JWT Token generation and validations: Spring Security:  
==============================================

### What is JWT?

JWT (JSON Web Token) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity-protected with a Message Authentication Code (MAC) and/or encrypted.

**Key Components of JWT:**

1. **Header**: Typically consists of two parts: the type of token (JWT) and the signing algorithm (e.g., HMAC SHA256 or RSA).
2. **Payload**: Contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.
3. **Signature**: To create the signature part, you take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

A JWT looks like this: xxxxx.yyyyy.zzzzz

Let's break down the structure of a JWT (JSON Web Token) and understand its components using an example token.

Structure of a JWT

A JWT consists of three parts:

Header

Payload

Signature

These three parts are encoded separately using Base64Url encoding and concatenated with dots (.) to form a single token. Here's a visual representation:

<Header>.<Payload>.<Signature>

Example Token

Let's take an example JWT:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJqd2R1c2VyIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

1. Header

The header typically consists of two parts:

The type of token, which is JWT.

The signing algorithm being used, such as HMAC SHA256 or RSA.

Example (Base64Url Decoded):

json

{

"alg": "HS256",

"typ": "JWT"

}

1. Payload

The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims:

Registered claims: Predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some examples are iss (issuer), exp (expiration time), sub (subject), and aud (audience).

Public claims: Can be defined at will by those using JWTs. But to avoid collisions, they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision-resistant namespace.

Private claims: Custom claims created to share information between parties that agree to use them.

Example (Base64Url Decoded):

json

{

"sub": "jwtuser",

"name": "John Doe",

"iat": 1516239022

}

1. Signature

To create the signature part, you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example, **if** you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

HMACSHA256(

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

secret)

Example (Decoded Parts)

Using the example token provided, let's decode it:

Header:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

Decoded:

json

{

"alg": "HS256",

"typ": "JWT"

}

Payload:

eyJzdWIiOiJqd3R1c2VyIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ

Decoded:

json

{

"sub": "jwtuser",

"name": "John Doe",

"iat": 1516239022

}

Signature:

SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

This part is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

Summary

A JWT is composed of three parts: the header, payload, and signature. The header and payload are Base64Url encoded, and the signature is used **for** verification. JWTs are commonly used **for** authentication and authorization purposes because they provide a secure way to transmit information between parties.

### Step-by-Step Implementation

add JWT token Dependency:  
========================

<dependencies>

<!-- Spring Boot Dependencies -->

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- Lombok -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.11.5</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

</dependencies>

Let's break down the JWT token-related classes and their importance in a Spring Boot application for authentication and authorization.

### Key JWT Token-Related Classes

1. **JwtUtil (JWT Utility Class)**
2. **JwtAuthenticationFilter (JWT Authentication Filter)**
3. **CustomUserDetailsService (User Details Service)**
4. **SecurityConfig (Security Configuration)**
5. **AuthController (Authentication Controller)**

### 1. JwtUtil (JWT Utility Class)

**Purpose**: This class is responsible for generating, parsing, and validating JWT tokens. It ensures that tokens are created and verified properly, securing the communication between the client and the server.

**Key Methods**:

* generateToken(String username): Generates a JWT token using the user's username.
* getUsernameFromToken(String token): Extracts the username from the token.
* validateToken(String token): Validates the token to ensure it has not expired or been tampered with.

### JWT Utility Class

**JwtUtil.java**

**package** com.k7it.security;

**import** java.util.Date;

**import** java.util.Map;

**import** javax.crypto.SecretKey;

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

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

**import** org.springframework.stereotype.Component;

**import** io.jsonwebtoken.Jwts;

**import** io.jsonwebtoken.SignatureAlgorithm;

**import** io.jsonwebtoken.io.Decoders;

**import** io.jsonwebtoken.security.Keys;

@Component

**public** **class** JwtUtil {

@Value("${jwt.secret}")

**private** String secret;

@Value("${jwt.expiration}")

**private** **long** expiration;

**private** SecretKey getSigningKey() {

**byte**[] keyBytes = Decoders.***BASE64***.decode(secret);

**return** Keys.*hmacShaKeyFor*(keyBytes);

}

**public** String generateToken(String username, Map<String, Object> claims) {

**return** Jwts.*builder*()

.setClaims(claims)

.setSubject(username)

.setIssuedAt(**new** Date())

.setExpiration(**new** Date(System.*currentTimeMillis*() + expiration \* 1000))

.signWith(getSigningKey(), SignatureAlgorithm.***HS512***)

.compact();

}

**public** String getUsernameFromToken(String token) {

**return** Jwts.*parserBuilder*()

.setSigningKey(getSigningKey())

