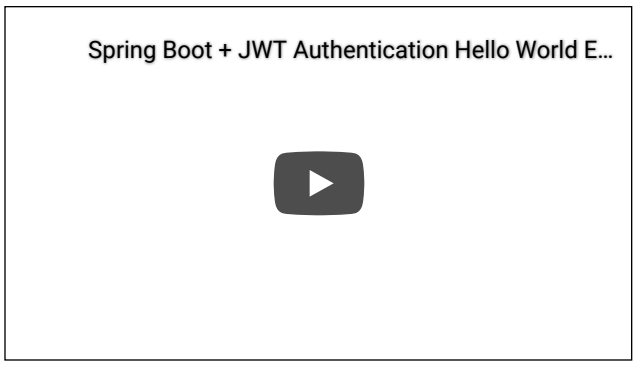
Spring Boot Security + JWT "Hello World" Example

(/users/3692891/ridashaikh.html) by Rida Shaikh (/users/3692891/ridashaikh.html) · May. 24, 19 · Security Zone (/application-web-network-security) · Tutorial

In this tutorial, we will be developing a Spring Boot application that makes use of JWT authentication for securing an exposed REST API. In this example, we will be making use of hard-coded user values for user authentication. In the next tutorial, we will be implementing Spring Boot + JWT + MYSQL JPA for storing and fetching user credentials. (https://www.javainuse.com/spring/boot-jwt-mysql) Any user will be able to consume this API only if it has a valid JSON Web Token (JWT). In a previous tutorial, we learned what is JWT and when and how to use it. (https://www.javainuse.com/spring/jwt)

This tutorial is explained in the following video:



For better understanding, we will be developing the project in stages:

Develop a Spring Boot application that exposes a simple REST GET API with mapping /hello.

Configure Spring Security for JWT. Expose REST POST API with mapping/authenticate using which User will get a valid JSON Web Token. And then, allow the user access to the API /hello only if it has a valid token

Spring Boot JWT Workflow

Develop a Spring Boot Application That Exposes a GET REST API

Develop a Spring Boot Application That Exposes a GET REST API

The Maven project will look as follows:


Spring Boot REST

The pom.xml is as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.javainuse</groupId>
<artifactId>spring-boot-jwt</artifactId>
<version>0.0.1-SNAPSHOT</version>

<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.1.1.RELEASE</version>
<relativePath /> <!-- lookup parent from repository -->
</parent>

<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
<java.version>1.8</java.version>
</properties>

<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
</dependencies>

</project>
```

Create a `Controller` class for exposing a GET REST API:

```
package com.javainuse.controller;

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

@RestController
public class HelloWorldController {

@RequestMapping({ "/hello" })
public String firstPage() {
return "Hello World";
}

}
```

Create the bootstrap class with the `SpringBoot` annotation:

```
package com.javainuse;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootHelloWorldApplication {

public static void main(String[] args) {
SpringApplication.run(SpringBootHelloWorldApplication.class, args);
}
}
```

Compile and then run the SpringBootHelloWorldApplication.java as a Java application.

Go to **localhost:8080/hello**

Spring Boot REST output

Spring Security and JWT Configuration

We will be configuring Spring Security and JWT to perform two operations:

Generating JWT: Expose a POST API with mapping **/authenticate**. On passing the correct username and password, it will generate a JSON Web Token (JWT).

Validating JWT: If a user tries to access the GET API with mapping **/hello**, it will allow access only if a request has a valid JSON Web Token (JWT).

The Maven project will look as follows:

Spring Boot JWT REST

The sequence flow for these operations will look as follows:

Generating JWT

Spring Boot JWT Generate Token

Spring Boot Security Authentication Manager

Validating JWT

Spring Boot JWT Validate Token

Add the Spring Security and JWT dependencies:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.javainuse</groupId>
<artifactId>spring-boot-jwt</artifactId>
<version>0.0.1-SNAPSHOT</version>

<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.1.1.RELEASE</version>
<relativePath /> <!-- lookup parent from repository -->
</parent>

<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
<java.version>1.8</java.version>
</properties>

<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
<groupId>io.jsonwebtoken</groupId>
<artifactId>jjwt</artifactId>
<version>0.9.1</version>
</dependency>
</dependencies>

</project>
```

Define the application.properties. As see in previous JWT tutorial, we specify the secret key, which we will be using for the hashing algorithm (https://www.javainuse.com/spring/jwt) The secret key is combined with the header and the payload to create a unique hash. We are only able to verify this hash if you have the secret key.

```
jwt.secret=javainuse
```

