**Bài báo cáo nghiên cứu Spring Boot 3 + Spring Security 6 - JWT Authentication and Authorization**

Họ và tên: Trần Kim Hoàng

Phần trình bày:

1. Các khái niệm:
2. Spring Security

Spring Security là một trong những core feature quan trọng của Spring Framework, nó giúp chúng ta phân quyền và xác thực người dùng trước khi cho phép họ truy cập vào các tài nguyên của chúng ta.

1. Authentication

Authentication là về việc xác thực thông tin đăng nhập của bạn như Tên người dùng / ID người dùng và mật khẩu để xác minh danh tính của bạn. Trong các public và private network, hệ thống xác thực danh tính người dùng thông qua mật khẩu đăng nhập. Authentication thường được thực hiện bởi tên người dùng và mật khẩu, và đôi khi kết hợp với các yếu tố xác thực, trong đó đề cập đến các cách khác nhau để được xác thực.

1. Authorization

Authorization là quá trình để xác định xem người dùng được xác thực có quyền truy cập vào các tài nguyên cụ thể hay không. Nó xác minh quyền của bạn để cấp cho bạn quyền truy cập vào các tài nguyên như thông tin, cơ sở dữ liệu, file, v.v. Authorization thường được đưa ra sau khi xác thực xác nhận các đặc quyền của bạn để thực hiện. Nói một cách đơn giản hơn, nó giống như cho phép ai đó chính thức làm điều gì đó hoặc bất cứ điều gì.

1. JWT (JSON Web Token)

Token-based authentication là phương thức xác thực bằng chuỗi má hóa. Một hệ thống sử dụng Token-based authentication cho phép người dùng nhập user/password (hoặc tương tự) để nhận về 1 chuỗi mã token. Mã này được sử dụng để "xác minh" quyền truy cập vào tài nguyên mà không cần phải cung cấp lại username/password nữa.

JSON Web Token (JWT) là 1 tiêu chuẩn mở (RFC 7519) định nghĩa cách thức truyền tin an toàn giữa các thành viên bằng 1 đối tượng JSON. Thông tin này có thể được xác thực và đánh dấu tin cậy nhờ vào "chữ ký" của nó. Phần chữ ký của JWT sẽ được mã hóa lại bằng HMAC hoặc RSA.

Như vậy, Bảo mật JWT là phuơng pháp xác thực quyền truy cập (Authentication) bằng JSON Web Token.

Dưới đây là 1 JSON Web Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJzdWIiOiJuaHMzMTA4IiwiZXhwIjoxNTU4MDYzODM3fQ.

449KVmOFWcpOUjnYGm-f1QWhY8N-DerKDfTK0JQm1Nc

* JWT trên bao gồm 3 phần:

+ Header (eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9)

+ Payload (eyJzdWIiOiJuaHMzMTA4IiwiZXhwIjoxNTU4MDYzODM3fQ)

+ Signature (449KVmOFWcpOUjnYGm-f1QWhY8N-DerKDfTK0JQm1Nc)

Được phân cách nhau bởi dấu "." (chấm).

* 1. Header

Header: Là phần đầu tiên trong chuỗi JWT và chứa hai thuộc tính: loại token (typ) và thuật toán mã hóa (alg). Thuật toán mã hóa được sử dụng để mã hóa payload của token.

* 1. Payload

Payload: Là phần chứa thông tin cần truyền tải trong chuỗi JWT. Payload có thể chứa các cặp key-value, được gọi là "claims". Có ba loại claim: registered, public và private.

* 1. Signature

Signature: Là phần cuối cùng của chuỗi JWT và được tạo ra bằng cách sử dụng thuật toán mã hóa đã được chỉ định trong header. Signature được tạo bằng cách mã hóa header và payload cùng với một secret key bí mật đã được thỏa thuận trước đó giữa các bên tham gia trao đổi thông tin.

