TP 03 : Authentification (JWT) et gestion des Roles –(Partie 2 Spring Boot)

1-Créer le modèle représente par la classe Java **Role** + le constructeur + les **getters** et **setters** :

@Entity public class Role {

@Id @GeneratedValue(strategy = GenerationType.AUTO)

private Long id; private String libelle;

----------}

2- modifier le modèle User1 en remplacant l'attribut role par :

```
@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinTable(
            name = "users_roles",
            joinColumns = @JoinColumn(name = "numUser"),
            inverseJoinColumns = @JoinColumn(name = "id")
            )
    private Set<Role> roles = new HashSet<>();
```

Note:n'oublie pas de modifier les getters, les setters et constructeur

3- Configure Spring Security:

Ouvrir **pom.xml** et ajouter les dependences suivantes dans :

```
<dependencies> ……
<dependency> <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-security</artifactId> </dependency>
<dependency> <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-validation</artifactId> </dependency>
<dependency> <groupId>jakarta.xml.bind</groupId>
     <artifactId> jakarta.xml.bind-api </artifactId> </dependency>
<dependency> <groupId>io.jsonwebtoken</groupId>
     <artifactId>jjwt</artifactId>        <version>0.9.1</version></dependency>
<dependency><groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId><version>2.2.7</version> </dependency>
    <dependency> <groupId>com.sun.xml.bind</groupId>
        <artifactId>jaxb-impl</artifactId>
        <version>2.2.5-b10</version>
    </dependency>
……. </dependencies>
```

4-Dans le package *sécurité*, on crée la class: ***WebSecurityConfig.java***

```
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.authentication.dao.DaoAuthenticationProvider;
import
org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfiguration;
import org.springframework.security.config.http.SessionCreationPolicy;
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;
import com.security.jwt.AuthEntryPointJwt;
import com.security.jwt.AuthTokenFilter;
import com.security.services.UserDetailsServiceImpl;

@Configuration
@EnableMethodSecurity(// securedEnabled = true,  // jsr250Enabled = true,
                prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfiguration {
        @Autowired
        UserDetailsServiceImpl userDetailsService;

        @Autowired
        private AuthEntryPointJwt unauthorizedHandler;

        @Bean
        public AuthTokenFilter authenticationJwtTokenFilter() {
                return new AuthTokenFilter();}

        @Bean
  public DaoAuthenticationProvider authenticationProvider() {
      DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();

      authProvider.setUserDetailsService(userDetailsService);
      authProvider.setPasswordEncoder(passwordEncoder());

      return authProvider; }


   @Bean
   public AuthenticationManager authenticationManager(AuthenticationConfiguration authConfiguration)
throws Exception {
    return authConfiguration.getAuthenticationManager();
  }


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

  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.cors().and().csrf().disable()
        .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
        .authorizeRequests().requestMatchers("/api/auth/**").permitAll()
        .requestMatchers("/api/test/**").permitAll()
        .anyRequest().authenticated();

    http.authenticationProvider(authenticationProvider());

    http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);
    return http.build(); }}
```

5-Implémenter UserDetails & UserDetailsService

Si le processus d'authentification réussit, nous pouvons obtenir les **informations** de l'utilisateur telles que le **nom d'utilisateur**, le mot de passe, les autorités à partir d'un objet d'authentification.

