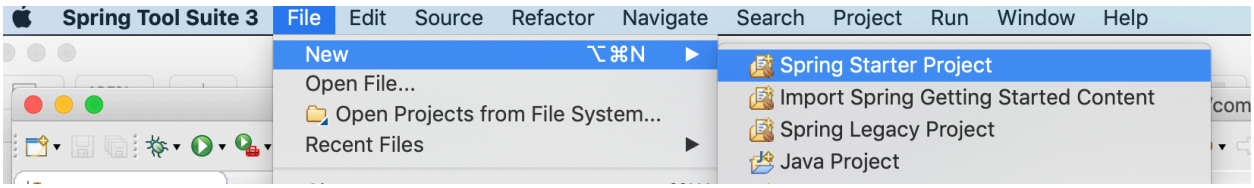
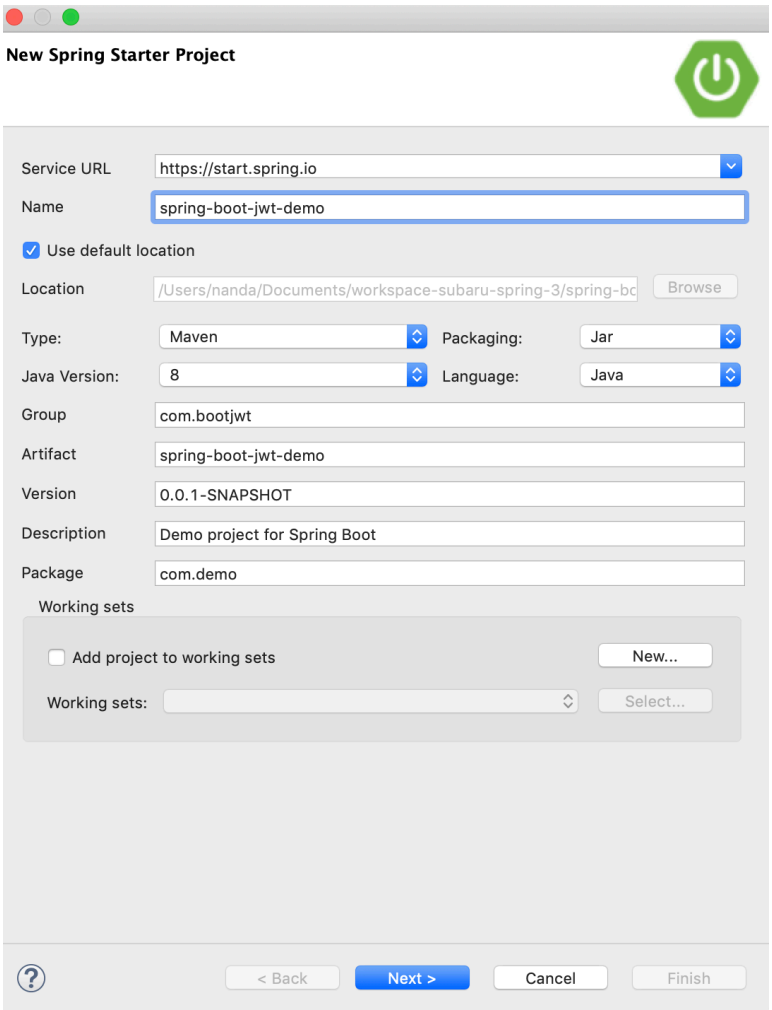
Create Spring Boot Application to use JSON Web Tokens (JWT) for REST APIs

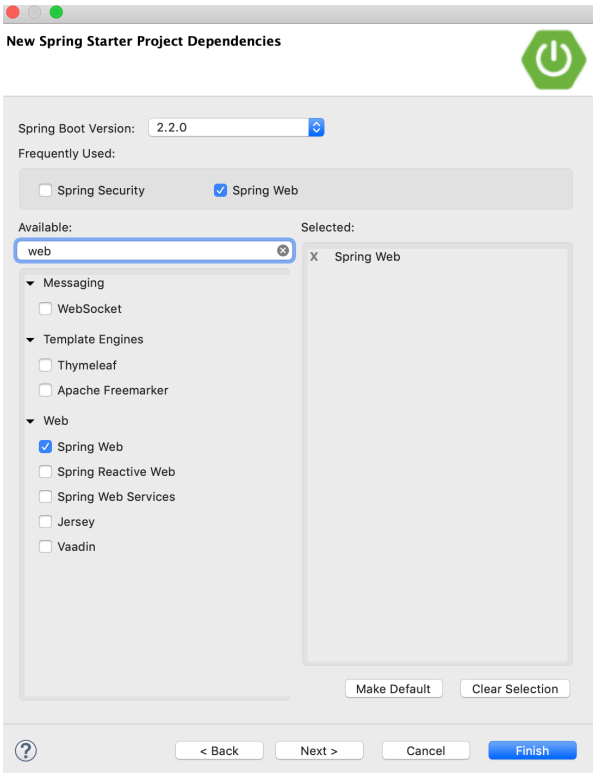
1. Create a simple Spring Boot App to start with.



