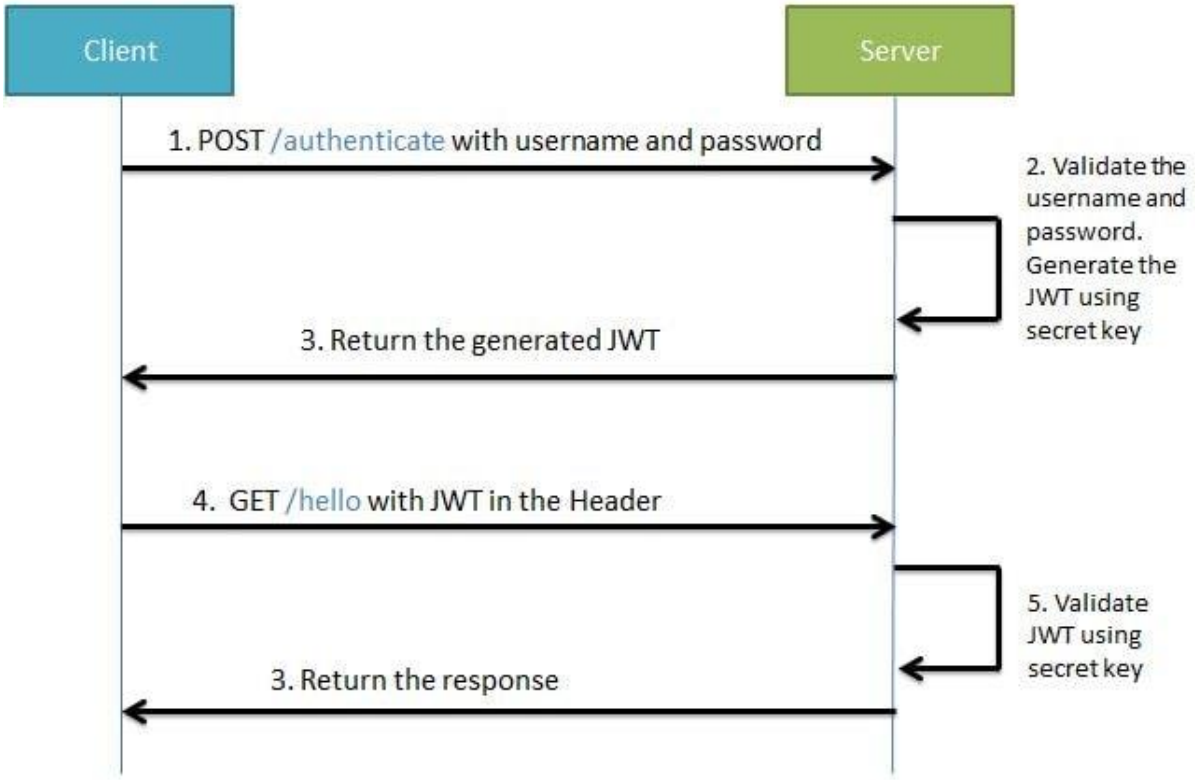
Spring Boot Security + JWT Hello World Example

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 next tutorial we will be implementing

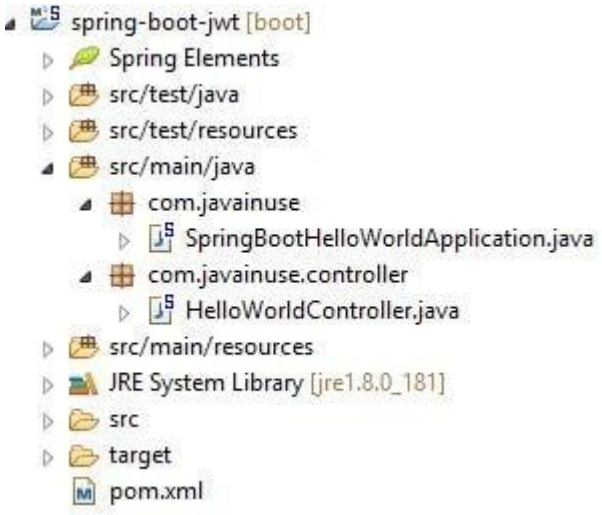
For better understanding we will be developing the project in stages