UserDetailsImpl.java

```
import java.util.Collection;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import com.example.backapp.User1;
import com.fasterxml.jackson.annotation.JsonIgnore;

public class UserDetailsImpl implements UserDetails {
	private static final long serialVersionUID = 1L;

	private Long id;

	private String username;

	//private String email;

	@JsonIgnore
	private String password;

	private Collection<? extends GrantedAuthority> authorities;

	public UserDetailsImpl(Long id, String username, /*String email,*/ String password,
			Collection<? extends GrantedAuthority> authorities) {
		this.id = id;
		this.username = username;
		//this.email = email;
		this.password = password;
		this.authorities = authorities;
	}

	public static UserDetailsImpl build(User1 user) {
		List<GrantedAuthority> authorities = user.getRoles().stream()
				.map(role -> new SimpleGrantedAuthority(role.getLibelle()))
				.collect(Collectors.toList());

		return new UserDetailsImpl(
				user.getNumUser(),
				user.getUserName(),
			/*	user.getEmail(),*/
				user.getMotPasse(),
				authorities);
	}

	@Override
	public Collection<? extends GrantedAuthority> getAuthorities() {
		return authorities;
	}

	public Long getId() {
		return id;
	}

	/*public String getEmail() {
		return email;
```

```
        }*/

        @Override
        public String getPassword() {
                return password;
        }

        @Override
        public String getUsername() {
                return username;
        }

        @Override
        public boolean isAccountNonExpired() {
                return true;
        }

        @Override
        public boolean isAccountNonLocked() {
                return true;
        }

        @Override
        public boolean isCredentialsNonExpired() {
                return true;
        }

        @Override
        public boolean isEnabled() {
                return true;
        }

        @Override
        public boolean equals(Object o) {
                if (this == o)
                        return true;
                if (o == null || getClass() != o.getClass())
                        return false;
                UserDetailsImpl user = (UserDetailsImpl) o;
                return Objects.equals(id, user.id);
        }
}
```

UserDetailsServiceImpl.java

```
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.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.example.backapp.User1;
import com.example.backapp.UserRepository;


@Service
public class UserDetailsServiceImpl implements UserDetailsService {

  @Autowired
  UserRepository userRepository;

  @Override
  @Transactional
  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    User1 user = userRepository.findByUserName(username)
        .orElseThrow(() -> new UsernameNotFoundException("User Not Found with username: " + username));

    return UserDetailsImpl.build(user);
  }}
```

6- Filtrer les demandes

Définissons un **filtre** qui s'exécute une fois par requête. Nous créons donc la classe **AuthTokenFilter** qui étend **OncePerRequestFilter** et remplace la méthode **doFilterInternal().**

AuthTokenFilter.java

```
import java.io.IOException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
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.util.StringUtils;
import org.springframework.web.filter.OncePerRequestFilter;
import com.security.services.UserDetailsServiceImpl;
import jakarta.servlet.*;

public class AuthTokenFilter extends OncePerRequestFilter {
  @Autowired
  private JwtUtils jwtUtils;

  @Autowired
  private UserDetailsServiceImpl userDetailsService;

  private static final Logger logger = LoggerFactory.getLogger(AuthTokenFilter.class);

  @Override
  protected void doFilterInternal(jakarta.servlet.http.HttpServletRequest request,
jakarta.servlet.http.HttpServletResponse response, FilterChain filterChain)
      throws ServletException, IOException {
    try {
      String jwt = parseJwt(request);
      if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
        String username = jwtUtils.getUserNameFromJwtToken(jwt);

        UserDetails userDetails = userDetailsService.loadUserByUsername(username);
        UsernamePasswordAuthenticationToken authentication = new
UsernamePasswordAuthenticationToken(userDetails, null,
            userDetails.getAuthorities());
        authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

        SecurityContextHolder.getContext().setAuthentication(authentication);
      }
    } catch (Exception e) {
      logger.error("Cannot set user authentication: {}", e);
    }

    filterChain.doFilter(request, response);
  }

  private String parseJwt(jakarta.servlet.http.HttpServletRequest request) {
    String headerAuth = request.getHeader("Authorization");

    if (StringUtils.hasText(headerAuth) && headerAuth.startsWith("Bearer ")) {
      return headerAuth.substring(7, headerAuth.length());
    }

    return null;
  }
}
```

7- Créer une classe d'utilitaire JWT

Cette classe a 3 fonctions :

1. Générer un JWT à partir du nom d'utilisateur, de la date, de l'expiration et du secret
2. Obtenir le nom d'utilisateur de JWT
3. Valider un JWT