JwtTokenUtil

The `JwtTokenUtil` is responsible for performing JWT operations like creation and validation. It makes use of the io.jsonwebtoken.Jwts for achieving this.

```
package com.javainuse.config;

import java.io.Serializable;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

@Component
public class JwtTokenUtil implements Serializable {

private static final long serialVersionUID = -2550185165626007488L;

public static final long JWT_TOKEN_VALIDITY = 5 * 60 * 60;

@Value("${jwt.secret}")
private String secret;

//retrieve username from jwt token
public String getUsernameFromToken(String token) {
return getClaimFromToken(token, Claims::getSubject);
}

//retrieve expiration date from jwt token
public Date getExpirationDateFromToken(String token) {
return getClaimFromToken(token, Claims::getExpiration);
}

public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {
final Claims claims = getAllClaimsFromToken(token);
return claimsResolver.apply(claims);
}
    //for retrieveing any information from token we will need the secret key
private Claims getAllClaimsFromToken(String token) {
return Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();
}

//check if the token has expired
private Boolean isTokenExpired(String token) {
final Date expiration = getExpirationDateFromToken(token);
return expiration.before(new Date());
}

//generate token for user
public String generateToken(UserDetails userDetails) {
Map<String, Object> claims = new HashMap<>();
return doGenerateToken(claims, userDetails.getUsername());
}

//while creating the token -
//1. Define  claims of the token, like Issuer, Expiration, Subject, and the ID
//2. Sign the JWT using the HS512 algorithm and secret key.
//3. According to JWS Compact Serialization(https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-41#section-3.1)
//   compaction of the JWT to a URL-safe string
private String doGenerateToken(Map<String, Object> claims, String subject) {
```

```
return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis()))
.setExpiration(new Date(System.currentTimeMillis() + JWT_TOKEN_VALIDITY * 1000))
.signWith(SignatureAlgorithm.HS512, secret).compact();
}

//validate token
public Boolean validateToken(String token, UserDetails userDetails) {
final String username = getUsernameFromToken(token);
return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
}
}
```

JwtAuthenticationController

Expose a POST API /authenticate using the `JwtAuthenticationController`. The POST API gets the username and password in the body. Using the Spring Authentication Manager, we authenticate the username and password. If the credentials are valid, a JWT token is created using the `JWTTokenUtil` and is provided to the client.

```
package com.javainuse.controller;

import java.util.Objects;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.DisabledException;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import com.javainuse.service.JwtUserDetailsService;


import com.javainuse.config.JwtTokenUtil;
import com.javainuse.model.JwtRequest;
import com.javainuse.model.JwtResponse;

@RestController
@CrossOrigin
public class JwtAuthenticationController {

@Autowired
private AuthenticationManager authenticationManager;

@Autowired
private JwtTokenUtil jwtTokenUtil;

@Autowired
private JwtUserDetailsService userDetailsService;

@RequestMapping(value = "/authenticate", method = RequestMethod.POST)
public ResponseEntity<?> createAuthenticationToken(@RequestBody JwtRequest authenticationRequest) throws Exception {

authenticate(authenticationRequest.getUsername(), authenticationRequest.getPassword());

final UserDetails userDetails = userDetailsService
.loadUserByUsername(authenticationRequest.getUsername());

final String token = jwtTokenUtil.generateToken(userDetails);

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

private void authenticate(String username, String password) throws Exception {
try {
authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(username, password));
} catch (DisabledException e) {
throw new Exception("USER_DISABLED", e);
} catch (BadCredentialsException e) {
throw new Exception("INVALID_CREDENTIALS", e);
}
}
}
```

JwtRequest

This class is required for storing the username and password we received from the client.

```
package com.javainuse.model;

import java.io.Serializable;

public class JwtRequest implements Serializable {

private static final long serialVersionUID = 5926468583005150707L;

private String username;
private String password;

//need default constructor for JSON Parsing
public JwtRequest()
{

}

public JwtRequest(String username, String password) {
this.setUsername(username);
this.setPassword(password);
}

public String getUsername() {
return this.username;
}

public void setUsername(String username) {
this.username = username;
}

public String getPassword() {
return this.password;
}

public void setPassword(String password) {
this.password = password;
}
}
```

JwtResponse

This class is required for creating a response containing the JWT to be returned to the user.

```
package com.javainuse.model;

import java.io.Serializable;

public class JwtResponse implements Serializable {

private static final long serialVersionUID = -8091879091924046844L;
private final String jwttoken;

public JwtResponse(String jwttoken) {
this.jwttoken = jwttoken;
}

public String getToken() {
return this.jwttoken;
}
}
```