- Develop a Spring Boot Application to expose 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



Develop a Spring Boot Application to expose a GET REST API

Maven Project will be as follows-



The pom.xml is as follows-

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/20
01/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.output
Encoding>
                <java.version>1.8</java.version>
        </properties>

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

</project>
```

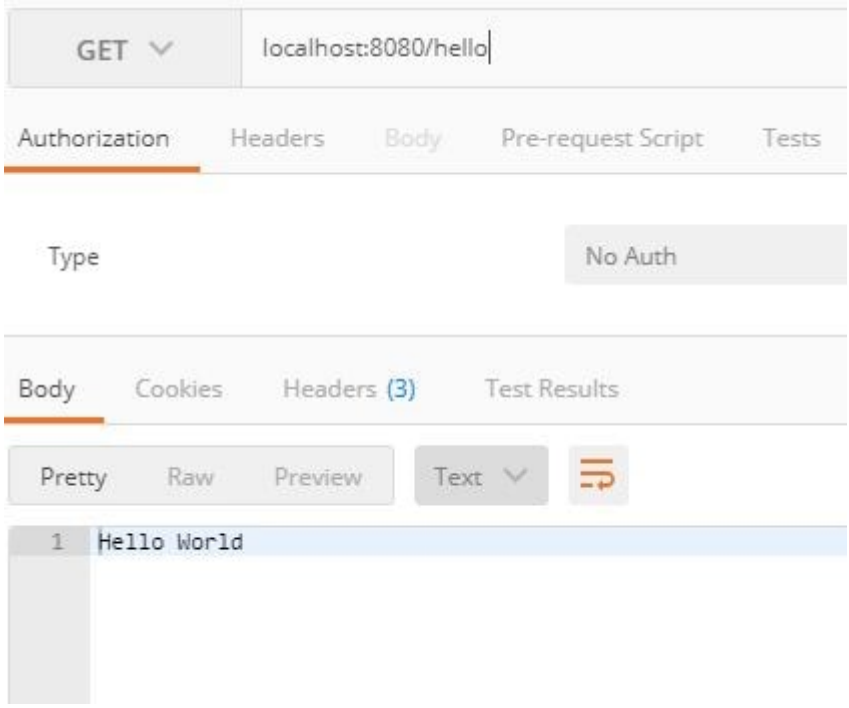
Create a Controller class for exposing a GET REST API-

```
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 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 the run the SpringBootHelloWorldApplication.java as a Java application.
Go to **localhost:8080/hello**