.build()

.parseClaimsJws(token)

.getBody()

.getSubject();

}

**public** **boolean** validateToken(String token, UserDetails userDetails) {

**final** String username = getUsernameFromToken(token);

**return** (username.equals(userDetails.getUsername()) && !isTokenExpired(token));

}

**private** **boolean** isTokenExpired(String token) {

**final** Date expiration = Jwts.*parserBuilder*()

.setSigningKey(getSigningKey())

.build()

.parseClaimsJws(token)

.getBody()

.getExpiration();

**return** expiration.before(**new** Date());

}

}

### JwtAuthenticationFilter (JWT Authentication Filter)

**Purpose**: This filter intercepts each request to validate the JWT token present in the Authorization header. If the token is valid, it sets the authentication in the security context, allowing the request to proceed.

**Key Methods**:

* doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain): Extracts the token, validates it, and sets the authentication context if valid.

### JWT Authentication Filter

**JwtAuthenticationFilter.java**

**package** com.k7it.security;

**import** java.io.IOException;

**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.core.userdetails.UserDetailsService;

**import** org.springframework.security.web.authentication.WebAuthenticationDetailsSource;

**import** org.springframework.stereotype.Component;

**import** org.springframework.web.filter.OncePerRequestFilter;

**import** io.jsonwebtoken.ExpiredJwtException;

**import** io.jsonwebtoken.MalformedJwtException;

**import** io.jsonwebtoken.~~SignatureException~~;

**import** jakarta.servlet.FilterChain;

**import** jakarta.servlet.ServletException;

**import** jakarta.servlet.http.HttpServletRequest;

**import** jakarta.servlet.http.HttpServletResponse;

@Component

**public** **class** JwtAuthenticationFilter **extends** OncePerRequestFilter {

@Autowired

**private** JwtUtil jwtUtil;

@Autowired

**private** UserDetailsService userDetailsService;

@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**;

**if** (requestTokenHeader != **null** && requestTokenHeader.startsWith("Bearer ")) {

jwtToken = requestTokenHeader.substring(7);

**try** {

username = jwtUtil.getUsernameFromToken(jwtToken);

} **catch** (IllegalArgumentException e) {

System.***out***.println("Unable to get JWT Token");

} **catch** (ExpiredJwtException e) {

System.***out***.println("JWT Token has expired");

} **catch** (~~SignatureException~~ | MalformedJwtException e) {

System.***out***.println("Invalid JWT Token");

}

} **else** {

logger.warn("JWT Token does not begin with Bearer String");

}

**if** (username != **null** && SecurityContextHolder.*getContext*().getAuthentication() == **null**) {

UserDetails userDetails = **this**.userDetailsService.loadUserByUsername(username);

**if** (jwtUtil.validateToken(jwtToken, userDetails)) {

**var** authenticationToken = **new** UsernamePasswordAuthenticationToken(

userDetails, **null**, userDetails.getAuthorities());

authenticationToken.setDetails(**new** WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.*getContext*().setAuthentication(authenticationToken);

}

}

chain.doFilter(request, response);

}

}

### CustomUserDetailsService (User Details Service)

**Purpose**: This service loads user-specific data during the authentication process. It fetches user details from the database based on the username.

**Key Methods**:

* loadUserByUsername(String username): Loads user details from the database by username.

### User Details Service

**CustomUserDetailsService.java**

**package** com.k7it.security;

**import** java.util.Set;

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

**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;

**import** com.k7it.repo.UserRepository;

@Service

**public** **class** CustomUserDetailsService **implements** UserDetailsService {

@Autowired

**private** UserRepository userRepository;

@Override

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

com.k7it.model.User user = userRepository.findByUsername(username);

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

**throw** **new** UsernameNotFoundException("User not found");

}

Set<String> roles = user.getRoles();

String[] rolesArray = roles.toArray(**new** String[0]);

**return** User.*builder*().username(user.getUsername()).password(user.getPassword()).roles(rolesArray).build();

}

}

### SecurityConfig (Security Configuration)

**Purpose**: This class configures Spring Security to require JWT authentication for protected endpoints, sets up password encoding, and adds the JWT filter to the security filter chain.

**Key Methods**:

* configure(HttpSecurity http): Configures security settings for HTTP requests, such as which endpoints are public and which require authentication.
* configure(AuthenticationManagerBuilder auth): Configures the authentication manager with user details service and password encoder.
* authenticationManagerBean(): Exposes the authentication manager as a bean.
* passwordEncoder(): Defines the password encoder bean.

