

CENTRO UNIVERSITÁRIO FACENS
CURSOS TECNOLÓGICOS



Desenvolvimento Web Back-End

Próximas Datas

- **22/05:** Matéria e exercícios + avaliação do back-end do Projeto Final
- **29/05:** Matéria + **considerações finais para AF e AS**
- **05/06:** Apresentação do Projeto de Startup Final (não haverá aula)
- **12/06:** Avaliação Final (AF)
- **19/06:** Feriado
- **26/06:** Avaliação Substitutiva

AVALIAÇÃO DA CPA

ATENÇÃO, A AVALIAÇÃO DA CPA COMEÇOU!

**CHEGOU A HORA DE COLHER IDEIAS
PARA TRANSFORMAR EM RESULTADOS**


RESPONDA ATRAVÉS DO QR CODE:



The illustration features a hand holding a pen over a bar chart with three bars of increasing height. To the right is a large QR code. Further right is a blue robot wearing a cap and holding a clipboard with the CPA logo. The background is dark blue with a subtle pattern.

Facens | CPA
Comissão Própria de Avaliação

? ****Por que sua participação é importante?****

- ✓ Porque ****ajuda a identificar pontos de melhoria**** nos cursos, nas estruturas e nos serviços.
 - ✓ Porque é por meio da CPA que ****suas opiniões chegam diretamente à gestão acadêmica****.
 - ✓ Porque ****uma instituição só melhora com a escuta ativa dos seus estudantes****!
-  ****Sua voz é importante!**** Vamos construir juntos uma ****Facens ainda melhor****.

Arquitetura Web Spring Security

Spring Security

Spring Security é um framework da família Spring que fornece **autenticação, autorização** e diversas funcionalidades de segurança para aplicações Java, especialmente aplicações web construídas com Spring Boot.

Ele cuida de proteger sua aplicação contra ameaças como:

- Acesso não autorizado;
- Ataques de sessão;
- Cabeçalhos de segurança;
- Entre outros.

Spring Security

Principais Funcionalidades:

1. **Autenticação:** Verificar a identidade do usuário (login).
2. **Autorização:** Controlar o acesso baseado em funções (roles) e permissões.
3. **Segurança de APIs REST:** Suporte a JWT.
4. **Segurança no nível de método:** Com anotações como `@PreAuthorize` ou `@Secured`.
5. **Proteções padrão:** segurança de sessão e cabeçalhos HTTP seguros.
6. **Integração com provedores externos:** Google, Facebook e GitHub.

Spring Security

Existem vários tipos de autenticação que podem ser utilizados em sistemas e aplicativos. Alguns dos tipos mais comuns incluem:

- **Autenticação baseada em formulários:** este tipo de autenticação é usado com mais frequência em aplicativos da web. O usuário fornece suas credenciais de login, como nome de usuário e senha, em um formulário da web para autenticação.
- **Autenticação baseada em tokens:** esse tipo de autenticação envolve o uso de um token (como um token JWT) para autenticar usuários. O token é geralmente enviado no cabeçalho de uma solicitação HTTP para o servidor, que pode então verificar se o token é válido e autorizar o usuário.



Facens

Spring Security

- **Autenticação de terceiros:** este tipo de autenticação permite que os usuários usem credenciais de login de uma conta de terceiros, como Facebook ou Google, para autenticação em um aplicativo.
- **Autenticação de certificado digital:** esse tipo de autenticação usa certificados digitais para verificar a identidade do usuário. Isso é comum em sistemas que exigem um nível mais alto de segurança, como em sistemas de comércio eletrônico ou em serviços financeiros.

Exemplo básico de configuração de Spring Security

1. Dependência no Maven

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

2. Configuração de Segurança

```
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import  
org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.web.SecurityFilterChain;
```

```
@Configuration  
public class SecurityConfig {
```

Continuação do código:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public/**").permitAll() // Liberado
            .anyRequest().authenticated() // Outros endpoints exigem login
        )
        .httpBasic() // Autenticação via Basic Auth
        .and()
        .csrf().disable(); // Desabilitado (para APIs, se necessário)
    return http.build();
}
```

3. Usuário em memória (para testes)

```
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.security.core.userdetails.User;  
import org.springframework.security.core.userdetails.UserDetails;  
import  
org.springframework.security.provisioning.InMemoryUserDetailsManager;
```

```
@Configuration  
public class UserConfig {
```

Continuação do código:

```
@Bean
public InMemoryUserDetailsManager userDetailsService() {
    UserDetails user = User
        .withUsername("usuario")
        .password("{noop}senha") // {noop} significa sem criptografia
        .roles("USER")
        .build();

    return new InMemoryUserDetailsManager(user);
}
```

Anotações Comuns no Spring Security:

- **@EnableWebSecurity** — Ativa a segurança web (opcional nas versões mais novas do Spring Boot).
- **@EnableMethodSecurity** — Ativa segurança nos métodos (substitui o antigo @EnableGlobalMethodSecurity).
- **@PreAuthorize("hasRole('ADMIN')")** — Protege métodos com base na role.
- **@Secured("ROLE_ADMIN")** — Outra forma de proteger métodos (mais simples).

Autenticação JWT

JWT - Json Web Tokens

É um **padrão aberto (RFC 7519)** que define uma maneira compacta, segura e independente de transmitir informações entre duas partes como um **objeto JSON**, que pode ser verificado e confiável porque é **assinado digitalmente**.

Para que serve o JWT?

Ele é muito usado para:

- **Autenticação:** Após o login, o servidor gera um token JWT e envia para o cliente. O cliente usa esse token para autenticar futuras requisições.
- **Autorização:** O token contém informações sobre o usuário e suas permissões.

Também pode ser usado para troca segura de informações.

Estrutura JWT?

Um JWT tem **3 partes**, separadas por pontos (.).

<HEADER>.<PAYLOAD>.<SIGNATURE>

1. Header (Cabeçalho)

Contém informações sobre o algoritmo de assinatura e o tipo de token.

O cabeçalho *geralmente* consiste em duas partes: o tipo de token, que é JWT, e o algoritmo de assinatura que está sendo usado, como HMAC SHA256 ou RSA.

Estrutura JWT?

2. Payload (Corpo)

Pode conter:

- Dados públicos: username, email, roles.
- Dados reservados: sub (subject), exp (expiração), iat (emitido em), iss (emissor).
- Dados privados: definidos pela aplicação.

Contém as declarações. As declarações são declarações sobre uma entidade e dados adicionais.

Estrutura JWT?

3. Signature (Assinatura)

Para criar a parte de assinatura, você precisa pegar o cabeçalho codificado, o payload codificado, uma senha, o algoritmo especificado no cabeçalho e assiná-lo.

A assinatura é usada para verificar se a mensagem não foi alterada ao longo do caminho e, no caso de tokens assinados com chave privada, também pode verificar se o remetente do JWT é quem diz ser.

JWT - Json Web Tokens

A saída são três strings Base64-URL separadas por pontos que podem ser facilmente passadas em ambientes HTML e HTTP:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09o1PSyXnrXCjTwXyr4BsezDI1AVTmud2fU4
```

Como funcionam os tokens da Web JSON?

- Na autenticação, quando o usuário fizer **login com sucesso** usando suas credenciais, um JSON Web **Token será retornado**. Como os tokens são credenciais, deve-se tomar muito cuidado para evitar problemas de segurança. Em geral, você **não deve manter os tokens por mais tempo do que o necessário**.
- Sempre que o usuário deseja **acessar uma rota ou recurso protegido**, o agente do usuário **deve enviar o JWT**, normalmente no cabeçalho **Authorization** usando o esquema **Bearer** . O conteúdo do cabeçalho deve ter a seguinte aparência:

`Authorization: Bearer <token>`

JWT - Json Web Tokens

As rotas protegidas do servidor verificarão se há um JWT válido no Authorization e, se estiver presente, o usuário terá permissão para acessar recursos protegidos.

Se o JWT contiver os dados necessários, a necessidade de consultar o banco de dados para determinadas operações pode ser reduzida.

JWT Service

O JwtService é uma classe responsável por toda a **gestão dos tokens JWT** dentro de uma aplicação, geralmente uma API protegida com **Spring Security**.

Ele tem 3 funções principais:

1. Gerar Token (Autenticação)

Quando o usuário faz login corretamente, o JwtService gera um **token JWT** que contém informações sobre o usuário.

Esse token é enviado para o cliente (navegador, app mobile, etc.).

O cliente usa esse token para acessar recursos protegidos da API.