JwtRequestFilter

The `JwtRequestFilter` extends the Spring Web Filter `OncePerRequestFilter` class. For any incoming request, this `Filter` class gets executed. It checks if the request has a valid JWT token. If it has a valid JWT Token, then it sets the authentication in context to specify that the current user is authenticated.

```
package com.javainuse.config;

import java.io.IOException;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import com.javainuse.service.JwtUserDetailsService;

import io.jsonwebtoken.ExpiredJwtException;

@Component
public class JwtRequestFilter extends OncePerRequestFilter {

@Autowired
private JwtUserDetailsService jwtUserDetailsService;

@Autowired
private JwtTokenUtil jwtTokenUtil;

@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
throws ServletException, IOException {

final String requestTokenHeader = request.getHeader("Authorization");

String username = null;
String jwtToken = null;
// JWT Token is in the form "Bearer token". Remove Bearer word and get
// only the Token
if (requestTokenHeader != null && requestTokenHeader.startsWith("Bearer ")) {
jwtToken = requestTokenHeader.substring(7);
try {
username = jwtTokenUtil.getUsernameFromToken(jwtToken);
} catch (IllegalArgumentException e) {
System.out.println("Unable to get JWT Token");
} catch (ExpiredJwtException e) {
System.out.println("JWT Token has expired");
}
} else {
logger.warn("JWT Token does not begin with Bearer String");
}

// Once we get the token validate it.
if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

UserDetails userDetails = this.jwtUserDetailsService.loadUserByUsername(username);

// if token is valid configure Spring Security to manually set
// authentication
if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {

UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(
userDetails, null, userDetails.getAuthorities());
usernamePasswordAuthenticationToken
.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
// After setting the Authentication in the context, we specify
// that the current user is authenticated. So it passes the
// Spring Security Configurations successfully.
SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
}
}
chain.doFilter(request, response);
}

}
```

JwtAuthenticationEntryPoint

JwtAuthenticationEntryPoint

This class will extend Spring's `AuthenticationEntryPoint` class and override its method to commence. It rejects every unauthenticated request and sends error code 401.

```
package com.javainuse.config;

import java.io.IOException;
import java.io.Serializable;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint, Serializable {

private static final long serialVersionUID = -7858869558953243875L;

@Override
public void commence(HttpServletRequest request, HttpServletResponse response,
AuthenticationException authException) throws IOException {

response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized");
}
}
```

WebSecurityConfig

This class extends the `WebSecurityConfigurerAdapter`. This is a convenience class that allows customization to both `WebSecurity` and `HttpSecurity`.

```
package com.javainuse.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
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.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

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

@Autowired
private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

@Autowired
private UserDetailsService jwtUserDetailsService;

@Autowired
private JwtRequestFilter jwtRequestFilter;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
// configure AuthenticationManager so that it knows from where to load
// user for matching credentials
// Use BCryptPasswordEncoder
auth.userDetailsService(jwtUserDetailsService).passwordEncoder(passwordEncoder());
}

@Bean
public PasswordEncoder passwordEncoder() {
return new BCryptPasswordEncoder();
}
```

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

@Override
protected void configure(HttpSecurity httpSecurity) throws Exception {
// We don't need CSRF for this example
httpSecurity.csrf().disable()
// dont authenticate this particular request
.authorizeRequests().antMatchers("/authenticate").permitAll().
// all other requests need to be authenticated
anyRequest().authenticated().and().
// make sure we use stateless session; session won't be used to
// store user's state.
exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint).and().sessionManagement()
.sessionCreationPolicy(SessionCreationPolicy.STATELESS);

// Add a filter to validate the tokens with every request
httpSecurity.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);
}
}
```

Then, start the Spring Boot application.

Generate a JSON Web Token

Create a POST request with URL localhost:8080/authenticate. The body should have a valid username and password. In our case, the username is javainuse and the password is password.

Spring Boot JWT Tutorial

Validate the JSON Web Token

Try accessing the URL localhost:8080/hello using the above-generated token in the header as follows:

Spring Boot JSON Web Token

And there you have it! We hope you enjoyed this demonstration on how to implement Spring Boot security via a JSON Web Token (JWT).

Topics: SPRING BOOT, JSON WEB TOKEN, JWT AUTHENTICATION, HELLO WORLD, SPRING BOOT SECURITY OAUTH2, SPRING BOOT SECURITY, SPRING SECURITY 5, SPRING SECURITY OAUTH, SECURITY

Published at DZone with permission of Rida Shaikh. See the original article here. (https://www.javainuse.com/spring/boot-jwt)
Opinions expressed by DZone contributors are their own.

Popular on DZone

- **Postgres Query Execution: JDBC Prepared Statements (/articles/postgres-query-execution-jdbc-prepared-statements?fromrel=true)**
- **Advanced PostgreSQL Features: A Guide (/articles/advanced-postgresql-features-a-guide?fromrel=true)**
- **How To Use SQL Subqueries (/articles/how-to-use-sql-subqueries?fromrel=true)**
- **Tips Every Spring Boot Developer Should Know (/articles/few-things-that-every-spring-boot-developer-should?fromrel=true)**