**Security Configuration**

**SecurityConfig.java  
=======================================**

**package** com.k7it.security;

**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.configuration.AuthenticationConfiguration;

**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.WebSecurityCustomizer;

**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.SecurityFilterChain;

**import** org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration

@EnableWebSecurity

**public** **class** SecurityConfig {

**private** **final** JwtAuthenticationFilter jwtAuthenticationFilter;

**private** **final** UserDetailsService userDetailsService;

**public** SecurityConfig(JwtAuthenticationFilter jwtAuthenticationFilter, UserDetailsService userDetailsService) {

**this**.jwtAuthenticationFilter = jwtAuthenticationFilter;

**this**.userDetailsService = userDetailsService;

}

@Bean

**public** SecurityFilterChain securityFilterChain(HttpSecurity http) **throws** Exception {

http.~~csrf~~().disable().authorizeHttpRequests(authorize -> authorize

.requestMatchers("/edit-student", "/view-student", "/students", "/create-student", "/", "/login",

"/register", "/auth/login", "/auth/register")

.permitAll() // Permit login and registration

.requestMatchers("/create-student", "/student/upload", "/student/update/\*\*", "/student/delete/\*\*")

.hasRole("ADMIN").requestMatchers("/student/view/\*\*", "/student/all").hasAnyRole("USER", "ADMIN")

.anyRequest().authenticated())

.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.**class**);

**return** http.build();

}

@Bean

**public** AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration)

**throws** Exception {

**return** authenticationConfiguration.getAuthenticationManager();

}

@Bean

**public** PasswordEncoder passwordEncoder() {

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

}

@Bean

**public** WebSecurityCustomizer webSecurityCustomizer() {

**return** (web) -> web.ignoring().requestMatchers("/resources/\*\*");

}

}

### AuthController (Authentication Controller)

**Purpose**: This controller handles user registration and login requests. It generates JWT tokens for authenticated users and registers new users in the system.

**Key Methods**:

* login(Map<String, String> loginData): Authenticates the user and generates a JWT token.
* register(UserDTO userDTO): Registers a new user with the provided details.

### Auth Controller

**AuthController.java**

**package** com.k7it.controller;

**import** com.k7it.dto.UserDTO;

**import** com.k7it.model.User;

**import** com.k7it.repo.UserRepository;

**import** com.k7it.security.JwtUtil;

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

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.security.authentication.AuthenticationManager;

**import** org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

**import** org.springframework.security.core.Authentication;

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

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

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

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

**import** java.util.HashMap;

**import** java.util.HashSet;

**import** java.util.Map;

**import** java.util.Set;

@RestController

@RequestMapping("/auth")

**public** **class** AuthController {

@Autowired

**private** AuthenticationManager authenticationManager;

@Autowired

**private** JwtUtil jwtUtil;

@Autowired

**private** UserRepository userRepository;

@Autowired

**private** PasswordEncoder passwordEncoder;

**private** Set<String> tokenBlacklist = **new** HashSet<>();

@PostMapping("/login")

**public** ResponseEntity<?> login(@RequestBody AuthRequest authRequest) {

Authentication authentication = authenticationManager.authenticate(

**new** UsernamePasswordAuthenticationToken(authRequest.getUsername(), authRequest.getPassword())

);

SecurityContextHolder.*getContext*().setAuthentication(authentication);

UserDetails userDetails = (UserDetails) authentication.getPrincipal();

Map<String, Object> claims = **new** HashMap<>();

claims.put("roles", userDetails.getAuthorities());

String token = jwtUtil.generateToken(userDetails.getUsername(), claims);

**return** ResponseEntity.*ok*(**new** AuthResponse(token));

}

@PostMapping("/register")

**public** ResponseEntity<?> register(@RequestBody UserDTO userDTO) {

**if** (userRepository.findByUsername(userDTO.getUsername()) != **null**) {

**return** ResponseEntity.*status*(400).body("User already exists");

}

User user = **new** User();

user.setUsername(userDTO.getUsername());

user.setPassword(passwordEncoder.encode(userDTO.getPassword()));

user.setRoles(userDTO.getRoles());

userRepository.save(user);

**return** ResponseEntity.*ok*("User registered successfully");

}

@PostMapping("/logout")

**public** ResponseEntity<?> logout(@RequestHeader("Authorization") String token) {

// Extract token from the "Bearer " prefix

token = token.substring(7);

tokenBlacklist.add(token);

**return** ResponseEntity.*ok*("User logged out successfully");

}

**public** **boolean** isTokenBlacklisted(String token) {

**return** tokenBlacklist.contains(token);

}

}

### Summary

**1. JwtUtil**:

* **Purpose**: Generates, parses, and validates JWT tokens.
* **Key Methods**:
  + generateToken(String username): Generates a JWT token using the user's username.
  + getUsernameFromToken(String token): Extracts the username from the token.
  + validateToken(String token): Validates the token to ensure it has not expired or been tampered with.