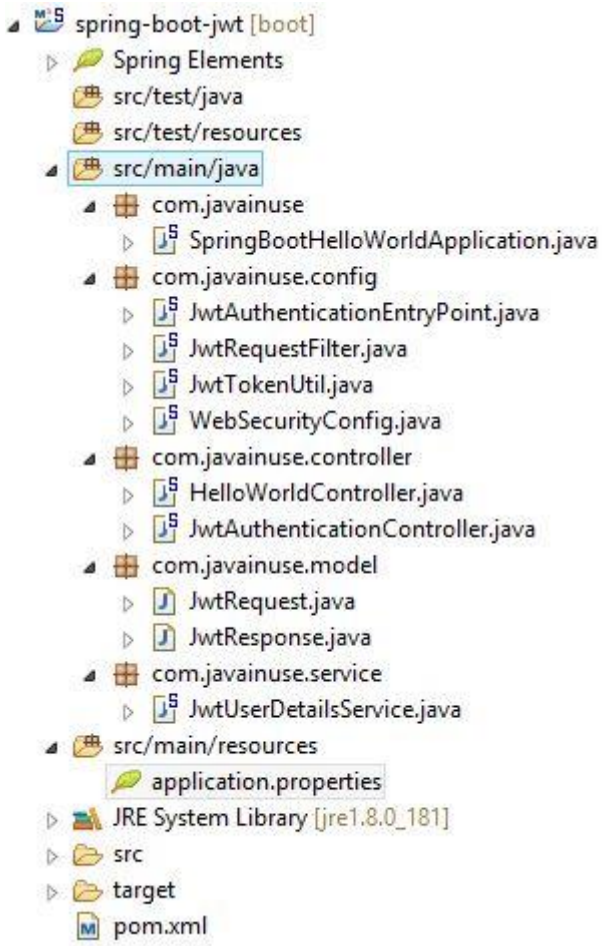


Spring Security and JWT Configuration

We will be configuring Spring Security and JWT for performing 2 operations-

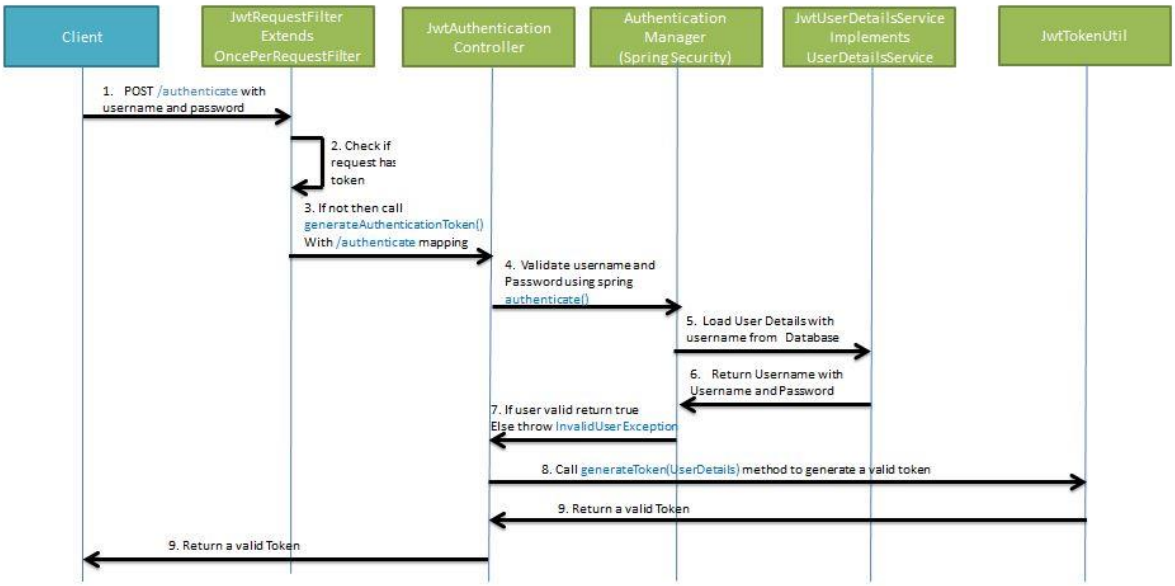
- **Generating JWT** - Expose a POST API with mapping **/authenticate**. On passing correct username and password it will generate a JSON Web Token(JWT)
- **Validating JWT** - If user tries to access GET API with mapping **/hello**. It will allow access only if request has a valid JSON Web Token(JWT)