JwtUtils.java

```
import java.util.Date;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.Authentication;
import org.springframework.stereotype.Component;
import com.security.services.UserDetailsImpl;
import io.jsonwebtoken.*;

@Component
public class JwtUtils {
  private static final Logger logger = LoggerFactory.getLogger(JwtUtils.class);

  @Value("${bezkoder.app.jwtSecret}")
  private String jwtSecret;

  @Value("${bezkoder.app.jwtExpirationMs}")
  private int jwtExpirationMs;

  public String generateJwtToken(Authentication authentication) {

    UserDetailsImpl userPrincipal = (UserDetailsImpl) authentication.getPrincipal();

    return Jwts.builder().setSubject((userPrincipal.getUsername())).setIssuedAt(new Date())
        .setExpiration(new Date((new Date()).getTime() +
jwtExpirationMs)).signWith(SignatureAlgorithm.HS512, jwtSecret)
        .compact();
  }

  public String getUserNameFromJwtToken(String token) {
    return Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(token).getBody().getSubject();
  }

  public boolean validateJwtToken(String authToken) {
    try {
      Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
      return true;
    } catch (SignatureException e) {
      logger.error("Invalid JWT signature: {}", e.getMessage());
    } catch (MalformedJwtException e) {
      logger.error("Invalid JWT token: {}", e.getMessage());
    } catch (ExpiredJwtException e) {
      logger.error("JWT token is expired: {}", e.getMessage());
    } catch (UnsupportedJwtException e) {
      logger.error("JWT token is unsupported: {}", e.getMessage());
    } catch (IllegalArgumentException e) {
      logger.error("JWT claims string is empty: {}", e.getMessage());
    }

    return false;
  }}
```

8-Gérer l'exception d'authentification

Nous créons maintenant la classe AuthEntryPointJwt qui implémente l'interface AuthenticationEntryPoint. Ensuite, nous redéfinissons la méthode commence(). Cette méthode sera déclenchée chaque fois qu'un utilisateur non authentifié demande une ressource HTTP sécurisée et qu'une AuthenticationException est levée.

AuthEntryPointJwt.java

```
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.MediaType;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

import com.fasterxml.jackson.databind.ObjectMapper;

@Component
public class AuthEntryPointJwt implements AuthenticationEntryPoint {

  private static final Logger logger = LoggerFactory.getLogger(AuthEntryPointJwt.class);

  @Override
  public void commence(jakarta.servlet.http.HttpServletRequest request,
jakarta.servlet.http.HttpServletResponse response, AuthenticationException authException)
      throws IOException, jakarta.servlet.ServletException {
    logger.error("Unauthorized error: {}", authException.getMessage());
    response.sendError(jakarta.servlet.http.HttpServletResponse.SC_UNAUTHORIZED, "Error: Unauthorized");
  }}
```

9-Définir les charges utiles pour Spring RestController

Permettez-moi de résumer les charges utiles de nos RestAPI :

– Demandes : LoginRequest : { nom d'utilisateur, mot de passe }

```
public class LoginRequest {
        @jakarta.validation.constraints.NotBlank
        private String username;

        @jakarta.validation.constraints.NotBlank
        private String 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;
        }
```

– Réponses :

JwtResponse : { jeton, type, identifiant, nom d'utilisateur, e-mail, rôles }

JwtResponse.java

```
import java.util.List;

public class JwtResponse {
	private String token;
	private String type = "Bearer";
	private Long id;
	private String username;

	private List<String> roles;

	public JwtResponse(String accessToken, Long id, String username, List<String> roles) {
		this.token = accessToken;
		this.id = id;
		this.username = username;
		this.roles = roles;
	}

	public String getAccessToken() {
		return token;
	}

	public void setAccessToken(String accessToken) {
		this.token = accessToken;
	}

	public String getTokenType() {
		return type;
	}

	public void setTokenType(String tokenType) {
		this.type = tokenType;
	}

	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 List<String> getRoles() {
		return roles;
	}
}
```

MessageRéponse : { message }

MessageRéponse.java

```
public class MessageResponse {
        private String message;

        public MessageResponse(String message) {
            this.message = message;
          }

        public String getMessage() {
                return message;
        }

        public void setMessage(String message) {
                this.message = message;
        }
}
```

10-Créer des contrôleurs Spring RestAPIs

10.1- Contrôleur d'authentification

Ce contrôleur fournit des API pour les actions d'enregistrement et de connexion.