**2. JwtAuthenticationFilter**:

* **Purpose**: Intercepts and validates JWT tokens in HTTP requests.
* **Key Methods**:
  + doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain): Extracts the token, validates it, and sets the authentication context if valid.

**3. CustomUserDetailsService**:

* **Purpose**: Loads user details from the database for authentication.
* **Key Methods**:
  + loadUserByUsername(String username): Loads user details from the database by username.

**4. SecurityConfig**:

* **Purpose**: Configures Spring Security to require JWT authentication for protected endpoints, sets up password encoding, and adds the JWT filter to the security filter chain.
* **Key Methods**:
  + configure(HttpSecurity http): Configures security settings for HTTP requests, such as which endpoints are public and which require authentication.
  + configure(AuthenticationManagerBuilder auth): Configures the authentication manager with user details service and password encoder.
  + authenticationManagerBean(): Exposes the authentication manager as a bean.
  + passwordEncoder(): Defines the password encoder bean.

**5. AuthController**:

* **Purpose**: Handles user registration and login requests. Generates JWT tokens for authenticated users and registers new users in the system.
* **Key Methods**:
  + login(Map<String, String> loginData): Authenticates the user and generates a JWT token.
  + register(UserDTO userDTO): Registers a new user with the provided details.

Create User.java:  
==============  
To manage roles and permissions we need user details   
**package** com.k7it.model;

**import** org.springframework.data.annotation.Id;

**import** org.springframework.data.mongodb.core.mapping.Document;

**import** lombok.Data;

**import** java.util.Set;

@Data

@Document(collection = "users")

**public** **class** User {

@Id

**private** String id;

**private** String username;

**private** String password;

**private** Set<String> roles; // e.g., ROLE\_USER, ROLE\_ADMIN

}

User Repository:  
=============  
**package** com.k7it.repository;

**import** com.k7it.model.User;

**import** org.springframework.data.mongodb.repository.MongoRepository;

**import** java.util.Optional;

**public** **interface** UserRepository **extends** MongoRepository<User, String> {

Optional<User> findByUsername(String username);

}

UserDTO:  
=============  
**package** com.k7it.service;

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

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

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

**import** com.k7it.dto.UserDTO;

**import** com.k7it.model.User;

**import** com.k7it.repo.UserRepository;

@Service

**public** **class** UserService {

@Autowired

**private** UserRepository userRepository;

@Autowired

**private** PasswordEncoder passwordEncoder;

**public** String registerUser(UserDTO userDTO) {

**if** (userRepository.findByUsername(userDTO.getUsername()) != **null**) {

**return** "User already exists";

}

User user = **new** User();

user.setUsername(userDTO.getUsername());

user.setPassword(passwordEncoder.encode(userDTO.getPassword()));

user.setRoles(userDTO.getRoles());

userRepository.save(user);

**return** "User registered successfully";

}

}

### HTML Login Page

**login.html  
==================**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Login</title>

<style>

.error {

color: red;

}

.navigation {

margin-bottom: 20px;

}

</style>

</head>

<body>

<h1>Login</h1>

<form id="loginForm">

<label for="username">Username:</label>

<input type="text" id="username" name="username" required><br><br>

<label for="password">Password:</label>

<input type="password" id="password" name="password" required><br><br>

<button type="submit">Login</button>

</form>

<script>

document.getElementById('loginForm').addEventListener('submit', **function**(event) {

event.preventDefault(); // Prevent the default form submission

**var** form = event.target;

**var** formData = {

username: form.username.value,

password: form.password.value

};

fetch('/auth/login', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(formData)

})

.then(response **=>** {

if (response.ok) {

return response.json();

} else {

throw new Error('Login failed');

}

})

.then(data **=>** {

if (data.token) {

localStorage.setItem('token', data.token); // Store the token

window.location.href = '/'; // Redirect to home page

} else {

alert('Login failed');

}

})

.catch(error **=>** {

console.error('Error during login:', error);

alert('Login failed: ' + error.message);

});

});

</script>

</body>

</html>

registration.html  
========================================  
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>User Registration</title>

<style>

.error {

color: red;

}

.navigation {

margin-bottom: 20px;