2.



3. Only **Spring Web** module is required



4. Create a Controller class for exposing a GET REST API-

HelloWorldController.java

```
package com.demo.controller;

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

@RestController
@CrossOrigin()
public class HelloWorldController {

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

}
```

5. RUN the APP

Main Class
    –> Run As
        –> Spring Boot App

6. Open POSTMAN

7. Test for **localhost:8080/hello** in POSTMAN

It looks good!

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)

8. Add the Spring Security and JWT dependencies

```
<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>
```

9. Secret key in application.properties.

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

application.properties
key.jwt.secret=javainuse

10.
JwtTokenUtil

The **JwtTokenUtil** is responsible for performing JWT operations like creation and validation of the Token.

It makes use of the **io.jsonwebtoken.Jwts** for achieving this.

JwtTokenUtil.java

```
package com.demo.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;

	public String getUsernameFromToken(String token) {
		return getClaimFromToken(token, Claims::getSubject);
	}

	public Date getIssuedAtDateFromToken(String token) {
		return getClaimFromToken(token, Claims::getIssuedAt);
	}

	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);
	}
```

```
private Claims getAllClaimsFromToken(String token) {
            return
Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();

      }

private Boolean isTokenExpired(String token) {
            final Date expiration = getExpirationDateFromToken(token);
            return expiration.before(new Date());
      }

      private Boolean ignoreTokenExpiration(String token) {
            // here you specify tokens, for that the expiration is ignored
            return false;
      }

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

      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();
      }

      public Boolean canTokenBeRefreshed(String token) {
            return (!isTokenExpired(token) || ignoreTokenExpiration(token));
      }

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

11. 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.

```
package com.demo.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);
		}
	}

}
```

Here we are getting the **user details from a hardcoded User List.**

JWTUserDetailsService.java

12.

```java
package com.demo.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;
	}
}
```

JwtRequest

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

JwtRequest.java

13.

```
package com.demo.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;
	}
}
```

JwtResponse

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

JwtResponse.java

14.

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.

```java
package com.demo.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.security.core.userdetails.UserDetailsService;
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.demo.config.JwtTokenUtil;
import com.demo.model.JwtRequest;
import com.demo.model.JwtResponse;

@RestController
@CrossOrigin
public class JwtAuthenticationController {

	@Autowired
	private AuthenticationManager authenticationManager;

	@Autowired
	private JwtTokenUtil jwtTokenUtil;

	@Autowired
	private UserDetailsService jwtInMemoryUserDetailsService;

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

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

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

		final String token = jwtTokenUtil.generateToken(userDetails);

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

	private void authenticate(String username, String password) throws Exception {
		Objects.requireNonNull(username);
		Objects.requireNonNull(password);

		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);
		}
	}
}
```

JwtAuthenticationController.java

15.
JwtRequestFilter

The JwtRequestFilter extends the Spring Web Filter OncePerRequestFilter class. For any incoming request this Filter class gets executed.

```
package com.demo.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.UsernamePasswordAuthenticationTo
ken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.web.authentication.WebAuthenticationDetailsSour
ce;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import com.demo.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(usernamePasswordAuthenticationT
oken);
                     }
              }
              chain.doFilter(request, response);
       }

}
```

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.

JwtRequestFilter.java

16. **JwtAuthenticationEntryPoint**

```
package com.demo.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");
	}
}
```

This class will extend Spring's AuthenticationEntryPoint class and override its method commence.

It rejects every unauthenticated request and send error code 401

JwtAuthenticationEntryPoint.java

17.
WebSecurityConfig

```
package com.demo.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.Authe
nticationManagerBuilder;
import
org.springframework.security.config.annotation.method.configuration.EnableGl
obalMethodSecurity;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSe
curity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurity
ConfigurerAdapter;
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.UsernamePasswordAuthenticati
onFilter;

@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(passwordEncod
er());
	}

	@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).an
d().sessionManagement()
				.sessionCreationPolicy(SessionCreationPolicy.STATELE
SS);