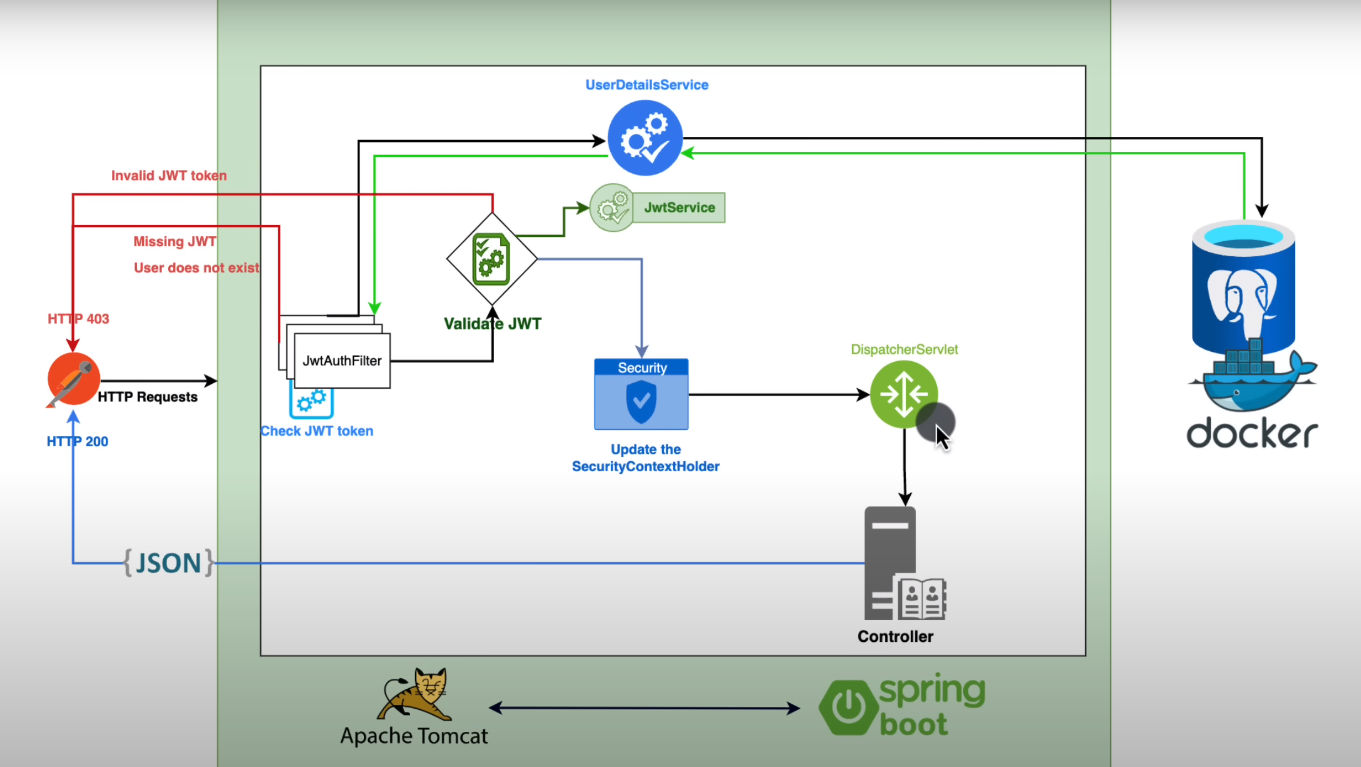
1. Stateless

Stateless là thiết kế không lưu dữ liệu của client trên server. Có nghĩa là sau khi client gửi dữ liệu lên server, server thực thi xong, trả kết quả thì “quan hệ” giữa client và server bị “cắt đứt” – server không lưu bất cứ dữ liệu gì của client. Như vậy, khái niệm “trạng thái” ở đây được hiểu là dữ liệu.

Nguyên lý hoạt động của stateless là mỗi yêu cầu được xử lý độc lập và không phụ thuộc vào bất kỳ thông tin trạng thái nào được lưu trữ trước đó.

Mỗi yêu cầu sẽ chứa tất cả các thông tin cần thiết để máy chủ xử lý nó, bao gồm các thông tin xác thực và ủy quyền để xác định quyền truy cập của người dùng. Các thông tin này thường được truyền qua các tiêu đề HTTP hoặc được mã hóa trong các mã thông báo, chẳng hạn như JSON Web Tokens (JWT).

1. Nguyên hoạt động

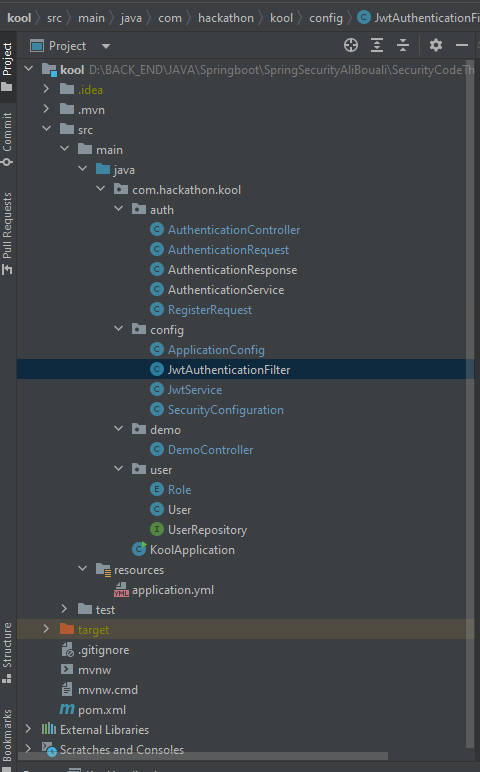


Cách thức hoạt động:

* Người dùng đăng nhập bằng cách cung cấp thông tin đăng nhập của mình.
* Ứng dụng gửi yêu cầu đến Spring Security để xác thực thông tin đăng nhập của người dùng.
* Spring Security sẽ kiểm tra thông tin đăng nhập của người dùng và nếu thông tin hợp lệ, nó sẽ tạo ra một JWT token và trả về cho ứng dụng.
* Ứng dụng sẽ sử dụng JWT token để xác thực và phân quyền truy cập của người dùng trong các yêu cầu tiếp theo.
* Khi người dùng yêu cầu một tài nguyên nào đó, ứng dụng sẽ kiểm tra xem JWT token có hợp lệ không.
* Nếu JWT token hợp lệ, ứng dụng sẽ cho phép người dùng truy cập vào tài nguyên đó.

