**Spring JWT**

**(**[**https://www.javainuse.com/spring/jwt**](https://www.javainuse.com/spring/jwt)**)**

**(https://www.akana.com/blog/what-is-jwt)**

**What is JWT**

JWT, or JSON Web Token, is an open standard used to share security information between two parties — a client and a server. This information can be verified and trusted because it is digitally signed. The client will need to authenticate with the server using the credentials only once. The server validates the credentials and returns the client a JSON Web Token(JWT). For all future requests the client can authenticate itself to the server using this JSON Web Token(JWT) and so does not need to send the credentials like username and password.

**What Are Tokens?**

A token is a string of data that represents something else, such as an identity.

**Why Use JWT?**

JWTs are used as a secure way to authenticate users and share information.

Typically, a private key, or secret, is used by the issuer to sign the JWT. The receiver of the JWT will verify the signature to ensure that the token hasn’t been altered after it was signed by the issuer. It is difficult for unauthenticated sources to guess the signing key and attempt to change the claims within the JWT.

The server that issues the JWT and the server that validates it do not have to be the same.

**Structure of JWT**

JWT has the following format -**header.payload.signature**  


**(**[**https://www.tutorialspoint.com/spring\_security/spring\_security\_with\_jwt.htm**](https://www.tutorialspoint.com/spring_security/spring_security_with_jwt.htm)**)**

**Header** − The Header of a JWT token contains the name of cryptographic operations that are applied to the JWT. This can be the signing technique, metadata information about the content-type and so on. The header is presented as a JSON object which is encoded to a base64URL. An example of a valid JWT header would be

{ "alg": "HS256", "typ": "JWT" }

It has two attributes – “**alg**”: gives us information about the type of algorithm used.

“**typ”:** gives us the type of the information.

**Payload** − The payload part of JWT contains the actual data to be transferred using the token. This part is also known as the “claims” part of the JWT token. The claims can be of three types – registered, public and private.

* The registered claims are the ones which are recommended but not mandatory claims such as iss(issuer), sub(subject), aud(audience) and others.
* Public claims are those that are defined by those using the JWTs.
* Private claims or custom claims are user-defined claims created for the purpose of sharing the information between the concerned parties.

Example of a payload object could be.

{ "sub": "12345", "name": "Johnny Hill", "admin": false }

The payload object, like the header object is base64Url encoded as well and this string forms the second part of the JWT.

**Signature**− The signature part of the JWT is used for the verification that the message wasn’t changed along the way. If the tokens are signed with private key, it also verifies that the sender is who it says it is. It is created using the encoded header, encoded payload, a secret and the algorithm specified in the header.

If we put the header, payload and signature we get a token as given below.

eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I

kpvaG4gRG9lIiwiYWRtaW4iOmZhbHNlfQ.gWDlJdpCTIHVYKkJSfAVNUn0ZkAjMxskDDm-5Fhe

WJ7xXgW8k5CllcGk4C9qPrfa1GdqfBrbX\_1x1E39JY8BYLobAfAg1fs\_Ky8Z7U1oCl6HL63yJq\_

wVNBHp49hWzg3-ERxkqiuTv0tIuDOasIdZ5FtBdtIP5LM9Oc1tsuMXQXCGR8GqGf1Hl2qv8MCyn

NZJuVdJKO\_L3WGBJouaTpK1u2SEleVFGI2HFvrX\_jS2ySzDxoO9KjbydK0LNv\_zOI7kWv-gAmA

j-v0mHdJrLbxD7LcZJEGRScCSyITzo6Z59\_jG\_97oNLFgBKJbh12nvvPibHpUYWmZuHkoGvuy5RLUA

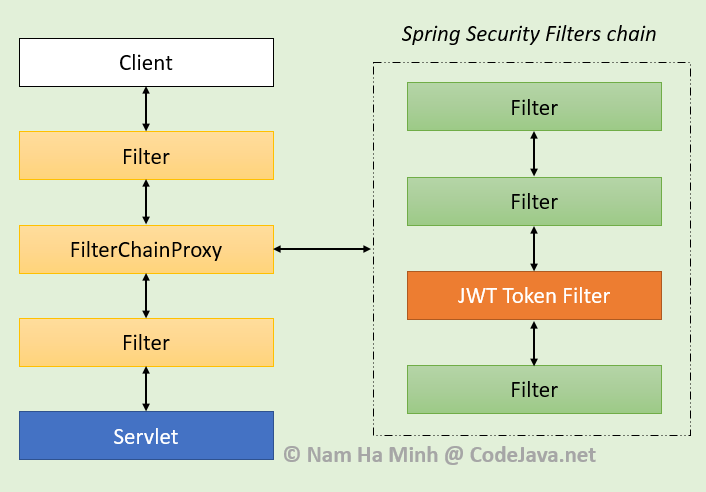
Now, this token can be used in the Authorization header using the Bearer schema.

**Authorization − Bearer** <token>

The use of JWT token for authorization is the most common of its applications. The token is usually generated in the server and sent to the client where it is stored in the session storage or local storage. To access a protected resource the client would send the JWT in the header as given above.

**(**[**https://www.codejava.net/frameworks/spring-boot/spring-security-jwt-authentication-tutorial**](https://www.codejava.net/frameworks/spring-boot/spring-security-jwt-authentication-tutorial)**)****Understand Spring Security Filters chain**

How Spring Security works? When a client sends a request to the server, the request will go through a sequence of filters before reaching the destination servlet which is actually responsible for processing the request. Let’s look at the following diagram:



The Spring Web framework plugs in a special filter called *FilterChainProxy* that picks a chain of internal filters used by Spring Security, based on the application’s security configuration. Each filter is responsible for applying a specific security concern to the current request. If you enable debugging for a security configuration class like this:

|  |  |
| --- | --- |
| 1  2 | @EnableWebSecurity(debug = **true**)  **public** **class** AppSecurityConfig  **extends** WebSecurityConfigurerAdapter { ... } |

Then you will see, in the console, it reveals all the filters involved in the current request - for example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Security filter chain:    WebAsyncManagerIntegrationFilter    SecurityContextPersistenceFilter    HeaderWriterFilter    RequestCacheAwareFilter    SecurityContextHolderAwareRequestFilter    AnonymousAuthenticationFilter    SessionManagementFilter    ExceptionTranslationFilter    FilterSecurityInterceptor  ] |

This list may differ, depending on security configuration.The filters used by Spring Security are internal to the framework and the container is not aware of them.