– /api/auth/inscription

Vérifier le nom d'utilisateur / e-mail existant

Créer un nouvel utilisateur (avec ROLE_USER si le rôle n'est pas spécifié)

Enregistrer l'utilisateur dans la base de données à l'aide de UserRepository

– /api/auth/signin

authentifier { nom d'utilisateur, mot de passe }

mettre à jour SecurityContext à l'aide de l'objet Authentication

Générer JWT

Obtenir les détails de l'utilisateur à partir de l'objet d'authentification

la réponse contient des données JWT et UserDetails

AuthController.java

```
import java.util.Date;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
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.crypto.password.PasswordEncoder;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import payload.request.LoginRequest;
import payload.request.SignupRequest;
import payload.response.JwtResponse;
import payload.response.MessageResponse;
import com.security.jwt.JwtUtils;
import com.security.services.UserDetailsImpl;

@CrossOrigin(origins ="http://localhost:4200")
@RestController
@RequestMapping("/api/auth")
public class AuthController {


        @Autowired(required=false)
        AuthenticationManager authenticationManager;

        @Autowired
        UserRepository userRepository;

        @Autowired
        RoleRepository roleRepository;

        @Autowired(required=false)
        PasswordEncoder encoder;

        @Autowired(required=false)
        JwtUtils jwtUtils;
//@jakarta.validation.Valid
        @PostMapping("/signin")
        public ResponseEntity<?> authenticateUser(@RequestBody LoginRequest loginRequest) {

                Authentication authentication = authenticationManager.authenticate(
                                new UsernamePasswordAuthenticationToken(loginRequest.getUsername(),
loginRequest.getPassword()));

                SecurityContextHolder.getContext().setAuthentication(authentication);
                String jwt = jwtUtils.generateJwtToken(authentication);

                UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();
                List<String> roles = userDetails.getAuthorities().stream()
                                .map(item -> item.getAuthority())
                                .collect(Collectors.toList());

                return ResponseEntity.ok(new JwtResponse(jwt,

userDetails.getId(),

userDetails.getUsername(),
                                                                                        roles));
        }
```

```
	@PostMapping("/signup")
	public ResponseEntity<?> registerUser(@jakarta.validation.Valid @RequestBody SignupRequest
signUpRequest) {
		if (userRepository.existsByUserName(signUpRequest.getUsername())) {
			return ResponseEntity
					.badRequest()
					.body(new MessageResponse("Error: Username is already taken!"));
		}

		/*if (userRepository.existsByEmail(signUpRequest.getEmail())) {
			return ResponseEntity
					.badRequest()
					.body(new MessageResponse("Error: Email is already in use!"));
		}*/

		// Create new user's account
		User1 user = new User1( signUpRequest.getUsername(),"","", new Date(),"",
				encoder.encode(signUpRequest.getPassword()), null, 0);

		Set<String> strRoles = signUpRequest.getRole();
		Set<Role> roles = new HashSet<>();

		if (strRoles == null) {
			Role userRole = roleRepository.findByLibelle(ERole.ROLE_USER.toString())
					.orElseThrow(() -> new RuntimeException("Error: Role is not
found."));
			roles.add(userRole);
		} else {
			strRoles.forEach(role -> {
				switch (role) {
				case "admin":
					Role adminRole =
roleRepository.findByLibelle(ERole.ROLE_ADMIN.toString())
							.orElseThrow(() -> new RuntimeException("Error:
Role is not found."));
					roles.add(adminRole);

					break;
				case "mod":
					Role modRole =
roleRepository.findByLibelle(ERole.ROLE_MODERATOR.toString())
							.orElseThrow(() -> new RuntimeException("Error:
Role is not found."));
					roles.add(modRole);

					break;
				default:
					Role userRole =
roleRepository.findByLibelle(ERole.ROLE_USER.toString())
							.orElseThrow(() -> new RuntimeException("Error:
Role is not found."));
					roles.add(userRole);
				}
			});
		}

		user.setRoles(roles);
		userRepository.save(user);

		return ResponseEntity.ok(new MessageResponse("User registered successfully!"));
	}
}
```