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

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

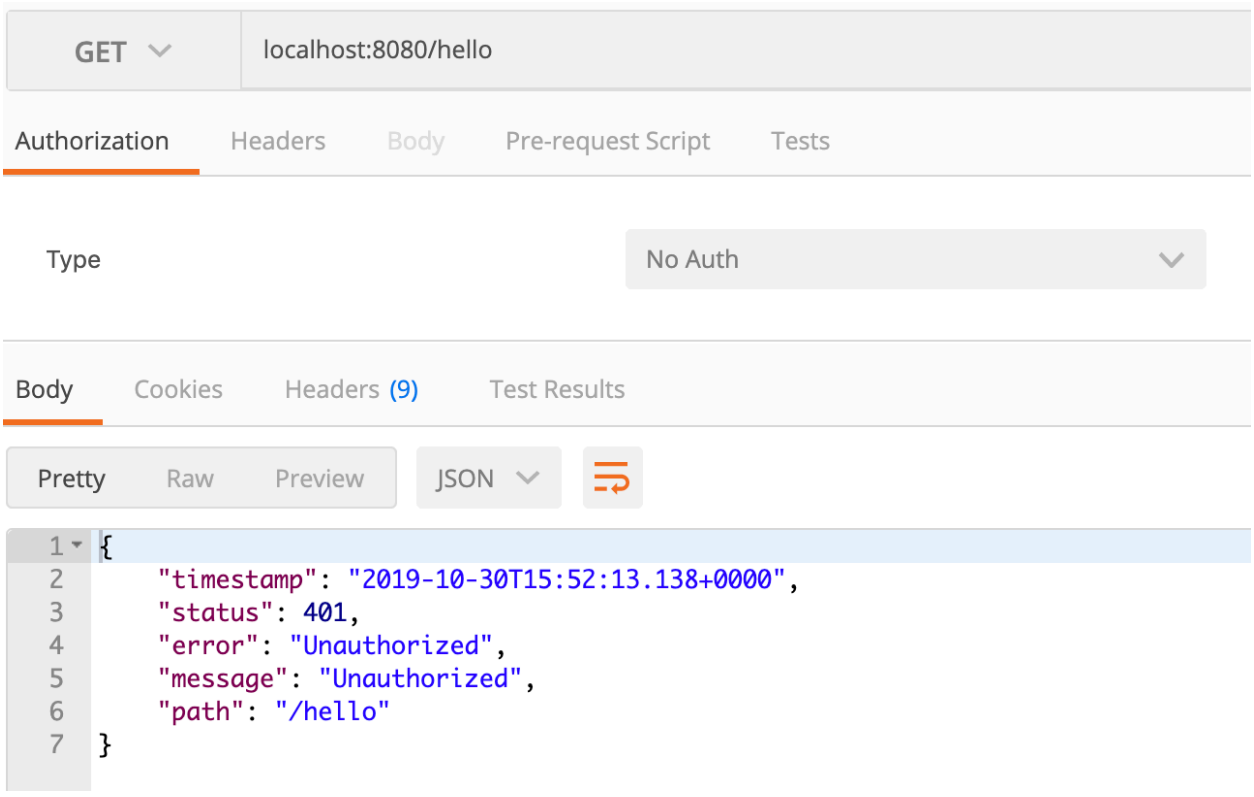
WebSecurityConfig.java

18.
Start the Spring Boot Application

```
Main Class
     -> Run As
          -> Spring Boot App
```

19. **Open POSTMAN**

Visit URL: ***localhost:8080/hello***



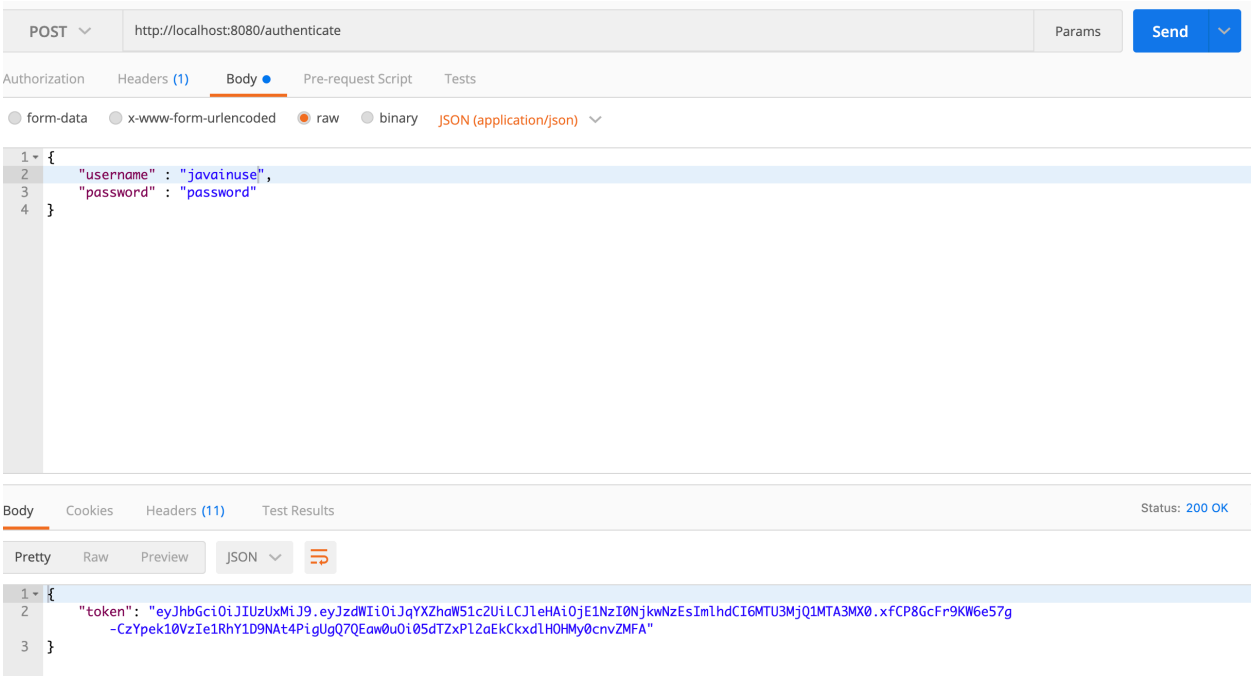
20.