If we need to insert our own custom filter (e.g. JWT Token Filter) in the middle of Spring Security filters chain. This filter will check availability and verify integrity of the access token. If the token is verified, the request is passed through the downstream filters and finally reaching the destination handler. Otherwise, an Unauthorized error should be raised.

**Required Dependencies (in POM.xml):**

Make sure that you have the following dependencies included in the pom.xml file:

* **spring-boot-starter-security**: used to apply security to the application
* **spring-boot-starter-web**: used to implement RESTful webservices, REST APIs
* **jjwt**: is the JWT library which we use to generate and verity JWT tokens
* **spring-boot-starter-validation**: used to validate values of a JavaBean’s fields which are JSON values in the request.
* **spring-boot-starter-test**: used for testing with JUnit and AssertJ
* **jaxb-api:** To initialize class javax.xml.bind.DatatypeConverterImpl (for Java11 & above)

**Extra Classes other than Normal Spring Boot classes:**

* **JwtTokenUtil**: It is responsible for performing JWT operations like creation and validation.
* **JWTUserDetailsService***:* It 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. Also the password for a user is stored in encrypted format .
* **JwtAuthenticationController**: It exposes 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.
* **JwtRequest:** This class is required for storing the username and password we receive from the client.
* **JwtResponse**: This is class is required for creating a response containing the JWT to be returned to the user.
* **JwtRequestFilter**: JwtRequestFilter extends *OncePerRequestFilter* class. For any incoming request this Filter class gets executed, it checks the request has a valid JWT token. If it has a valid then it sets the Authentication in the context, to specify that the current user is authenticated.
* **JwtAuthenticationEntryPoint:** This class extends AuthenticationEntryPoint class and override method commence. It rejects every unauthenticated request and send error code 401.
* **EncoderConfig:** To configure the BCryptEncoder class.
* **WebSecurityConfig:** This class extends the WebSecurityConfigurerAdapter is a convenience class that allows customization to both WebSecurity and HttpSecurity.

**Configure application.properties**:

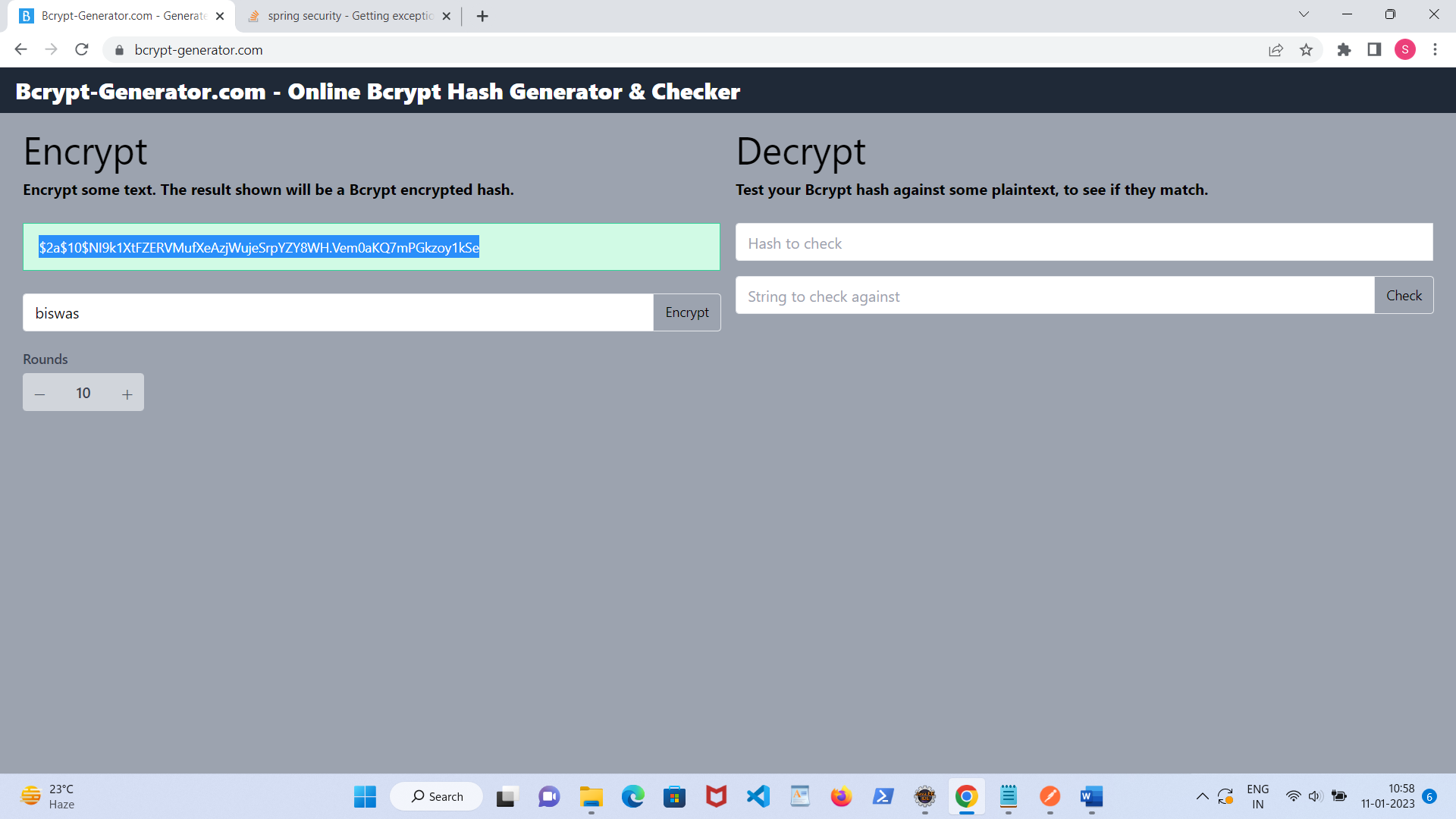
We need to configure this file like below:

*server.port=9596*

*jwt.secret=suramya*

The @Value("${jwt.secret}") is used in JwtTokenUil class & JwtUserDetailService class.