10.2-Contrôleur pour tester l'autorisation

Il existe 4 API :

- /api/test/all pour un accès public

- /api/test/user pour les utilisateurs a ROLE_USER ou ROLE_MODERATOR ou ROLE_ADMIN

- /api/test/mod pour les utilisateurs a ROLE_MODERATOR

- /api/test/admin pour les utilisateurs a ROLE_ADMIN

TestController.java

```
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/test")
public class TestController {
	@GetMapping("/all")
	public String allAccess() {
		return "Public Content.";
	}

	@GetMapping("/user")
	@PreAuthorize("hasRole('USER') or hasRole('MODERATOR') or hasRole('ADMIN')")
	public String userAccess() {
		return "User Content.";
	}

	@GetMapping("/mod")
	@PreAuthorize("hasRole('MODERATOR')")
	public String moderatorAccess() {
		return "Moderator Board.";
	}

	@GetMapping("/admin")
	@PreAuthorize("hasRole('ADMIN')")
	public String adminAccess() {
		return "Admin Board.";
	}
}
```

11. Modifier le fichier ***application.properties*** en ajoutant les lignes suivantes :

bezkoder.app.jwtSecret= bezKoderSecretKey

bezkoder.app.jwtExpirationMs= 86400000

12. Inserer les roles dans la table Role en utilisant **PgAdmin**:

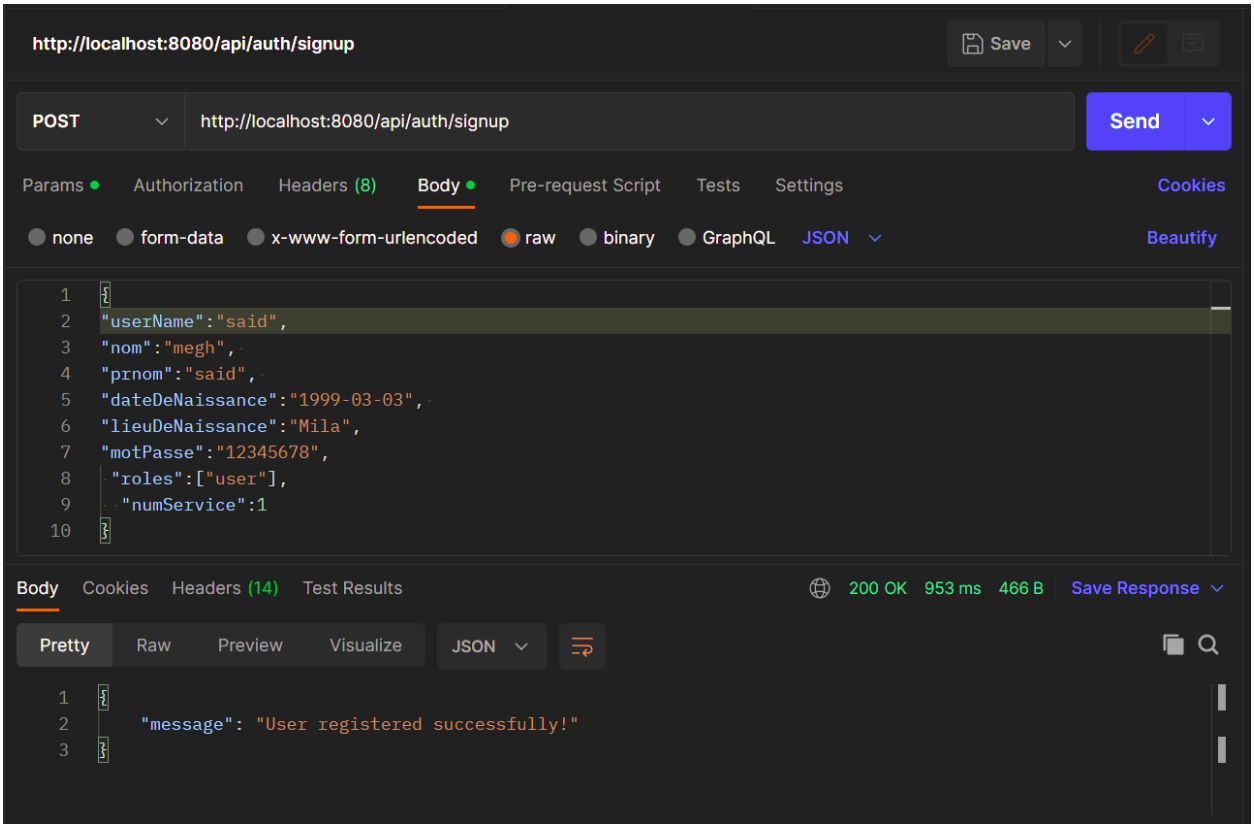
```
NSERT INTO role(libelle) VALUES('ROLE_USER');
INSERT INTO role(libelle) VALUES('ROLE_MODERATOR');
INSERT INTO role(libelle) VALUES('ROLE_ADMIN');
```