Generate a JSON Web Token –

Create a POST request with url
http://localhost:8080/authenticate

Body should have valid username and password.

In our case username is **javainuse** and password is **password**.

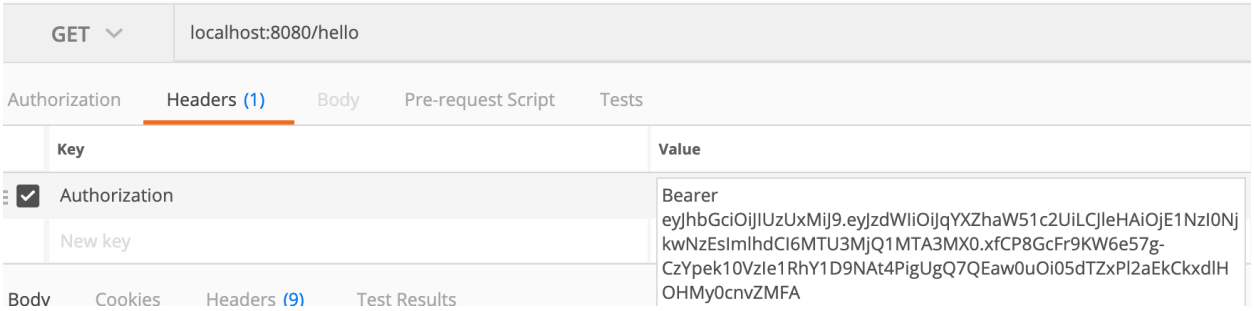


COPY & SAVE this token locally

21.

Add JSON Web Token to GET request

Go back to the GET call tab in POSTMAN & click on **Headers** Tab



Create a Key called **Authorization**

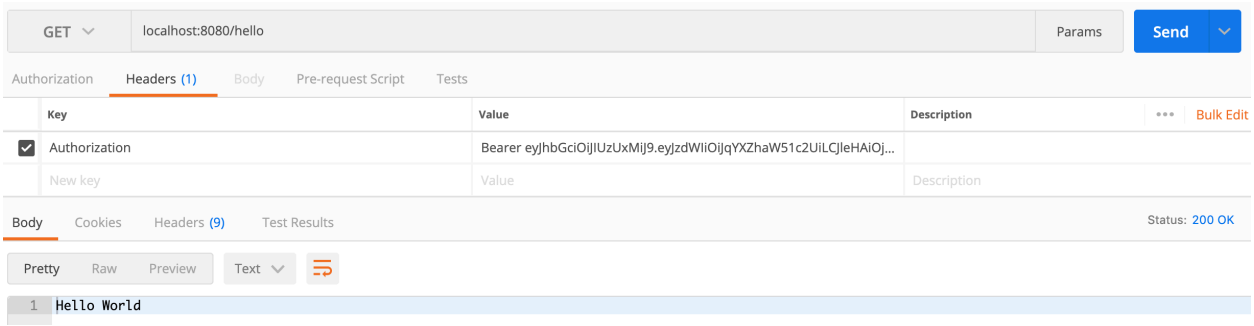
In the **Value** field, first type the string **“Bearer “**

And then Paste the token generated by **/authenticate** in the **Value** field of **Authorization** after the **“Bearer “** string as shown above.

NOTE: Prefix your token in the Value field with “Bearer “

22.

Now Click on **Send** Button and you should see the API response successfully!


GET
localhost:8080/hello
Params
Send
Authorization
Headers (1)
Body
Pre-request Script
Tests
Key
Value
Description
Bulk Edit
Authorization
Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJqYXZhaW51c2UiLCJleHAiOj...
New key
Value
Description
Body
Cookies
Headers (9)
Test Results
Status: 200 OK
Pretty
Raw
Preview
Text
1
Hello World