**Steps:**

* Let us consider the password is “biswas”.
* Go for BCrypt encode the password from online BCryptEncoder.
* 
* Put this encoded password into JwtUserDetailsService class as follows:

*@Service*

*public class JwtUserDetailsService implements UserDetailsService{*

*@Override*

*public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {*

*if ("suramya".equals(username)) {*

*return new User("suramya", "$2a$10$NI9k1XtFZERVMufXeAzjWujeSrpYZY8WH.Vem0aKQ7mPGkzoy1kSe",*

*new ArrayList<>());*

*}*

*else {*

*throw new UsernameNotFoundException("User not found with username: " + username);*

*}}}*

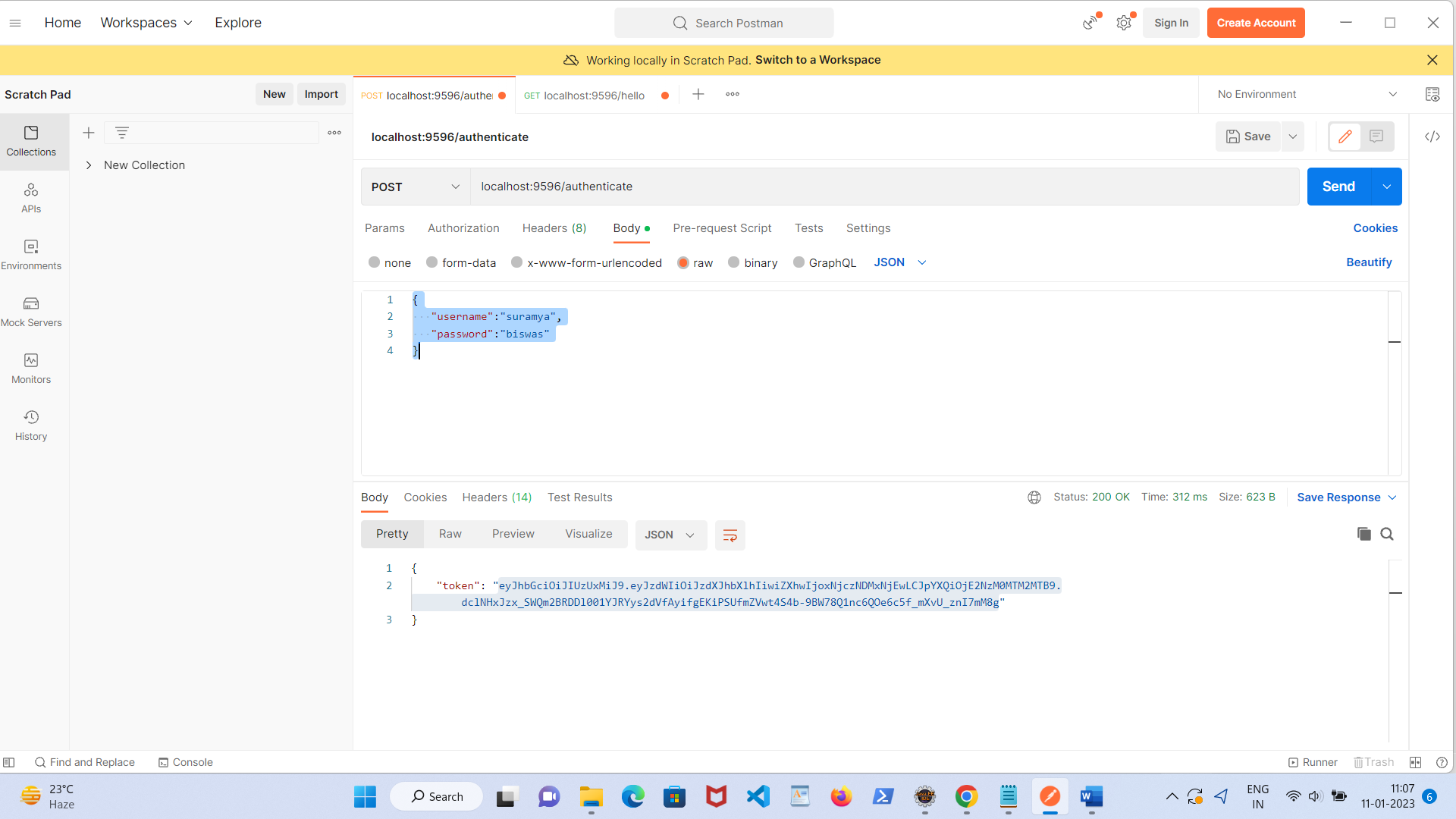
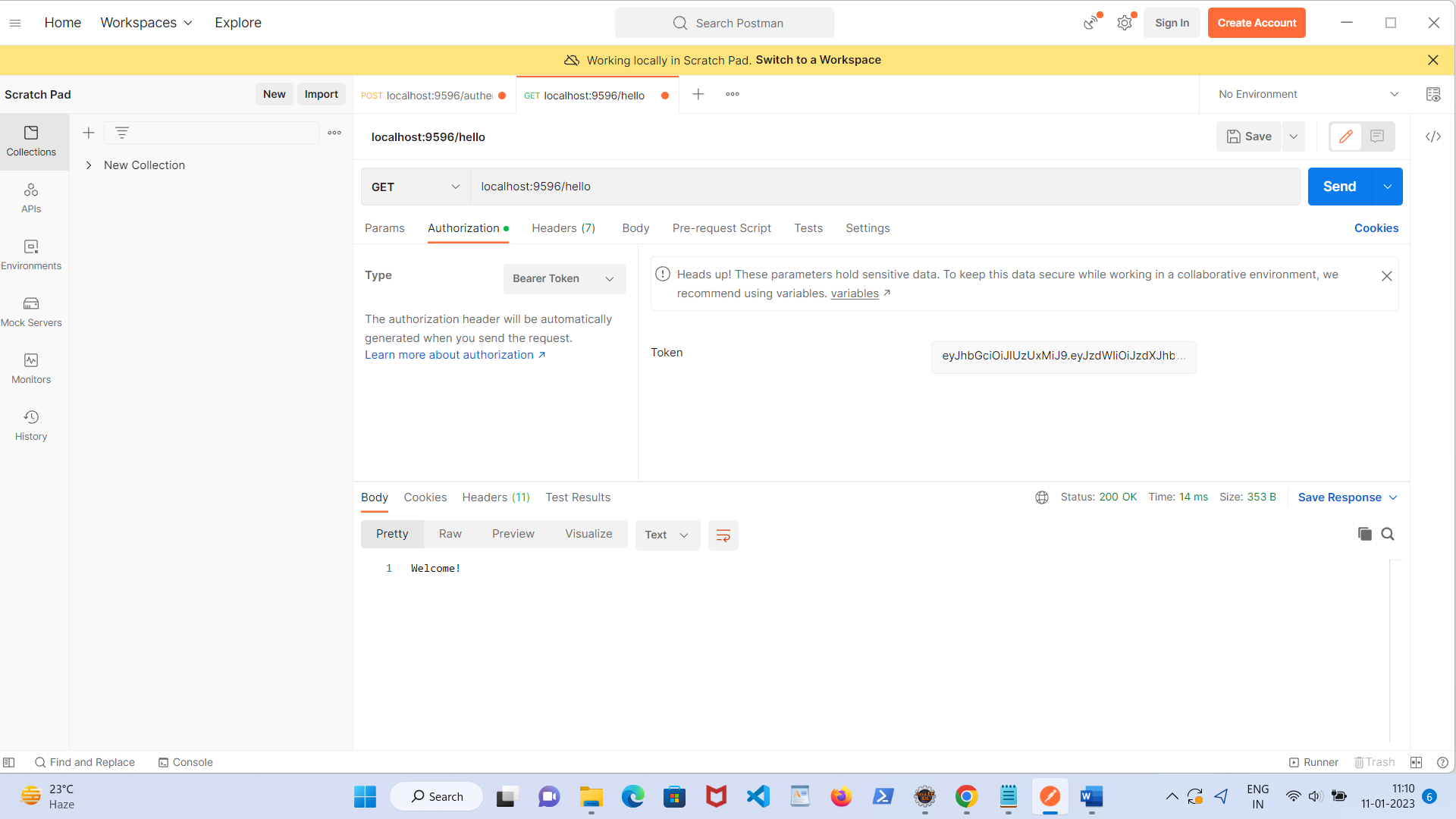
* Go to the Postman make a post request for /authenticate where in json format you put