}

</style>

</head>

<body>

<h1>User Registration</h1>

<form id="registrationForm">

<label for="username">Username:</label>

<input type="text" id="username" name="username" required><br><br>

<label for="password">Password:</label>

<input type="password" id="password" name="password" required><br><br>

<label for="roles">Roles:</label>

<select id="roles" name="roles" multiple required>

<option value="USER">User</option>

<option value="ADMIN">Admin</option>

</select><br><br>

<button type="submit">Register</button>

</form>

<script>

document.getElementById('registrationForm').addEventListener('submit', **function**(event) {

event.preventDefault();

**var** form = event.target;

**var** formData = {

username: form.username.value,

password: form.password.value,

roles: Array.from(form.roles.selectedOptions).map(option **=>** option.value)

};

fetch('/auth/register', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(formData)

})

.then(response **=>** response.text())

.then(data **=>** {

alert(data);

if (data === "User registered successfully") {

window.location.href = '/login'; // Redirect to login page

}

})

.catch(error **=>** {

console.error('Error during registration:', error);

alert('Registration failed: ' + error.message);

});

});

</script>

</body>

</html>

Application.properties:  
=======================

spring.application.name=FileUpload

spring.data.mongodb.uri=mongodb://localhost:27017/k7itStudentDB

jwt.secret=RcDQilvvQPF+uQ8CMLfzOl1KSLgiXuIVnb9yzkZ6uK9wkqpwaPCGQboZQV3Ktjl9sHQDQgVmjHyruAiY6mHUqw==

jwt.expiration=3600

use this code to generate security key :  
=================================

**package** com.k7it;

**import** java.security.Key;

**import** io.jsonwebtoken.SignatureAlgorithm;

**import** io.jsonwebtoken.io.Encoders;

**import** io.jsonwebtoken.security.Keys;

**public** **class** KeyGenerator {

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

Key key = Keys.*secretKeyFor*(SignatureAlgorithm.***HS512***); // Generate a key for HS512

String encodedKey = Encoders.***BASE64***.encode(key.getEncoded()); // Encode the key as a Base64 string

System.***out***.println(">>>"+encodedKey+"<<<");

}

}

Output:

YRGzRA8TK60J/DDETgvrjGzdsQL0HZ39Zt/03rwpksKVz09HXZ+xGpd0CfIte/FYQkaN+tscKYs8M2IzbMcLPQ==

**Step-by-Step Guide to Test JWT Tokens in Postman**

\*Perform a login request using your username and password.

\*Retrieve the JWT token from the response

**Testing LogIn Endpoint**

**1.**Open Postman and select the Post method.  
**2.**Enter the login endpoint:  
[**http://localhost:8080/auth/login**](http://localhost:8080/auth/login)

**3.**Navigate to the Body tab:

**i)**Choose the **RAW** option.

**ii)**From the dropdown, select **JSON** format.

**4.**Enter the **JSON** payload in the format:

{

    "username": "k7it",

    "password": "java@143"