13.1 Insérer un utilisateur dans la table user1 avec PgAdmin :

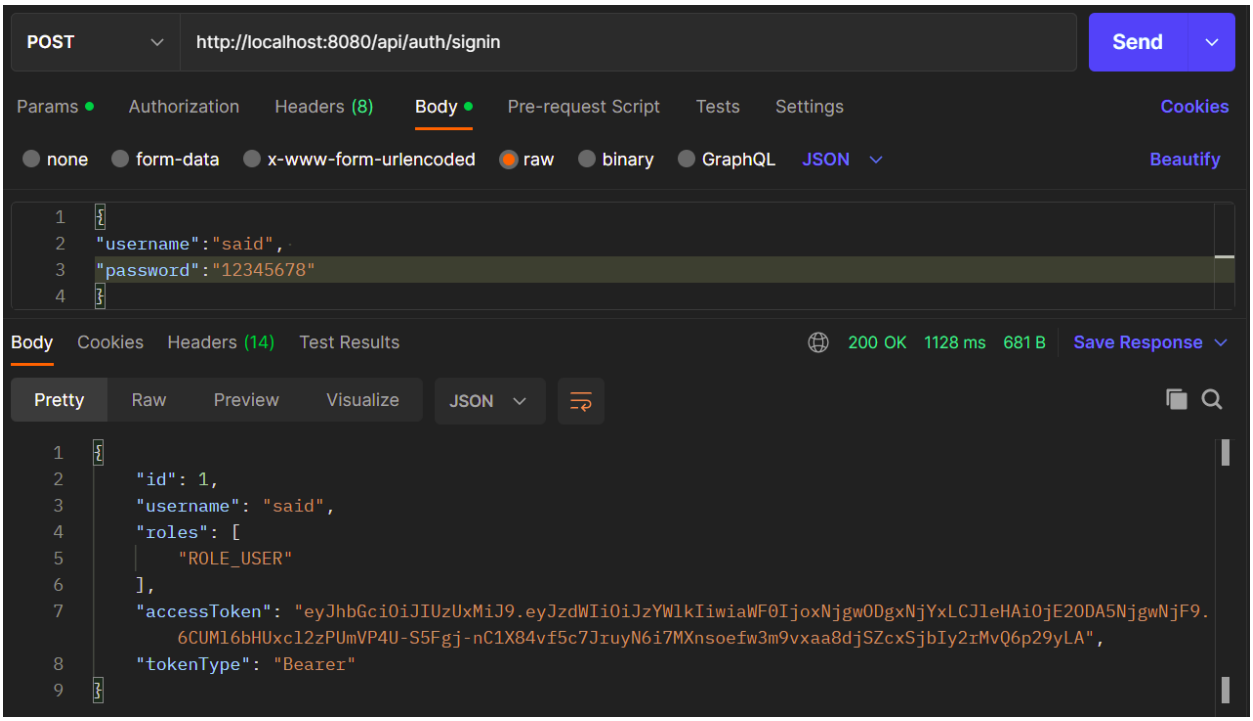
```
INSERT INTO user1 (date_de_naissance, mot_passe, num_service, user_name)VALUES
('2023-04-07 16:18:36.229','$2a$10$XdoGU5nATZxPxLitk8SeuetxDnpvoZf9kFEkEmCwXAzzJbNnSLzLe', 1, 'samir');

INSERT INTO users_roles (num_user,id) VALUES(1,1);
```

