom.xml file:

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt</artifactId>

<version>0.9.0</version>

</dependency>

After adding the dependency, run the Maven build using:

mvn clean package

Then, update the project in **Eclipse** to ensure the library is added.

#### **Code for generateJwt Method in AuthenticationController.java**

Now, let's add the method to generate the JWT token:

package com.cognizant.springlearn.controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RestController;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import io.jsonwebtoken.JwtBuilder;

import java.util.Base64;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

@RestController

public class AuthenticationController {

private static final Logger logger = LoggerFactory.getLogger(AuthenticationController.class);

@GetMapping("/authenticate")

public Map<String, String> authenticate(@RequestHeader("Authorization") String authHeader) {

// Start logger

logger.info("Start Authentication");

// Log the Authorization header value at the debug level

logger.debug("Authorization Header: {}", authHeader);

// Get the username using the getUser method

String user = getUser(authHeader);

logger.debug("Extracted Username: {}", user);

// Generate the token for the user

String token = generateJwt(user);

// Create a map to return the response with the token

Map<String, String> response = new HashMap<>();

response.put("token", token);

// End logger

logger.info("End Authentication");

return response;

}

// Private method to decode the Authorization header and extract the username

private String getUser(String authHeader) {

logger.debug("Decoding Authorization header");

String encodedCredentials = authHeader.substring(6); // "Basic " is 6 characters

byte[] decodedBytes = Base64.getDecoder().decode(encodedCredentials);

String decodedString = new String(decodedBytes);

logger.debug("Decoded Credentials: {}", decodedString);

// Extract the username (before the colon)

return decodedString.split(":")[0];

}

// Private method to generate the JWT token for the user

private String generateJwt(String user) {

logger.debug("Generating JWT for user: {}", user);

// Build the JWT token using JJWT library

JwtBuilder builder = Jwts.builder();

builder.setSubject(user);

// Set the token issue time as current time

builder.setIssuedAt(new Date());

// Set the token expiry as 20 minutes from now

builder.setExpiration(new Date(System.currentTimeMillis() + 1200000));

// Sign the token with a secret key (HS256 algorithm)

builder.signWith(SignatureAlgorithm.HS256, "secretkey");

// Compact the JWT into a string

String token = builder.compact();

logger.debug("Generated Token: {}", token);

return token;

}

}

#### **Explanation of generateJwt Method:**

* **JwtBuilder**: The **JJWT** library's JwtBuilder is used to build the token.  
  + **setSubject(user)**: Sets the subject (username) of the JWT.
  + **setIssuedAt(new Date())**: Sets the issue time as the current date and time.
  + **setExpiration(new Date(System.currentTimeMillis() + 1200000))**: Sets the expiration time to 20 minutes from the current time (in milliseconds).
  + **signWith(SignatureAlgorithm.HS256, "secretkey")**: Signs the token using the **HS256** algorithm and a secret key.
* **builder.compact()**: Converts the JWT builder into a compact string format (the actual JWT token).

### **Step 3: Testing the Implementation**

Once you have added the methods for reading the Authorization header and generating the JWT token, you can test it by running the following **cURL** command:

curl -s -u user:pwd http://localhost:8090/authenticate

**Expected Response**:

{

"token": "your-generated-jwt-token-here"

}

#### **Log Output:**

The **Authorization** header should be logged at the **debug level** as:  
  
 pgsql  
Authorization Header: Basic dXNlcjpwd2Q=

1. The decoded credentials (user:pwd) should be logged.
2. The generated JWT token should be logged as well.

### **Detailed Explanation of JWT Authorization Filter Implementation**

To ensure all incoming requests to the application are validated with JWT, we will implement a **JWT Authorization Filter**. This filter will intercept requests and validate the token before they are processed further.

Let's go step by step through the process of setting up the JwtAuthorizationFilter and configuring it within SecurityConfig.

### **Step 1: Create JwtAuthorizationFilter Class**

We need to create a new class, JwtAuthorizationFilter, inside the com.cognizant.springlearn.security package. This class will extend **BasicAuthenticationFilter** provided by Spring Security.