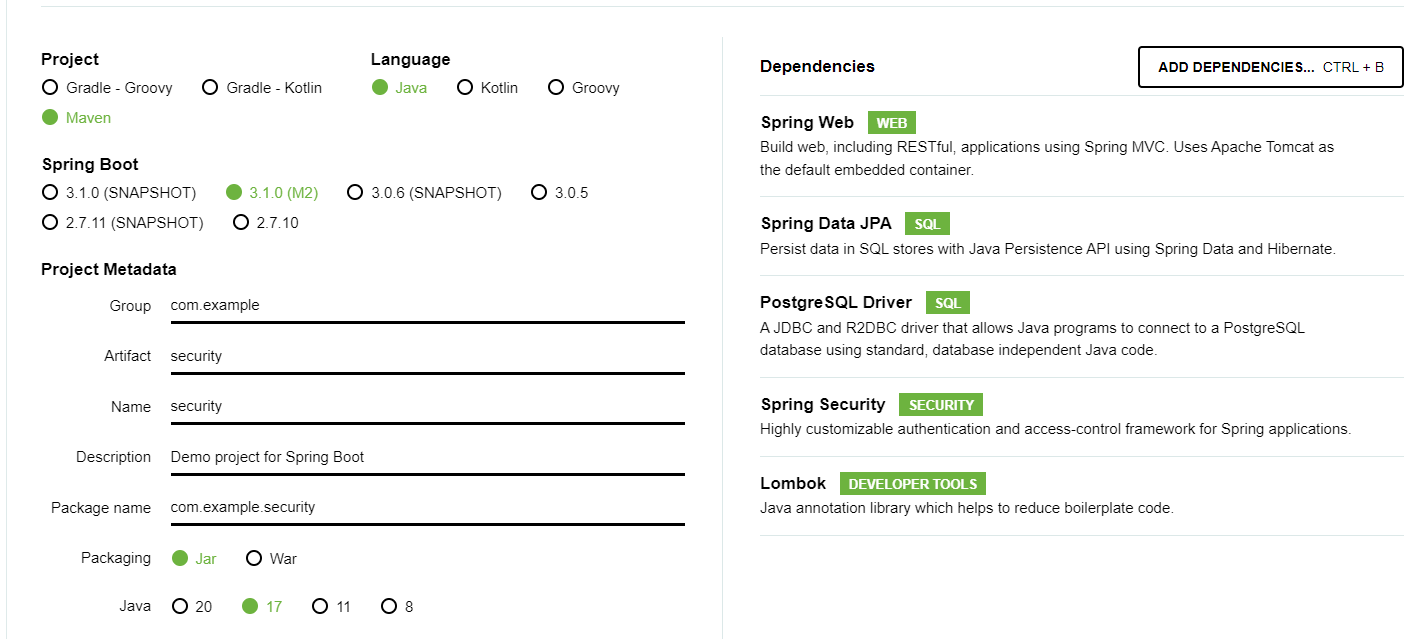
1. Cài đặt

*Path project*



1. Tạo project

* Link: < <https://start.spring.io/>>



1. Cấu hình file aplication.yml

* Đổi định dạng file .properties -> .yml
* Tạo database trong postgresql
* Kết nối với database

spring:  
 datasource:  
 url: jdbc:postgresql://localhost:5432/kool  
 username: postgres  
 password: 123  
 driver-class-name: org.postgresql.Driver  
 jpa:  
 hibernate:  
 ddl-auto: update  
 show-sql: true  
 properties:  
 hibernate:  
 format\_sql: true  
 database: postgresql  
 database-platform: org.hibernate.dialect.PostgreSQLDialect

1. Tạo Entity User

@Data  
@Builder  
@NoArgsConstructor  
@AllArgsConstructor  
@Entity  
@Table(name = "users")  
public class User implements UserDetails {  
 @Id  
 @GeneratedValue  
 private Long id;  
 @Column(name = "first\_name")  
 private String firstname;  
 @Column(name = "last\_name")  
 private String lastname;  
 @Column(name = "user\_name")  
 private String username;  
 private String email;  
 private String password;  
 @Enumerated(EnumType.*STRING*)  
 private Role role;  
  
 @Override  
 public Collection<? extends GrantedAuthority> getAuthorities() {  
 return List.*of*(new SimpleGrantedAuthority(role.name()));  
 }  
  
 @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;  
 }  
}

* Mặc định Spring Security sử dụng một đối tượng UserDetails để chứa toàn bộ thông tin về người dùng. Vì vậy, chúng ta cần tạo ra một class mới giúp chuyển các thông tin của User thành UserDetails.
* GrantedAuthority là một interface đại diện cho quyền được cấp cho một người dùng trong hệ thống. Nó chứa thông tin về quyền hạn của một người dùng được cấp trong hệ thống. Ví dụ, trong một ứng dụng web, một số quyền hạn có thể là "ROLE\_ADMIN", "ROLE\_USER", "ROLE\_MANAGER", v.v.

1. Tạo Enum Role để lưu các giá trị Authorization

public enum Role {  
 *USER*, *ADMIN*}

1. Tạo UserRepository

public interface UserRepository extends JpaRepository<User, Long> {  
 Optional<User> findByUsername(String Username);  
}

1. Tạo bộ lọc filter: JwtAuthenticationFilter

@Component  
@RequiredArgsConstructor  
public class JwtAuthenticationFilter extends OncePerRequestFilter {  
  
 @Override  
 protected void doFilterInternal(  
 HttpServletRequest request,  
 HttpServletResponse response,  
 FilterChain filterChain  
 ) throws ServletException, IOException {

}  
}

* Bộ lọc được sử dụng để thực hiện các xử lý liên quan đến xác thực, phân quyền và bảo mật cho ứng dụng web. Filter được sử dụng để kiểm tra và xử lý các yêu cầu gửi đến ứng dụng và tùy thuộc vào cấu hình, có thể cho phép hoặc từ chối truy cập đến các tài nguyên trong ứng dụng.
* OncePerRequestFilter là một lớp filter trong Spring Framework được sử dụng để đảm bảo rằng filter sẽ được gọi duy nhất một lần cho mỗi request. Nó cung cấp một cơ chế để kiểm tra và thực hiện các hành động với các request đến ứng dụng.
* Một trong những ứng dụng phổ biến của OncePerRequestFilter trong Spring Security là xác thực và phân quyền dựa trên JWT (JSON Web Token). Filter này sẽ lấy token từ request, xác minh tính hợp lệ của token, trích xuất thông tin xác thực và phân quyền từ token, và sau đó lưu trữ các thông tin này để sử dụng trong việc kiểm tra quyền truy cập của người dùng trong các yêu cầu tiếp theo.