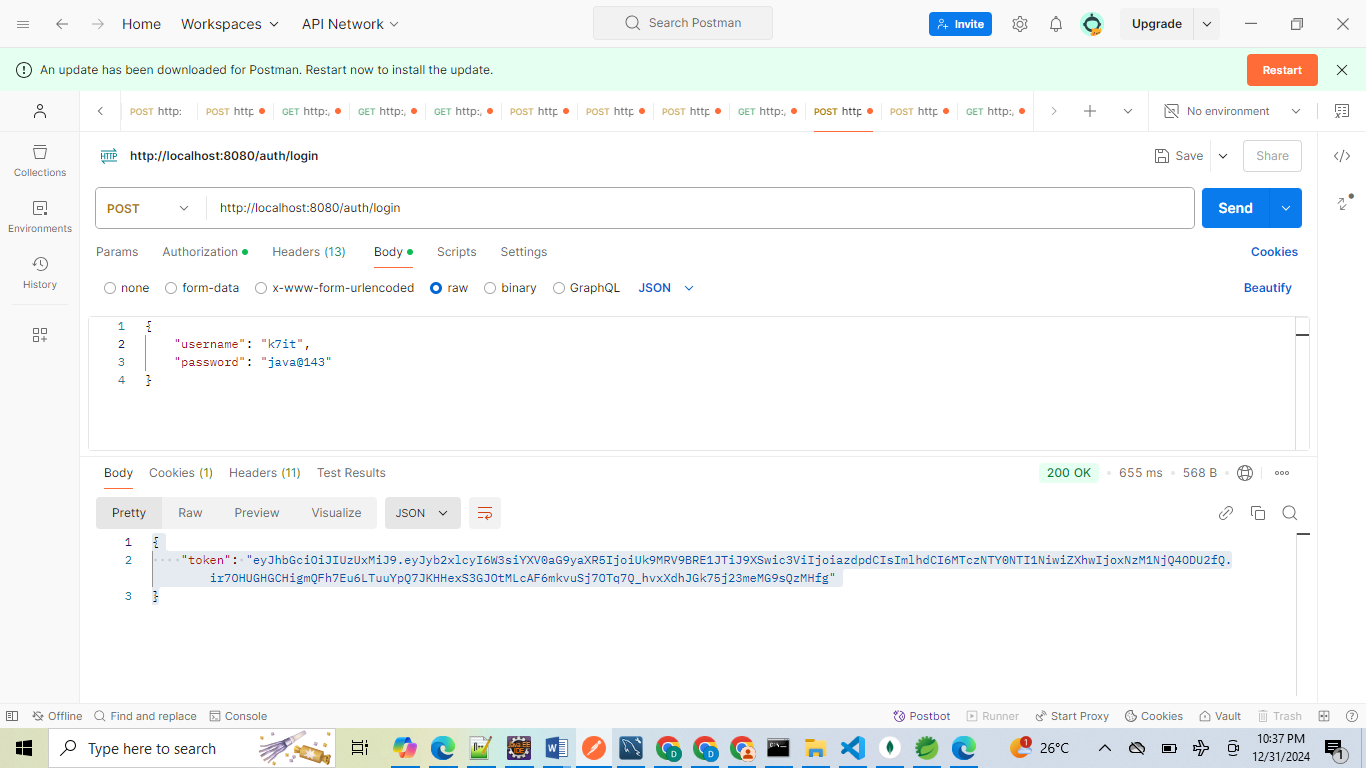
}

**5.**Click **Send** to submit the request..  
**6.** On successful authentication, the API will return a response containing a JWT token ,typically in this format..