{

"username":"suramya",

"password":"biswas"

}

* It will generate a JWT token
* 
* Now go for the get request for /hello, in Authorization for Bearer paste the token, you will see the result of get for hello.
* 
* If we decode the jwt token (<http://calebb.net/>) it will looks like below:  
   {

  alg: "HS512"

}.

{

  sub: "suramya",

  exp: 1673535895,

  iat: 1673517895

}.

[signature]

**JwtAuthentication with Database:**

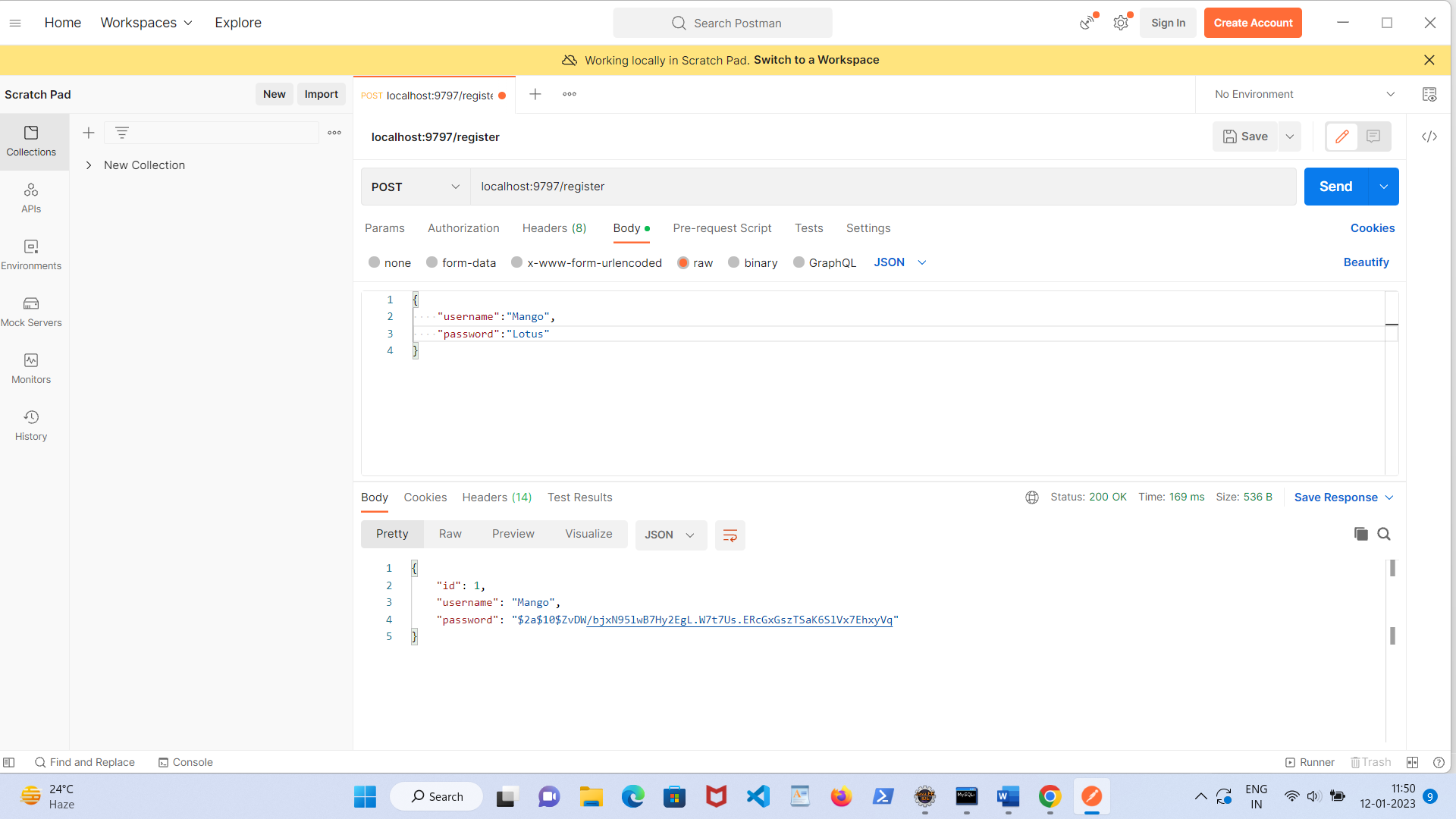
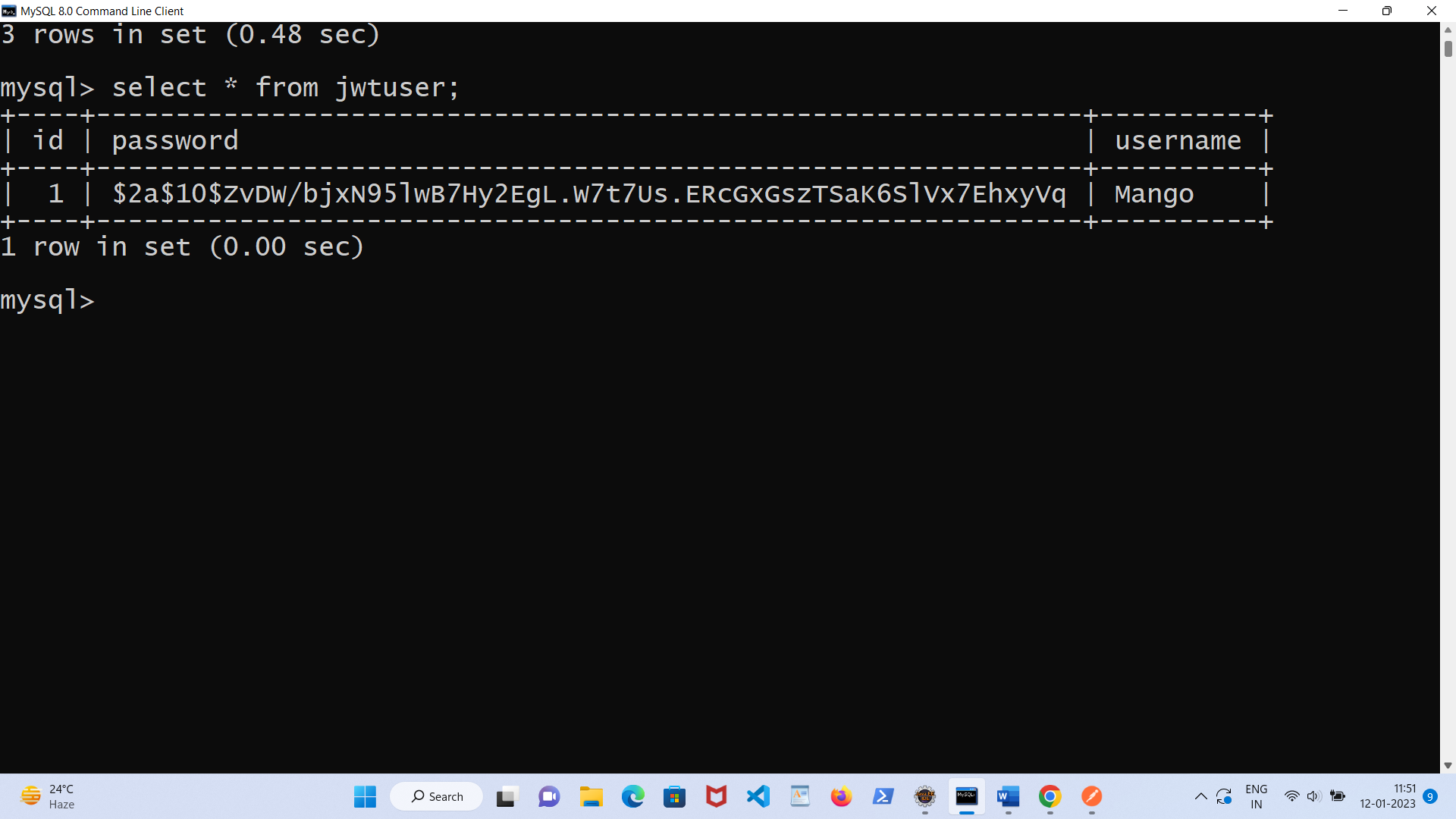
* Run SpringBoot application (port no 9797)
* Open postman
* Create user “Mango” password “Lotus” & type localhost:9797/register in post mode & json format.