1. Tạo lớp JwtService

@Service  
public class JwtService {  
 //Khóa bảo mật  
 //SECRET\_KEY được sử dụng để tạo chữ ký cho JWT  
 private static final String *SECRET\_KEY* = "67566B5970337336763979244226452948404D6351655468576D5A7134743777";  
   
 //Trích xuất tên đăng nhập của người dùng từ chuỗi JSON Web Token (JWT) bằng cách gọi phương thức extractClaim() đã được giải thích trước đó.  
 public String extractUsername(String token) {  
 //getSubject để lấy username trả về chuỗi == userName  
 return extractClaim(token, Claims::getSubject);  
 }  
  
 //Map để lưu trữ các thông tin tùy chỉnh của người dùng  
 public String generateToken(UserDetails userDetails) {  
 return generateToken(new HashMap<>(), userDetails);  
 }  
  
 //hàm tạo mã token  
 public String generateToken(Map<String, Object> extraClaims, UserDetails userDetails) {  
 return Jwts  
 .*builder*()  
 .setClaims(extraClaims)  
 .setSubject(userDetails.getUsername())  
 .setIssuedAt(new Date(System.*currentTimeMillis*()))  
 .setExpiration(new Date(System.*currentTimeMillis*() + 1000 \* 60 \* 24))  
 .signWith(getSignInkey(), SignatureAlgorithm.*HS256*)  
 .compact();  
 }  
  
 //Kiểm tra token hợp lệ không  
 public boolean isTokenValid(String token, UserDetails userDetails) {  
 final String username = extractUsername(token);  
 return (username.equals(userDetails.getUsername())) && !isTokenExpired(token);  
 }  
  
 //Kiểm tra ngày hết hạn của token  
 private boolean isTokenExpired(String token) {  
 return extractExpiration(token).before(new Date());  
 }  
  
 private Date extractExpiration(String token) {  
 return extractClaim(token, Claims::getExpiration);  
 }  
  
 //để trích xuất các thông tin cụ thể từ chuỗi JSON Web Token (JWT). Các thông tin này được đóng gói trong đối tượng Claims.  
 public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {  
 final Claims claims = extractAllClaims(token);  
 return claimsResolver.apply(claims);  
 }  
  
 //truy cập các trường thông tin trong chuỗi JWT(token), chúng ta trả về đối tượng Claims  
 private Claims extractAllClaims(String token) {  
 return Jwts  
 .*parserBuilder*()  
 .setSigningKey(getSignInkey())  
 .build()  
 .parseClaimsJws(token)  
 .getBody();  
 }  
  
 //Phương thức này được sử dụng để lấy khóa ký để giải mã hoặc ký một chuỗi JWT.  
 private Key getSignInkey() {  
 byte[] keyBytes = Decoders.*BASE64*.decode(*SECRET\_KEY*);  
 return Keys.*hmacShaKeyFor*(keyBytes);  
 }  
}

* SECRET\_KEY là một chuỗi ký tự bí mật được sử dụng để mã hóa và giải mã thông tin trong các hệ thống bảo mật. Trong các ứng dụng web, SECRET\_KEY thường được sử dụng để tạo và xác thực JSON Web Token (JWT).
* SECRET\_KEY được sử dụng để tạo chữ ký cho JWT, đảm bảo tính hợp lệ của JWT và đảm bảo rằng thông tin trong JWT không bị thay đổi bởi bên thứ ba. Nếu SECRET\_KEY không được bảo vệ tốt, nó có thể bị lộ và thông tin trong JWT có thể bị đánh cắp hoặc giả mạo.
* Chúng ta có thể lấy SECRET\_KEY ở trang web < https://www.allkeysgenerator.com/Random/Security-Encryption-Key-Generator.aspx > và chọn 256 bit
* Cài hạn của token là 1 ngày.
* Claims là một đối tượng JSON chứa các thông tin xác thực về người dùng hoặc các thông tin khác về JWT. Nó được sử dụng để chứa thông tin định danh và quyền truy cập của người dùng và được đưa vào trong phần Payload của JWT. Claims bao gồm các cặp key-value, với mỗi key đại diện cho một thông tin cụ thể và value.
* Spring Security khởi tạo một private key sử dụng thuật toán mã hóa HMAC-SHA (Hash-based Message Authentication Code - Secure Hash Algorithm). SECRET\_KEY được giải mã từ chuỗi Base64 và lưu trữ trong mảng byte keyBytes. Sau đó, phương thức hmacShaKeyFor() của lớp Keys được gọi để tạo ra một private key sử dụng thuật toán HMAC-SHA từ mảng byte keyBytes. Private key này được sử dụng để tạo chữ ký cho JSON Web Token (JWT).