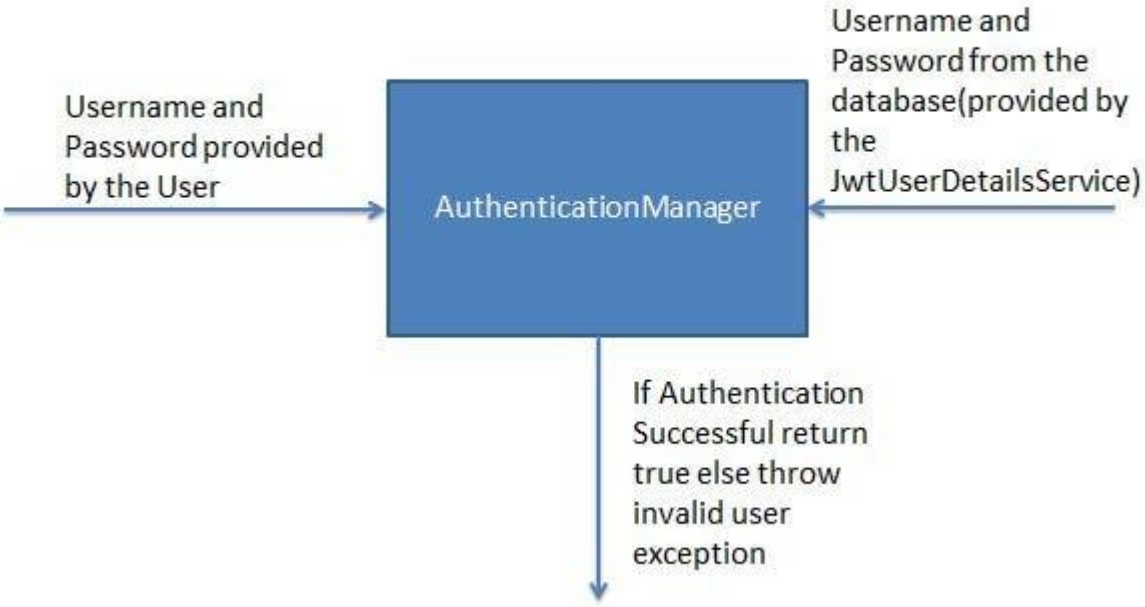
Maven Project will be as follows-



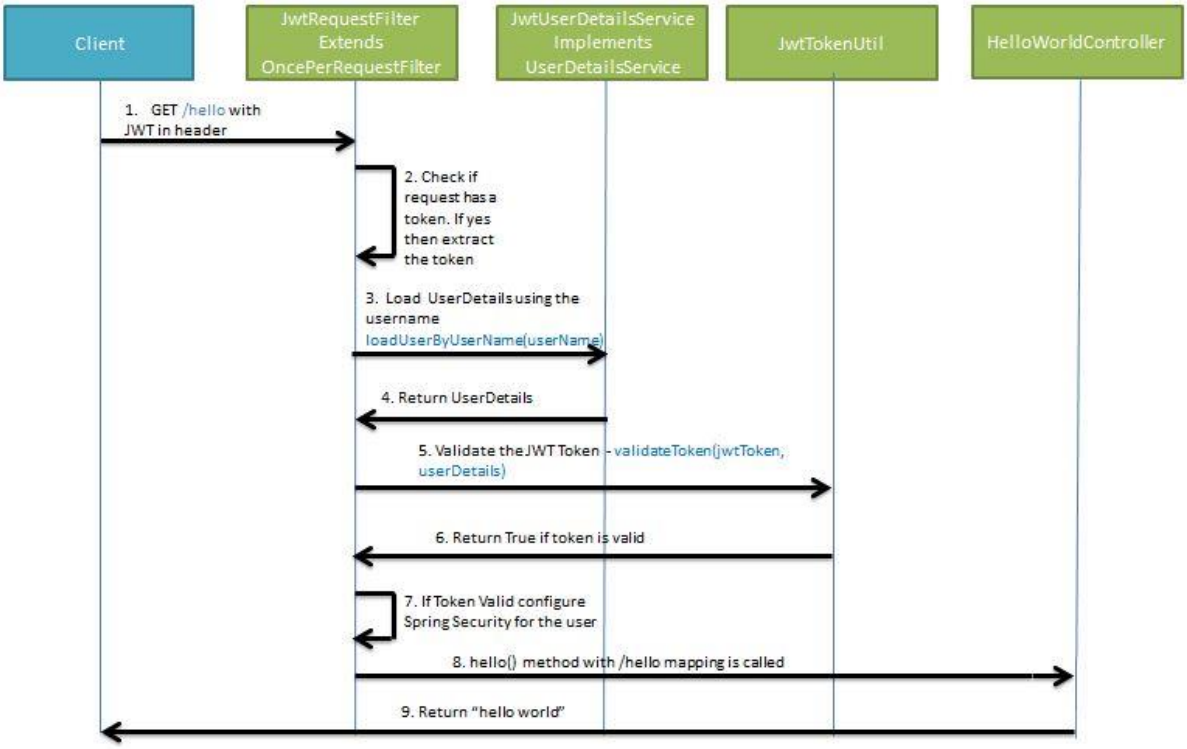
The sequence flow for these operations will be as follows-

Generating JWT





Validating JWT



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.output
Encoding>
                <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 using which we will be using for hashing algorithm. 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> claimsRe
solver) {
		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 t
he ID
	//2. Sign the JWT using the HS512 algorithm and secret key.
	//3. According to JWS Compact Serialization(https://tools.ietf.org/html/d
raft-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).setIs
suedAt(new Date(System.currentTimeMillis()))
				.setExpiration(new Date(System.currentTimeMillis
() + JWT_TOKEN_VALIDITY * 1000))
				.signWith(SignatureAlgorithm.HS512, secret).comp
act();
	}