{

"username":"Mango",

"password":"Lotus"

}

* 
* Check with database record stored in database.
* 
* In postman to generate token type the following

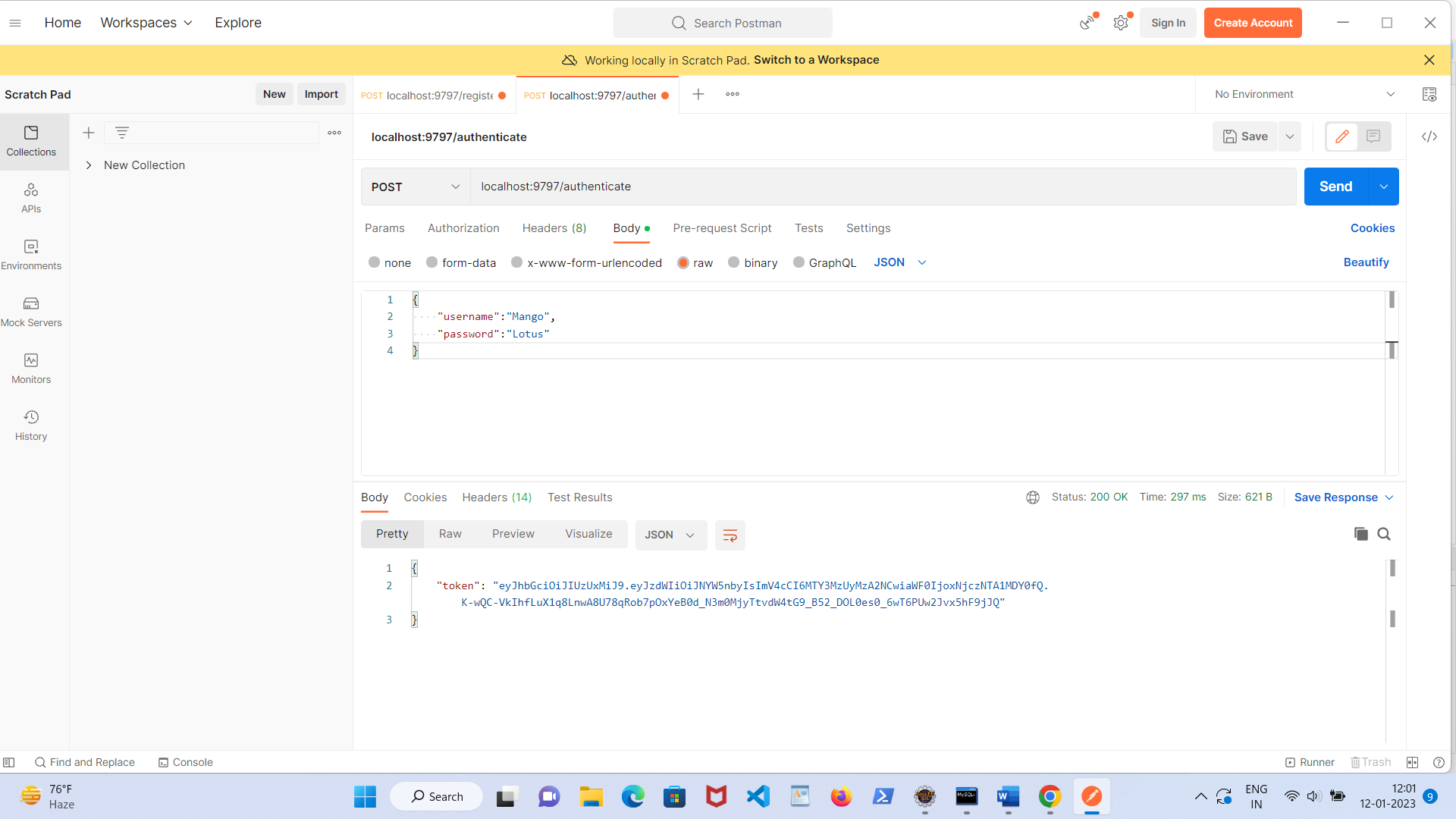
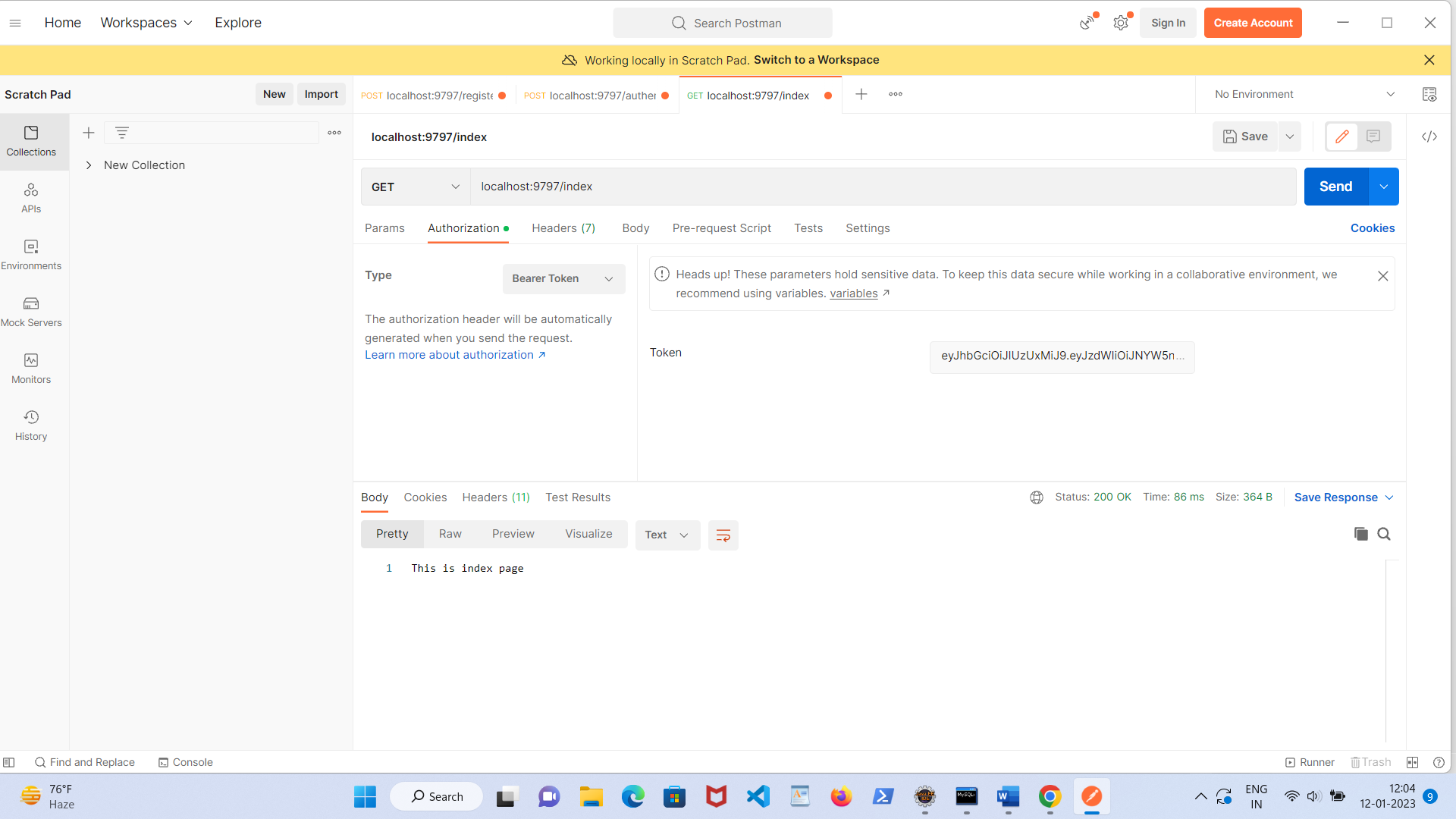
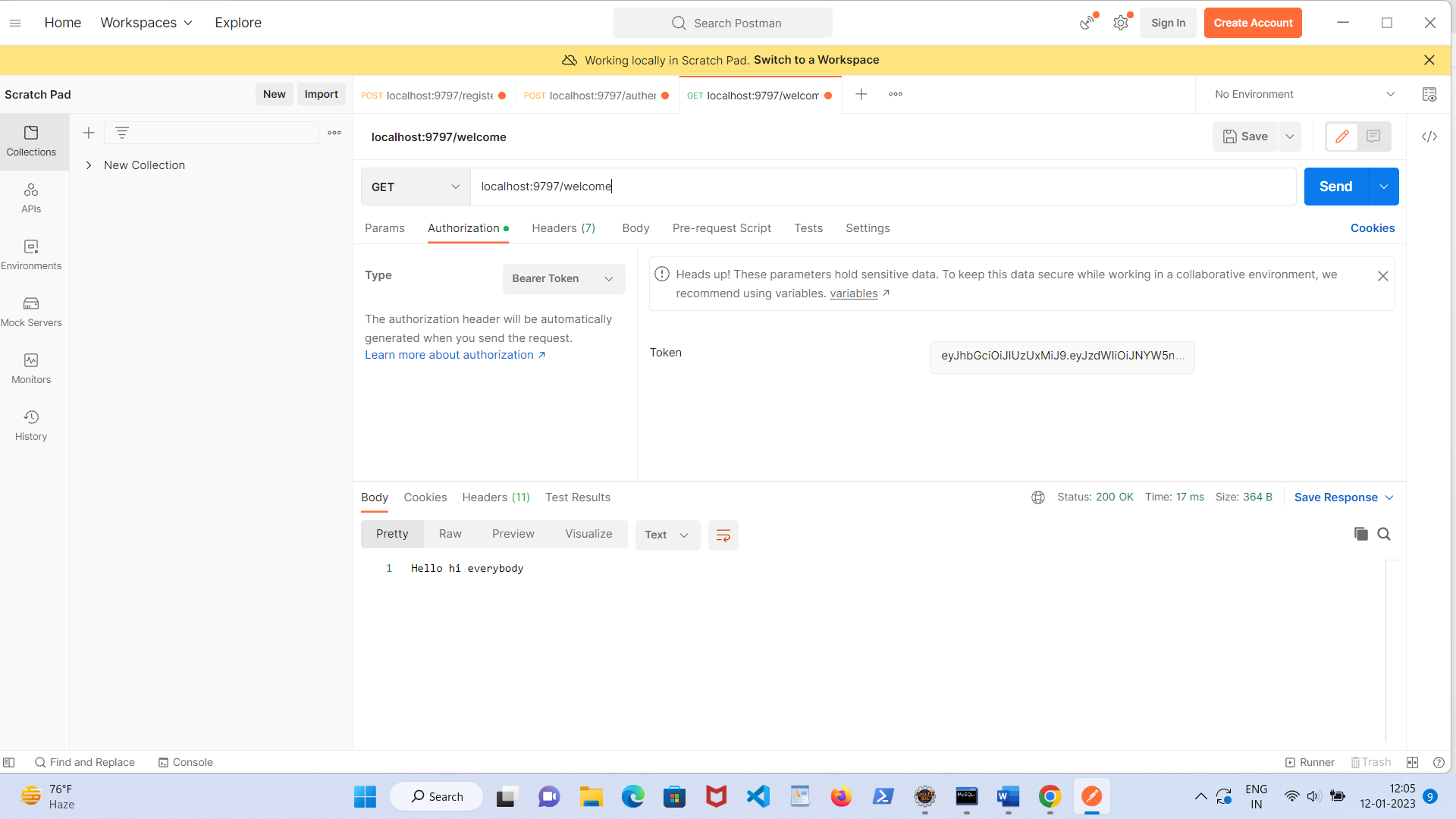
{