1. Tạo lớp cấu hình ApplicationConfig

@Configuration  
@RequiredArgsConstructor  
public class ApplicationConfig {  
 private final UserRepository repository;  
  
 @Bean  
 public UserDetailsService userDetailsService() {  
 return username -> repository.findByUsername(username)  
 .orElseThrow(()->new UsernameNotFoundException("User not found"));  
}

* Trong mã nguồn trên, một đối tượng UserDetailsService được khởi tạo và trả về bởi phương thức userDetailsService() của lớp cấu hình. UserDetailsService là một interface của Spring Security được sử dụng để cung cấp thông tin chi tiết về người dùng cho việc xác thực trong quá trình đăng nhập.
* Phương thức userDetailsService() trả về một đối tượng UserDetailsService được xây dựng bằng cách sử dụng lambda expression. Nó nhận vào một tham số đầu vào là username, sau đó sử dụng repository.findByUsername(username) để tìm kiếm thông tin của người dùng trong cơ sở dữ liệu. Nếu người dùng có tồn tại, thông tin của họ sẽ được trả về dưới dạng một đối tượng UserDetails của Spring Security. Nếu không tìm thấy người dùng, phương thức sẽ ném một ngoại lệ UsernameNotFoundException.

1. Cài đặt bộ lọc

@Component  
@RequiredArgsConstructor  
public class JwtAuthenticationFilter extends OncePerRequestFilter {  
 private final JwtService jwtService;  
 private final UserDetailsService userDetailsService;  
  
 @Override  
 protected void doFilterInternal(  
 HttpServletRequest request,  
 HttpServletResponse response,  
 FilterChain filterChain  
 ) throws ServletException, IOException {  
 final String authHeader = request.getHeader("Authorization");  
 final String jwt;  
 final String userEmail;  
 if(authHeader==null || !authHeader.startsWith("Bearer ")) {  
 filterChain.doFilter(request, response);  
 return;  
 }  
 //trích thông báo jwt  
 jwt = authHeader.substring(7);  
 //trích xuất username người dùng  
 userEmail = jwtService.extractUsername(jwt);  
 if(userEmail!=null && SecurityContextHolder.*getContext*().getAuthentication()==null) {  
 UserDetails userDetails = this.userDetailsService.loadUserByUsername(userEmail);  
  
 if(jwtService.isTokenValid(jwt, userDetails)) {  
 UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(  
 userDetails,  
 null,  
 userDetails.getAuthorities()  
 );  
 authToken.setDetails(  
 new WebAuthenticationDetailsSource().buildDetails(request)  
 );  
 SecurityContextHolder.*getContext*().setAuthentication(authToken);  
 }  
 }  
 filterChain.doFilter(request, response);  
 }  
}

* Sử dụng đối tượng HttpServletRequest để lấy thông tin về tiêu đề Authorization của yêu cầu và trích xuất token JWT từ tiêu đề đó. Sau đó, nếu token không hợp lệ (không có tiêu đề Authorization hoặc token không bắt đầu bằng "Bearer"), thì các bộ lọc khác của Spring Security sẽ được thực hiện và yêu cầu sẽ được truyền tiếp đến các bộ lọc khác.
* Nếu token JWT hợp lệ, phương thức sẽ trích xuất tên người dùng từ token JWT và kiểm tra xem tên người dùng đã được xác thực hay chưa. Nếu tên người dùng đã được xác thực, phương thức sẽ tạo một đối tượng UserDetails sử dụng UserDetailsService đã được cấu hình và kiểm tra tính hợp lệ của token JWT. Nếu token JWT hợp lệ, phương thức sẽ tạo một đối tượng UsernamePasswordAuthenticationToken để đại diện cho thông tin xác thực của người dùng, sau đó đặt đối tượng này vào SecurityContextHolder của Spring Security để lưu trữ thông tin xác thực của người dùng.
* Cuối cùng, phương thức sẽ gọi filterChain.doFilter(request, response) để cho phép các bộ lọc khác của Spring Security thực hiện tiếp các xử lý trên yêu cầu và phản hồi.

1. Cấu hình và kích hoạt Security

@Configuration  
@EnableWebSecurity  
@RequiredArgsConstructor  
public class SecurityConfiguration {  
 private final JwtAuthenticationFilter jwtAuthFilter;  
 private final AuthenticationProvider authenticationProvider;  
  
 @Bean  
 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
 http  
 .csrf()  
 .disable()  
 .authorizeHttpRequests()  
 .requestMatchers("/api/v1/auth/\*\*")  
 .permitAll()  
 .anyRequest()  
 .authenticated()  
 .and()  
 .sessionManagement()  
 .sessionCreationPolicy(SessionCreationPolicy.*STATELESS*)  
 .and()  
 .authenticationProvider(authenticationProvider)  
 .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);  
  
 return http.build();  
 }  
}

* JwtAuthenticationFilter là bộ lọc ta đã định nghĩa trước đó
* AuthenticationProvider là một interface trong Spring Security được sử dụng để xác thực người dùng trong quá trình đăng nhập. Nó chứa phương thức authenticate() để thực hiện xác thực và trả về một đối tượng Authentication trong trường hợp xác thực thành công hoặc ném ra một ngoại lệ AuthenticationException nếu không xác thực được.
* Lớp cấu hình này cấu hình một bộ lọc Spring Security cho các yêu cầu gửi tới ứng dụng. Trong phương thức securityFilterChain(), lớp cấu hình thiết lập một loạt các cấu hình bảo mật, bao gồm:

+ Disable CSRF (Cross-Site Request Forgery) để giảm thiểu các cuộc tấn công CSRF.

+ Xác thực các yêu cầu gửi tới các địa chỉ khác nhau trên API của ứng dụng. Phương thức authorizeHttpRequests() được sử dụng để chỉ định các yêu cầu nào sẽ được xác thực và yêu cầu nào sẽ được bỏ qua.

+ Khởi tạo phiên cho các yêu cầu đã được xác thực. SessionCreationPolicy.STATELESS được sử dụng để chỉ định rằng phiên sẽ không được sử dụng cho các yêu cầu đã được xác thực.

+ Sử dụng AuthenticationProvider để xác thực người dùng.