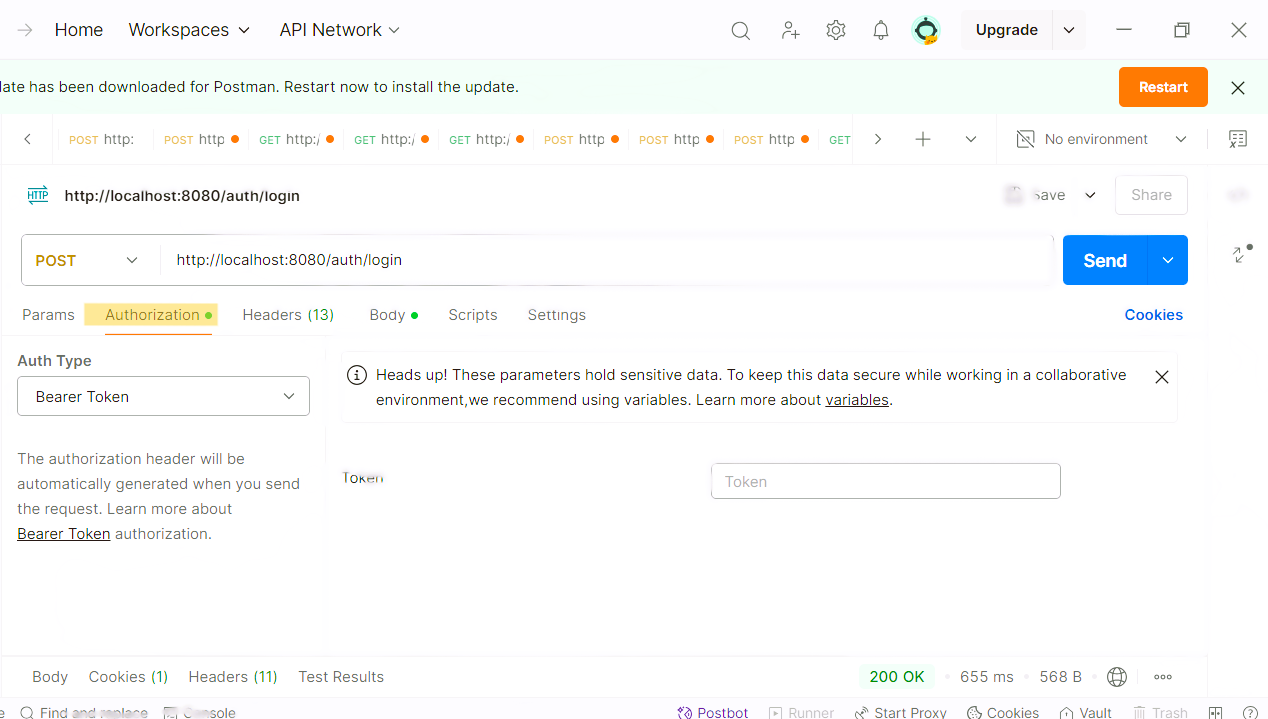
{

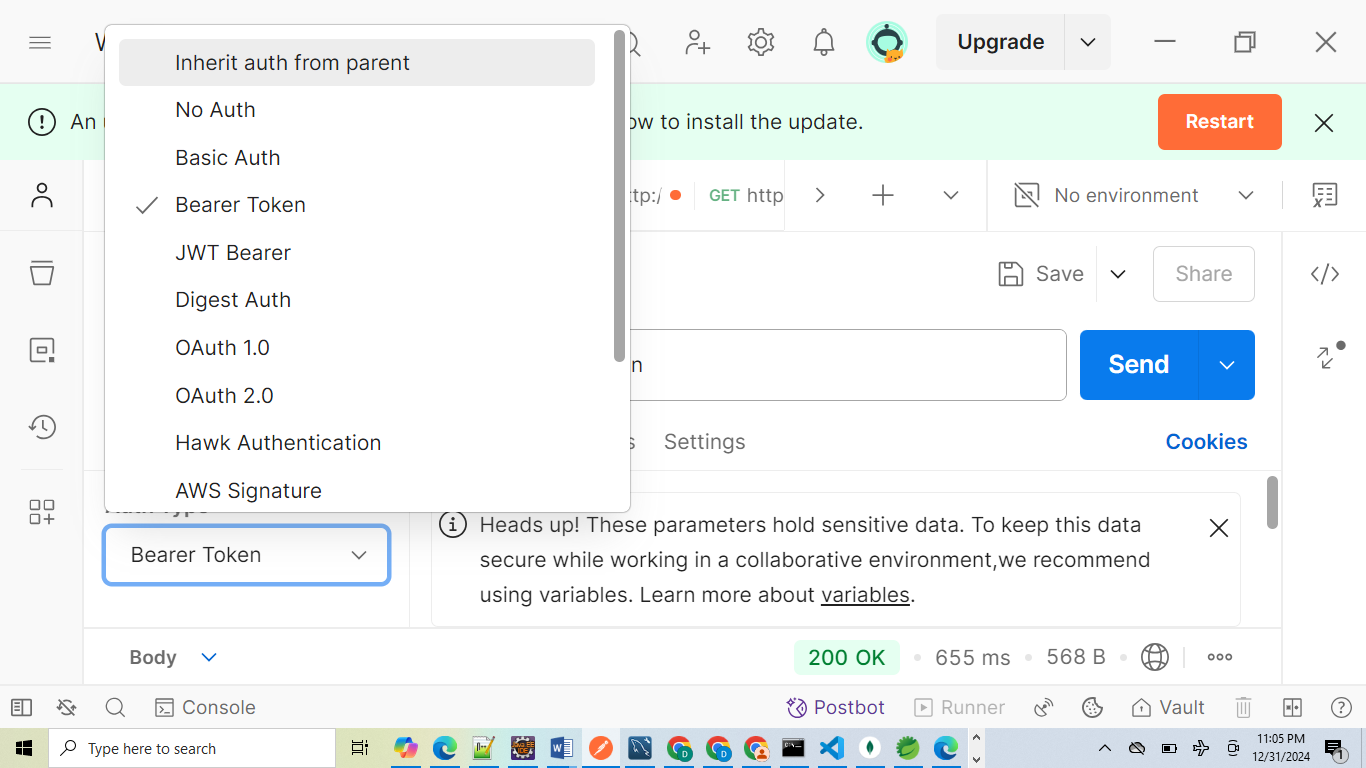
    "token": "eyJhbGciOiJIUzUxMiJ9.eyJyb2xlcyI6W3siYXV0aG9yaXR5IjoiUk9MRV9BRE1JTiJ9XSwic3ViIjoiazdpdCIsImlhdCI6MTczNTY0NTI1NiwiZXhwIjoxNzM1NjQ4ODU2fQ.ir7OHUGHGCHigmQFh7Eu6LTuuYpQ7JKHHexS3GJOtMLcAF6mkvuSj7OTq7Q\_hvxXdhJGk75j23meMG9sQzMHfg"