	//validate token
```

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

- JWTUserDetailsService

  JWTUserDetailsService implements the Spring Security UserDetailsService interface. It overrides the loadUserByUsername for fetching user details from the database using the username. The Spring Security Authentication Manager calls this method for getting the user details from the database when authenticating the user details provided by the user. Here we are getting the **user details from a hardcoded User List**. In the next tutorial we will be adding the DAO implementation for fetching User Details from the Database. Also the password for a user is stored in encrypted format using BCrypt. Previously we have seen Spring Boot Security - Password Encoding Using Bcrypt. Here using the Online Bcrypt Generator you can generate the Bcrypt for a password.

```
package com.javainuse.service;

import java.util.ArrayList;

import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class JwtUserDetailsService implements UserDetailsService {

	@Override
	public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
```

```
		if ("javainuse".equals(username)) {
			return new User("javainuse", "$2a$10$slYQmyNdGzTn7ZLBXBChFOC9f6kFjAqPhccnP6DxlWXx2lPk1C3G6",
					new ArrayList<>());
		} else {
			throw new UsernameNotFoundException("User not found with username: " + username);
		}
	}
}
```

- JwtAuthenticationController

  Expose a POST API /authenticate using the JwtAuthenticationController. The POST API gets username and password in the body- Using Spring Authentication Manager we authenticate the username and password.If the credentials are valid, a JWT token is created using the JWTTokenUtil and 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 JwtReques
t authenticationRequest) throws Exception {

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

		final UserDetails userDetails = userDetailsService
				.loadUserByUsername(authenticationRequest.getUse
rname());

		final String token = jwtTokenUtil.generateToken(userDetails);

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

	private void authenticate(String username, String password) throws Except
ion {
		try {
```

```
                        authenticationManager.authenticate(new UsernamePasswordAu
thenticationToken(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 recieve 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 is 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 the 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.UsernamePasswordAuthenticati
onToken;

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

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.web.authentication.WebAuthenticationDetails
Source;

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, HttpServletRe
sponse response, FilterChain chain)
			throws ServletException, IOException {


		final String requestTokenHeader = request.getHeader("Authorizatio
n");


		String username = null;
		String jwtToken = null;
		// JWT Token is in the form "Bearer token". Remove Bearer word an
d get
		// only the Token
		if (requestTokenHeader != null && requestTokenHeader.startsWith("
Bearer ")) {
			jwtToken = requestTokenHeader.substring(7);
			try {
				username = jwtTokenUtil.getUsernameFromToken(jwt
Token);
			} 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().getAut
hentication() == null) {


			UserDetails userDetails = this.jwtUserDetailsService.load
UserByUsername(username);
```

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

                                UsernamePasswordAuthenticationToken usernamePass
wordAuthenticationToken = new UsernamePasswordAuthenticationToken(
                                                userDetails, null, userDetails.g
etAuthorities());
                                usernamePasswordAuthenticationToken
                                                .setDetails(new WebAuthenticatio
nDetailsSource().buildDetails(request));
                                // After setting the Authentication in the conte
xt, we specify
                                // that the current user is authenticated. So it
passes the
                                // Spring Security Configurations successfully.
                                SecurityContextHolder.getContext().setAuthentica
tion(usernamePasswordAuthenticationToken);
                        }
                }
                chain.doFilter(request, response);
        }

}
```

- JwtAuthenticationEntryPoint

  This class will extend Spring's AuthenticationEntryPoint class and override its method commence. It rejects every unauthenticated request and send 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, S
erializable {


      private static final long serialVersionUID = -7858869558953243875L;


      @Override

      public void commence(HttpServletRequest request, HttpServletResponse resp
onse,

                        AuthenticationException authException) throws IOException
{


               response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthor
ized");

      }

}
```

- WebSecurityConfig

  This class extends the WebSecurityConfigurerAdapter 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.A
uthenticationManagerBuilder;

import org.springframework.security.config.annotation.method.configuration.Enab
leGlobalMethodSecurity;