+ Thêm JwtAuthenticationFilter vào danh sách các bộ lọc trước đối tượng UsernamePasswordAuthenticationFilter. JwtAuthenticationFilter sẽ được thực hiện trước đối tượng UsernamePasswordAuthenticationFilter và được sử dụng để xác thực thông tin của token JWT trên tiêu đề của yêu cầu.

* Cuối cùng, phương thức trả về đối tượng SecurityFilterChain đã được thiết lập để xác thực và phân quyền cho các yêu cầu gửi tới ứng dụng.

1. Cấu hình thêm cho lớp ApplicationConfig

@Bean  
public AuthenticationProvider authenticationProvider() {  
 DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();  
 authProvider.setUserDetailsService(userDetailsService());  
 authProvider.setPasswordEncoder(passwordEncoder());  
 return authProvider;  
}  
  
@Bean  
public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {  
 return config.getAuthenticationManager();  
}  
  
@Bean  
public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
}

* authenticationProvider(): trả về một AuthenticationProvider, cụ thể là một DaoAuthenticationProvider được cấu hình với UserDetailsService và PasswordEncoder. AuthenticationProvider được sử dụng để xác thực thông tin đăng nhập của người dùng.
* authenticationManager(config): trả về một AuthenticationManager. AuthenticationManager quản lý quá trình xác thực cho các yêu cầu đến ứng dụng. Nó sử dụng các AuthenticationProvider để thực hiện quá trình xác thực.
* passwordEncoder(): trả về một PasswordEncoder, cụ thể là một BCryptPasswordEncoder. PasswordEncoder được sử dụng để mã hóa mật khẩu của người dùng.

Tổng quan, các Bean trên được sử dụng để cấu hình quá trình xác thực trong ứng dụng Spring Security. DaoAuthenticationProvider là một AuthenticationProvider cơ bản mà sử dụng UserDetailsService và PasswordEncoder để xác thực thông tin đăng nhập của người dùng. AuthenticationManager quản lý quá trình xác thực cho các yêu cầu đến ứng dụng và sử dụng các AuthenticationProvider để thực hiện quá trình xác thực. BCryptPasswordEncoder là một PasswordEncoder được sử dụng để mã hóa mật khẩu của người dùng trước khi được lưu trữ trong cơ sở dữ liệu.

1. Tạo lớp AuthenticationController

@RestController  
@RequestMapping("/api/v1/auth")  
@RequiredArgsConstructor  
public class AuthenticationController {  
 private final AuthenticationService service;  
  
 @PostMapping("/register")  
 public ResponseEntity<AuthenticationResponse> register(  
 @RequestBody RegisterRequest request  
 ) {  
 return ResponseEntity.*ok*(service.register(request));  
 }  
 @PostMapping("/authenticate")  
 public ResponseEntity<AuthenticationResponse> authenticate(  
 @RequestBody AuthenticationRequest request  
 ) {  
 return ResponseEntity.*ok*(service.authenticate(request));  
 }  
}

* Chỉ định 2 endpoint đăng kí và xác thực(đăng nhập).

1. Tạo AuthenticationResponse

@Data  
@Builder  
@AllArgsConstructor  
@NoArgsConstructor  
public class AuthenticationResponse {  
  
 private String token;  
}

* Phản hồi trả về từ phía máy chủ khi người dùng đăng nhập thành công và được cấp mã thông báo xác thực JWT. Nó có thuộc tính token để lưu trữ mã thông báo JWT được tạo ra bởi máy chủ để người dùng có thể sử dụng để truy cập các tài nguyên được bảo vệ.

1. Tạo lớp RegisterRequest

@Data  
@Builder  
@AllArgsConstructor  
@NoArgsConstructor  
public class RegisterRequest {  
 private String firstname;  
 private String lastname;  
 private String email;  
 private String username;  
 private String password;  
}

* Các thông tin đăng kí.

1. Tạo AuthenticationRequest

@Data  
@Builder  
@AllArgsConstructor  
@NoArgsConstructor  
public class AuthenticationRequest {  
 private String username;  
 private String password;  
}

* Thông tin đăng nhập gồm tài khoản mật khẩu

1. Tạo lớp AuthenticationService

@Service  
@RequiredArgsConstructor  
public class AuthenticationService {  
 private final UserRepository repository;  
 private final PasswordEncoder passwordEncoder;  
 private final JwtService jwtService;  
 private final AuthenticationManager authenticationManager;  
  
 public AuthenticationResponse register(RegisterRequest request) {  
 var user = User.*builder*()  
 .firstname(request.getFirstname())  
 .lastname(request.getLastname())  
 .username(request.getUsername())  
 .email(request.getEmail())  
 .password(passwordEncoder.encode(request.getPassword()))  
 .role(Role.*USER*)  
 .build();  
 repository.save(user);  
 var jwtToken = jwtService.generateToken(user);  
 return AuthenticationResponse.*builder*()  
 .token(jwtToken)  
 .build();  
 }  
  
 public AuthenticationResponse authenticate(AuthenticationRequest request) {  
 authenticationManager.authenticate(  
 new UsernamePasswordAuthenticationToken(  
 request.getUsername(),  
 request.getPassword()  
 )  
 );  
 var user = repository.findByUsername(request.getUsername())  
 .orElseThrow();  
 var jwtToken = jwtService.generateToken(user);  
 return AuthenticationResponse.*builder*()  
 .token(jwtToken)  
 .build();  
 }  
}