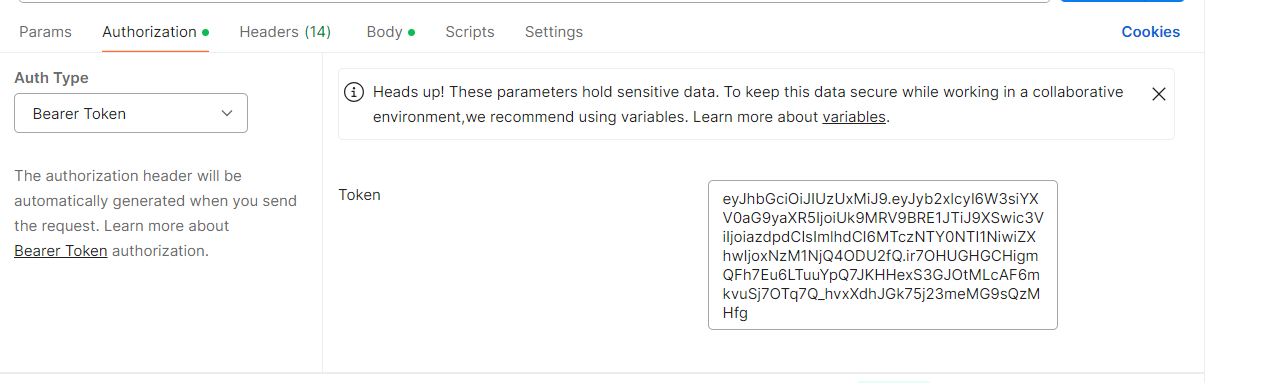
}



**Authorizing Requests with the Token:**

**1.**For authorized action ,navigate to the **Authorization Tab** in postman.  


**2.**Choose **Bearer Token** from the dropdown.  


**3.**Paste the copied token into the Token field.  


**Access other Endpoints:**

**1.**Use the above **Token** for further requests to protected endpoints like:

[**http://localhost:8080/student/view**](http://localhost:8080/student/view) **(GET)**

[**http://localhost:8080/student/update**](http://localhost:8080/student/update) **(PUT)**

[**http://localhost:8080/student/delete**](http://localhost:8080/student/delete) **(DELETE)**

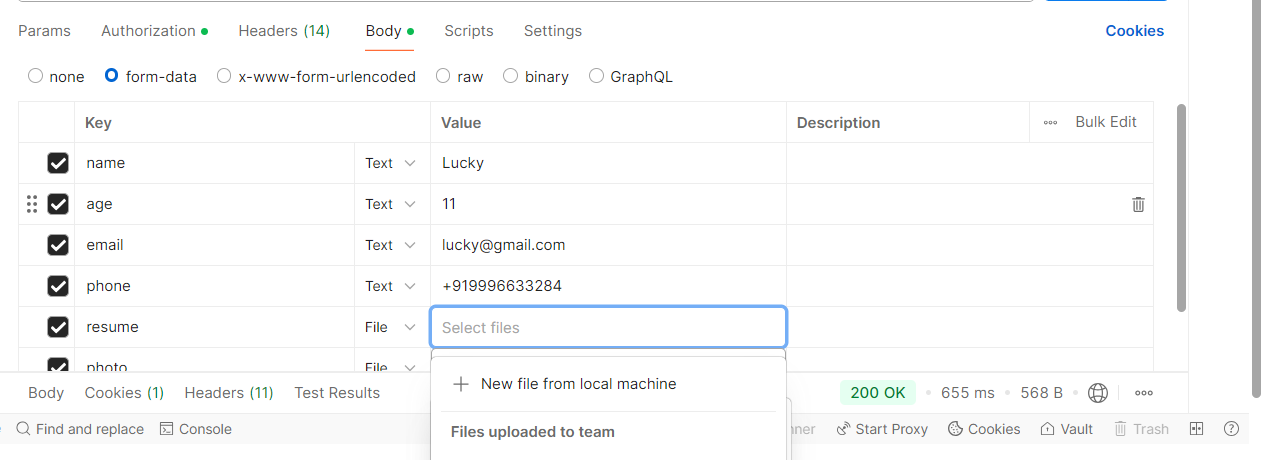
[**http://localhost:8080/student/upload**](http://localhost:8080/student/upload) **(POST)**

**2.**Ensure the correct **HTTP** **method** is selected based on the action.  
  
**Steps to Send Form Data in Postman:**

**1.**Navigate to the **Body Tab** by clicking body tab.  
**2.**Choose the format for sending Form Data for uploading files along with fields..  
**3.**Enter the key and value for each field as required by the API.  
 Example:  
 1)key :username value: your\_username

2)key :password value: your\_password

**4.** If you want to specify the value is text /file .Select the dropdown from the value field and choose File.



**How To Open Development Tool:  
1.**Open browser and click on three dots i.e customize option .  
**2.**click on **more tools** and select **DEVELOPER TOOL** or **click F12**.