13.2 Ajouter un utilisateur avec **postman** ***(optionelle)*** :

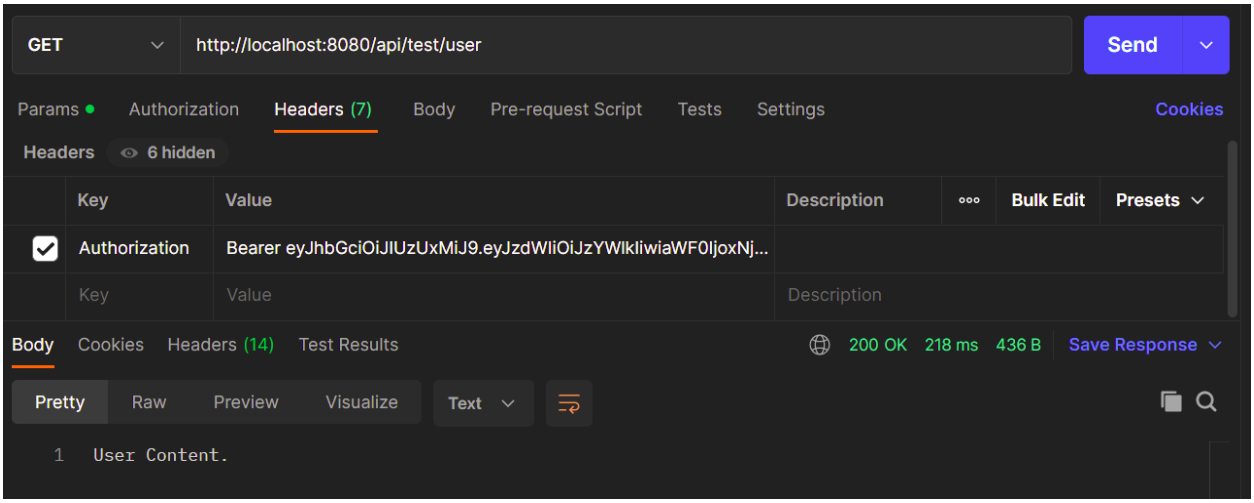
```
{
"userName":"said",
"nom":"megh",
"prnom":"said",
"dateDeNaissance":"1999-03-03",
"lieuDeNaissance":"Mila",
"motPasse":"12345678",
 "roles":["user"],
  "numService":1
}
```



14- s’authentifier avec Postman :



15- Accéder à la page de l’utilisateur avec Postman :



Références :

https://www.bezkoder.com/angular-15-jwt-auth/

https://www.bezkoder.com/spring-boot-security-postgresql-jwt-authentication/

https://www.bezkoder.com/angular-15-spring-boot-jwt-auth/