* Phương thức register nhận vào một RegisterRequest sau đó tạo một đối tượng User nhận các thông tin của RegisterRequest lưu User sau đó tạo mã Token và trả về đối tượng AuthenticationResponse chứa mã đó.
* Phương thức authenticate nhận đầu vào là đối tượng AuthenticationRequest chứa thông tin đăng nhập của người dùng. Phương thức sử dụng authenticationManager để xác thực thông tin đăng nhập này. Nếu thông tin đăng nhập không hợp lệ, phương thức sẽ ném ra ngoại lệ AuthenticationException. Nếu thông tin đăng nhập hợp lệ, phương thức sẽ lấy thông tin người dùng từ UserRepository dựa trên username được cung cấp trong AuthenticationRequest. Nếu không tìm thấy người dùng, phương thức sẽ ném ra ngoại lệ. Nếu tìm thấy người dùng, phương thức sẽ sử dụng jwtService để tạo ra JWT token và trả về đối tượng AuthenticationResponse chứa JWT token này.

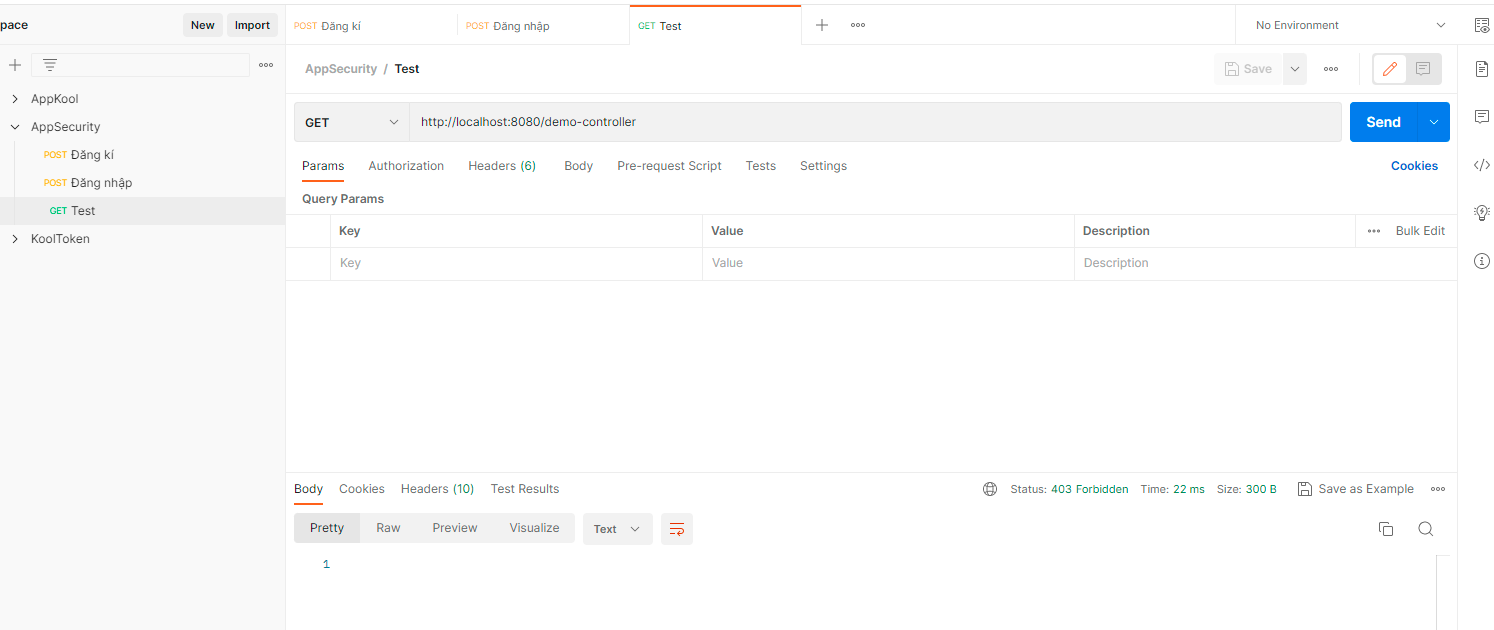
1. Tạo lớp demo

@RestController  
@RequestMapping("/demo-controller")  
public class DemoController {  
 @GetMapping  
 public ResponseEntity<String> sayHello() {  
 return ResponseEntity.*ok*("Hello toi la Tran Kim Hoang");  
 }  
}