"username":"Mango",

"password":"Lotus"

}

Then type the url localhost:9797/authenticate in post mode & json format.

* 
* Token created.
* Now go for the get request for /index, in Authorization for Bearer paste the token, you will see the result of /index.
* 
* One more url /welcome
* 
* If we decode the jwt token online like <http://calebb.net/>

We find the output like below:

{

alg: "HS512"

}.

{

sub: "Mango",

exp: 1673523064,

iat: 1673505064

}.

**Now the Application in details (Jwt with Database):**

1. **application.yml**

server:

port: 9797

jwt:

secret: suramya



spring:

datasource:

driverClassName: com.mysql.cj.jdbc.Driver

url: jdbc:mysql://localhost:3306/jiodb

username: root

password: root

jpa:

hibernate.ddl-auto: update

hibernate.show\_sql: true

mvc:

view:

prefix: /WEB-INF/jsp/

suffix: .jsp

1. **Spring Boot stater class**

**import** org.springframework.boot.SpringApplication;

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

@SpringBootApplication

**public** **class** JwtDatabaseAppApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(JwtDatabaseAppApplication.**class**, args);

}}

1. **User Model class**

**import** com.fasterxml.jackson.annotation.JsonIgnore;

**import** javax.persistence.\*;

@Entity

**public** **class** Jwtuser {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long id;

**private** String username;

**private** String password;

**public** Jwtuser() {

**super**();

}

**public** Jwtuser(String username, String password) {

**super**();

**this**.username = username;

**this**.password = password;

}

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getUsername() {

**return** username;

}

**public** **void** setUsername(String username) {

**this**.username = username;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}}

1. **UserDTO class to avoid direct access to table of Database**

**public** **class** UserDto {

**private** String username;

**private** String password;

**public** UserDto() {

**super**();

}

**public** UserDto(String username, String password) {

**super**();

**this**.username = username;

**this**.password = password;

}

**public** String getUsername() {

**return** username;

}

**public** **void** setUsername(String username) {

**this**.username = username;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}}

1. **JwtRequest class**

**import** java.io.Serializable;

**public** **class** JwtRequest **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 5926468583005150707L;

**private** String username;

**private** String password;

**public** JwtRequest(){

**super**();

}

**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;

}}

1. **JwtResponse class**

**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;

}}

1. **JwtUserRepository class**

**import** org.springframework.data.repository.CrudRepository;

**import** com.mydemo.jwtDatabaseApp.model.Jwtuser;

**public** **interface** JwtuserRepository **extends** CrudRepository<Jwtuser,Long> {

Jwtuser findByUsername(String username);

}

1. **JwtUserDetailService class**

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

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

**import** org.springframework.security.core.userdetails.UserDetailsService;

**import** org.springframework.security.core.userdetails.UsernameNotFoundException;

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

**import** org.springframework.stereotype.Service;

**import** org.springframework.security.core.userdetails.User;

**import** com.mydemo.jwtDatabaseApp.dao.JwtuserRepository;

**import** com.mydemo.jwtDatabaseApp.model.Jwtuser;

**import** com.mydemo.jwtDatabaseApp.model.UserDto;

**import** java.util.ArrayList;

@Service

**public** **class** JwtUserDetailsService **implements** UserDetailsService{

@Autowired

**private** JwtuserRepository userRepository;

@Autowired

**private** PasswordEncoder bcryptEncoder;

@Override

**public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {

Jwtuser user = userRepository.findByUsername(username);

**if** (user == **null**) {

**throw** **new** UsernameNotFoundException("User not found with username: " + username);

}

**return** **new** User(user.getUsername(), user.getPassword(),**new** ArrayList<>());

}

**public** Jwtuser save(UserDto user) {

Jwtuser newUser = **new** Jwtuser(user.getUsername(), bcryptEncoder.encode(user.getPassword()));

newUser.setUsername(user.getUsername());

**return** userRepository.save(newUser);

}}

1. **JwtAuthenticationEntryPoint class**

**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");

}}

1. **JwtTokenUtil class**

**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));

}

}

1. **JwtRequestFilter class**

**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.mydemo.jwtDatabaseApp.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);

}}

1. **EncoderCofig class (required java17)**

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

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

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

@Configuration

**public** **class** EncoderConfig {

@Bean

**public** PasswordEncoder passwordEncoder(){

**return** **new** BCryptPasswordEncoder();

}}

1. **SecurityConfig class**

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.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled = true)

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

@Autowired

private UserDetailsService jwtUserDetailsService;

@Autowired

private JwtRequestFilter jwtRequestFilter;

@Autowired

private EncoderConfig econfig;

@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(econfig.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", "/register").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);

}}

1. **LoginController**

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

**import** com.mydemo.jwtDatabaseApp.config.JwtTokenUtil;

**import** com.mydemo.jwtDatabaseApp.model.JwtRequest;

**import** com.mydemo.jwtDatabaseApp.model.JwtResponse;

**import** com.mydemo.jwtDatabaseApp.model.UserDto;

**import** com.mydemo.jwtDatabaseApp.service.JwtUserDetailsService;

**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.\*;

@RestController

@CrossOrigin

**public** **class** LoginController {

@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));

}

@RequestMapping(value = "/register", method = RequestMethod.***POST***)

**public** ResponseEntity<?> saveUser(@RequestBody UserDto user) **throws** Exception {

System.***out***.println(user.getUsername()+" "+user.getPassword());

**return** ResponseEntity.*ok*(userDetailsService.save(user));

}

**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);

}}}

1. **AppController class**

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

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

@RestController

**public** **class** AppController {

@GetMapping("/welcome")

**public** String sayHello() {

**return** "Hello hi everybody";

}

@GetMapping("/index")

**public** String showIndexPage() {

**return** "This is index page";

}}

1. **index.jsp**

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

<h1>

Hello User, You have successfully logged on

</h1>

</body>

</html>

1. **loginPage.jsp**

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"* isELIgnored=*"false"*%>

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>

<%@ taglib prefix=*"form"* uri=*"http://www.springframework.org/tags/form"* %>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

<c:url value=*"/login"* var=*"login"*/>

<div align=*"center"*>

<h3>

<form:form action=*"*${login}*"*>

Enter User Id:<input type=*"text"* name=*"username"*/>

<br/><br/>

Enter Password:<input type=*"password"* name=*"password"*/>

<br/><br/>

<input type=*"submit"* value=*"Submit"*/>

</form:form>

</h3>

</div>

</body>

</html>

1. **loginError.jsp**

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"* isELIgnored=*"false"*%>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

<div align=*"center"*>

<h1 style="color:*red*">

Wrong credentials.....re-enter...

</h1>

</div>

<jsp:include page=*"loginPage.jsp"* flush=*"true"*/>

</body>

</html>