#### **JwtAuthorizationFilter.java**

package com.cognizant.springlearn.security;

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.Jws;

import io.jsonwebtoken.JwtException;

import io.jsonwebtoken.Jwts;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.web.authentication.www.BasicAuthenticationFilter;

import javax.servlet.FilterChain;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

import java.util.ArrayList;

public class JwtAuthorizationFilter extends BasicAuthenticationFilter {

private static final Logger LOGGER = LoggerFactory.getLogger(JwtAuthorizationFilter.class);

// Constructor that sets the authentication manager

public JwtAuthorizationFilter(AuthenticationManager authenticationManager) {

super(authenticationManager);

LOGGER.info("Start");

LOGGER.debug("Authentication Manager: {}", authenticationManager);

}

// Overriding the doFilterInternal method to validate JWT token

@Override

protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain)

throws IOException, ServletException {

LOGGER.info("Start");

String header = req.getHeader("Authorization");

LOGGER.debug("Authorization Header: {}", header);

if (header == null || !header.startsWith("Bearer ")) {

chain.doFilter(req, res);

return;

}

// Validate the JWT token and set the authentication

UsernamePasswordAuthenticationToken authentication = getAuthentication(req);

if (authentication != null) {

SecurityContextHolder.getContext().setAuthentication(authentication);

}

chain.doFilter(req, res);

LOGGER.info("End");

}

// Method to parse the JWT token and extract the username

private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {

String token = request.getHeader("Authorization");

if (token != null) {

try {

// Parse the JWT token

Jws<Claims> jws = Jwts.parser()

.setSigningKey("secretkey")

.parseClaimsJws(token.replace("Bearer ", ""));

// Get the username from the token

String user = jws.getBody().getSubject();

LOGGER.debug("Extracted User: {}", user);

if (user != null) {

// Return an authenticated token with the username and roles

return new UsernamePasswordAuthenticationToken(user, null, new ArrayList<>());

}

} catch (JwtException ex) {

LOGGER.error("Invalid token: {}", ex.getMessage());

}

}

return null;

}

}

#### **Explanation of the Code:**

1. **JwtAuthorizationFilter Constructor**:  
   * This constructor calls the BasicAuthenticationFilter constructor and passes the AuthenticationManager to it.
   * We also log the initialization of the filter using **SLF4J** logging.
2. **doFilterInternal Method**:  
   * This method intercepts the HTTP request.
   * It checks if the **Authorization header** is present and starts with "Bearer ". If not, it passes the request to the next filter in the chain.
   * If the header is valid, the method calls getAuthentication(req) to validate the token.
3. **getAuthentication Method**:  
   * This method extracts the JWT token from the **Authorization header**.
   * It uses **JJWT** library to parse and validate the token.
   * If the token is valid, it extracts the username (subject) and returns a **UsernamePasswordAuthenticationToken**.
   * If the token is invalid or not present, it returns null.

### **Step 2: Modify SecurityConfig to Use the JwtAuthorizationFilter**

Now, we need to update **SecurityConfig** to register our JwtAuthorizationFilter and ensure it is applied to all incoming requests.

#### **Updated SecurityConfig.java**

package com.cognizant.springlearn.security;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.authentication.AuthenticationManager;

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity httpSecurity) throws Exception {

httpSecurity.csrf().disable()

.httpBasic().and() // Enable HTTP Basic Authentication

.authorizeRequests()

// Uncomment if you want to enforce role-based access to /countries

//.antMatchers("/countries").hasRole("USER")

.antMatchers("/authenticate").hasAnyRole("USER", "ADMIN")

.anyRequest().authenticated() // Other requests must be authenticated

.and()

// Add our JWT filter

.addFilter(new JwtAuthorizationFilter(authenticationManager()));

}

}

#### **Explanation of Changes:**

* **addFilter(new JwtAuthorizationFilter(authenticationManager()))**: This line adds the JwtAuthorizationFilter to the Spring Security filter chain. The filter will be applied to all incoming HTTP requests to validate the JWT.

### **Step 3: Test the Implementation**

After the filter has been implemented and configured, we can test the JWT authorization by following these steps:

#### **Test 1: Generate JWT Token**

First, generate a fresh JWT token using the /authenticate endpoint:

curl -s -u user:pwd http://localhost:8090/authenticate

**Expected Response**:

{

"token": "your-jwt-token-here"

}

Copy the generated token, as it will be used in the next step.

#### **Test 2: Access a Protected Service**

Use the **JWT token** to access any protected endpoint (e.g., /countries):

curl -s -H "Authorization: Bearer your-jwt-token-here" http://localhost:8090/countries

**Expected Response**: The response will depend on the authorization configured in Spring Security. If the user has the necessary roles, the request will succeed and return the requested data.

#### **Test 3: Access with an Invalid Token**

Now, modify the token slightly (e.g., change a character in the token) and try accessing the service again:

curl -s -H "Authorization: Bearer invalid-token" http://localhost:8090/countries

**Expected Response** (Unauthorized):

{

"timestamp": "2021-09-14T14:23:45.678+0000",

"status": 401,

"error": "Unauthorized",

"message": "Unauthorized",

"path": "/countries"

}

This confirms that the server correctly invalidates any request with an incorrect token.

### 