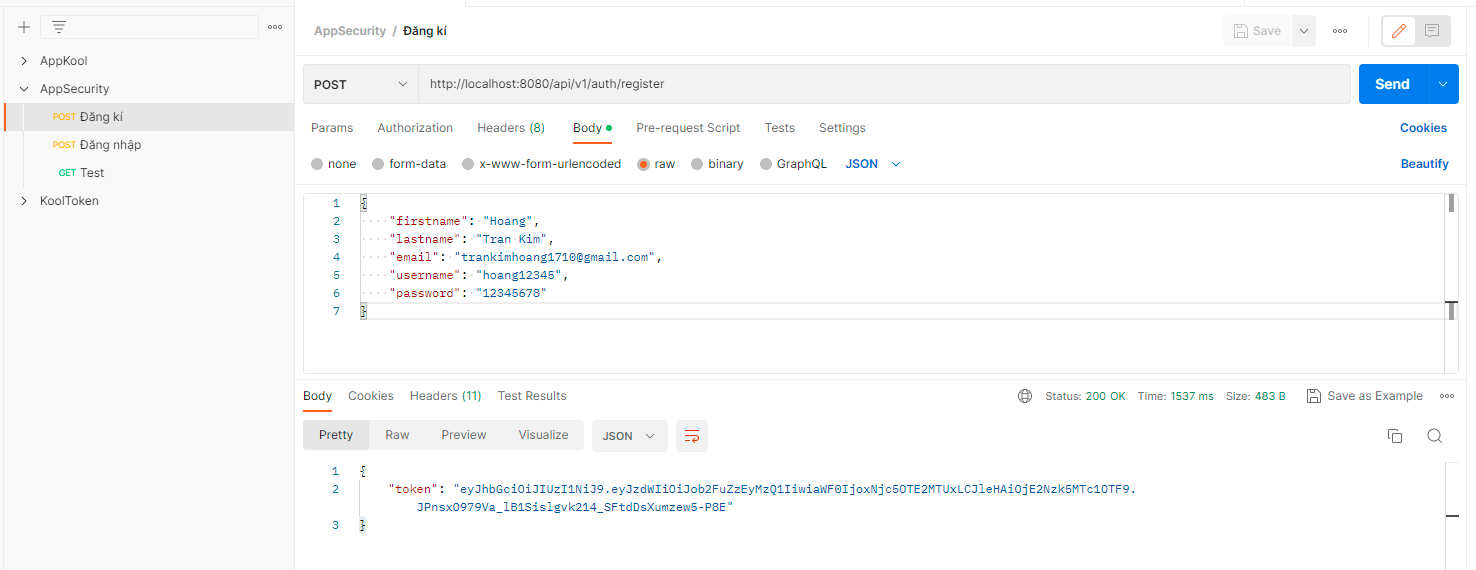
1. Test API

Mở ứng dụng Postman thực hiện các Resquest:

* Truy cập trang demo khi chưa xác thực:



* Đăng kí mới:

 *Đăng kí thành công!*

* Đăng nhập:

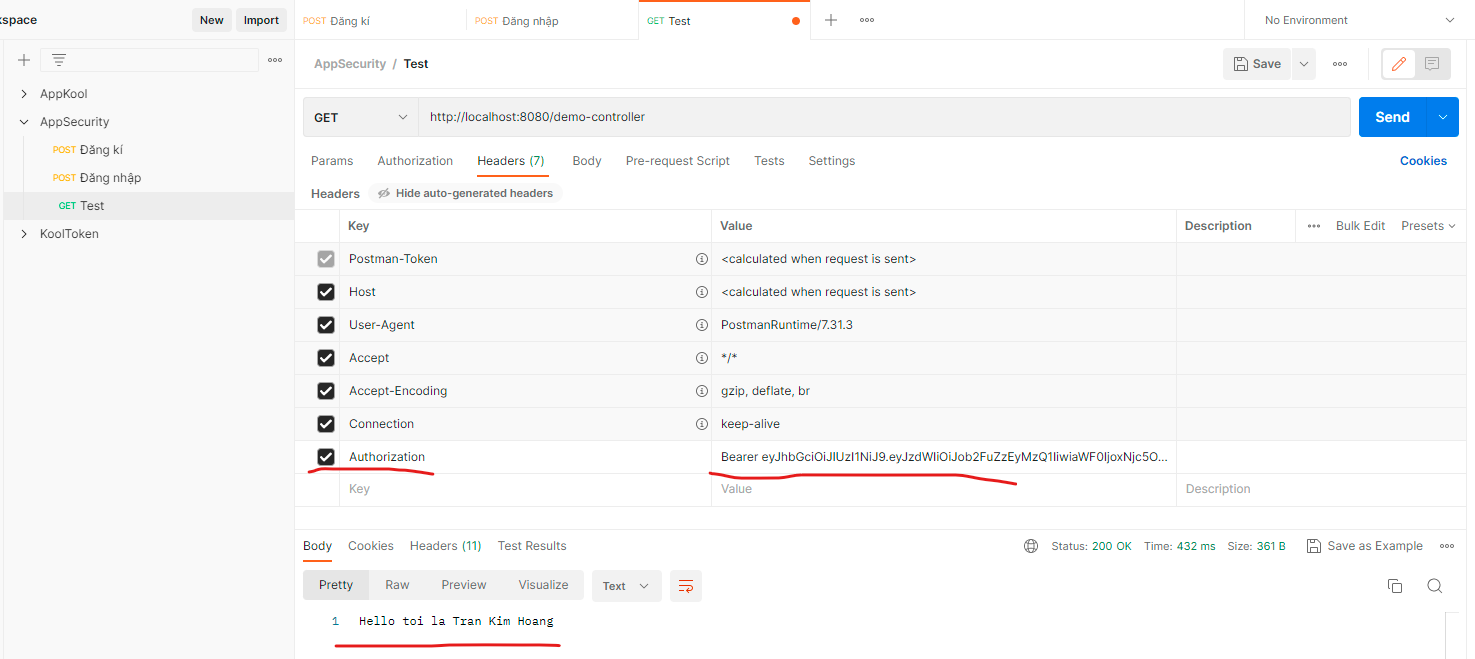
*Ảnh có chứa văn bản

Description automatically generatedXác thực thất bại!*

*Ảnh có chứa văn bản

Description automatically generatedXác thực thành công!*

* Test demo khi có mã xác thực: các bạn thêm vào Header key:value chứa mã xác thực mà vừa đăng nhập.

*Thành công!*

1. Tổng kết

* Spring Security là một kiến thức rất quan trọng trên đây là một hướng dẫn được trích từ các nguồn khác nhau và tổng hợp lại. Phần này là kiến thức khó nên mong các bạn đọc đi đọc lại để hiểu.
* Link git có chứa dự án, các file tạo bảng và test Api < https://github.com/TranKimHoang17102003/kool>