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

```
import org.springframework.security.config.annotation.web.configuration.EnableW
ebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecu
rityConfigurerAdapter;

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.UsernamePasswordAuthenti
cationFilter;


@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 Exc
eption {

		// configure AuthenticationManager so that it knows from where to
load

		// user for matching credentials

		// Use BCryptPasswordEncoder

		auth.userDetailsService(jwtUserDetailsService).passwordEncoder(pa
sswordEncoder());

	}


	@Bean

	public PasswordEncoder passwordEncoder() {

		return new BCryptPasswordEncoder();
```

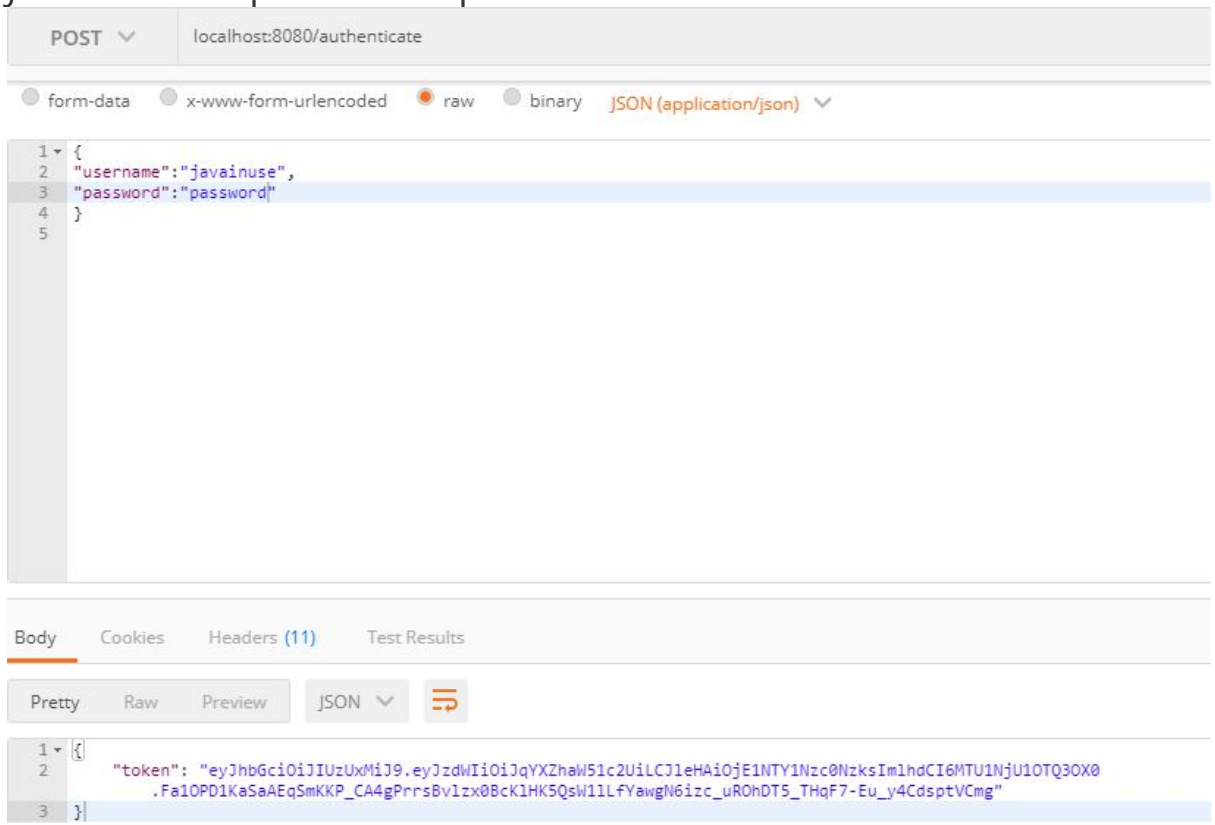
```
	}


	@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 w
on't be used to
				// store user's state.
				exceptionHandling().authenticationEntryPoint(jwt
AuthenticationEntryPoint).and().sessionManagement()
				.sessionCreationPolicy(SessionCreationPolicy.STA
TELESS);


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

Start the Spring Boot Application

- Generate a JSON Web Token -

  Create a POST request with url localhost:8080/authenticate. Body should have valid username and password. In our case username is

javainuse and password is password.



- Validate the JSON Web Token

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