Exemplo:

```
String token = jwtService.generateToken("usuario123");
```

2. Extrair Informações

O JwtService consegue **ler** o token e extrair informações como:

- O username (sub)
- Data de expiração (exp)
- Outros dados que você colocar no token

Exemplo:

```
String username = jwtService.extractUsername(token);
```

3. Validar Token (Autorização)

Toda vez que o cliente faz uma requisição protegida, o token enviado é:

- Verificado se não foi alterado (assinatura digital válida).
- Verificado se ainda não expirou.
- Verificado se realmente pertence ao usuário.

Outros exemplos

JwtService

- Criar um pacote security e uma classe JwtService:

```
@Service
public class JwtService {

    @Value("${security.jwt.expiracao}")
    private String expiracao;

    @Value("${security.jwt.chave-assinatura}")
    private String chaveAssinatura;

    public String gerarToken(Usuario usuario) {

        long expString = Long.valueOf(expiracao);

        LocalDateTime dataHoraExpiracao = LocalDateTime.now().plusMinutes(expString);

        Date data = Date.from(dataHoraExpiracao.atZone(ZoneId.systemDefault()).toInstant());

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

        claims.put("id", usuario.getId());

        claims.put("email", usuario.getEmail());

        claims.put("perfil", usuario.getPerfil());

        return Jwts.builder()

            .setClaims(claims)

            .setExpiration(data)

            .signWith(getSignKey(), SignatureAlgorithm.HS256)

            .compact();

    }
}
```

```
public Claims obterClaims(String token) throws ExpiredJwtException {

    return Jwts.parserBuilder()

        .setSigningKey(getSignKey())

        .build()

        .parseClaimsJws(token)

        .getBody();

}

private Key getSignKey() {

    byte[] key = Decoders.BASE64.decode(chaveAssinatura);

    return Keys.hmacShaKeyFor(key);

}

public boolean validarToken(String token) {

    try {

        Claims claims = obterClaims(token);

        Date dataExpiracao = claims.getExpiration();

        LocalDateTime data = dataExpiracao.toInstant().atZone(ZoneId.systemDefault()).toLocalDateTime();

        return !LocalDateTime.now().isAfter(data);

    } catch (Exception ex) {

        return false;

    }

}

public String obterLoginUsuario(String token) throws ExpiredJwtException {

    Claims c = obterClaims(token);

    return (String) c.get("email");

}
}
```

Testar JwtService

- Crie um método main na classe JwtService para testarmos:

```
public static void main(String[] args){
    ConfigurableApplicationContext contexto = SpringApplication.run(ExemplospringdatajpaApplication.class);
    JwtService service = contexto.getBean(JwtService.class);
    UsuarioRepository usuarioRepository = contexto.getBean(UsuarioRepository.class);
    PasswordEncoder passwordEncoder = contexto.getBean(PasswordEncoder.class);

    Usuario usuario = new Usuario(0, "Cilene", "cilene.real@facens.br", passwordEncoder.encode("123"),
                                "Administrador");
    String token = service.gerarToken(usuario);
    System.out.println(token);
    boolean isValid = service.validarToken(token);
    System.out.println("Token válido? " + isValid);
    System.out.println("Usuário: " + service.obterLoginUsuario(token));

    usuarioRepository.save(usuario);
}
```

ApplicationConfig

- Criar uma classe ApplicationConfig no pacote config:

```
@Configuration
```

```
@RequiredArgsConstructor
```

```
public class ApplicationConfig {  
    private final UsuarioRepository repository;
```

```
    @Bean
```

```
    public UserDetailsService userDetailsService() {  
        return username -> repository.findByEmail(username);  
    }
```

```
    @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();  
    }
```

JwtAuthFilter – Interceptar requisição

- Criar uma classe JwtAuthFilter

@Component

@RequiredArgsConstructor

```
public class JwtAuthFilter extends OncePerRequestFilter {
```

```
    private final JwtService jwtService;
```

```
    private final UserDetailsService userDetailsService;
```

@Override

```
    protected void doFilterInternal(HttpServletRequest request,  
    HttpServletResponse response, FilterChain filterChain)
```

```
        throws ServletException, IOException {
```

```
        String authorization = request.getHeader("Authorization");
```

```
        if (authorization != null && authorization.startsWith("Bearer")) {
```

```
            String token = authorization.split(" ")[1];
```

```
            boolean isValid = jwtService.validarToken(token);
```

```
            if (isValid) {
```

```
                String loginUsuario = jwtService.obterLoginUsuario(token);
```

```
                UserDetails usuario =
```

```
                userDetailsService.loadUserByUsername(loginUsuario);
```

```
                UsernamePasswordAuthenticationToken user = new
```

```
                UsernamePasswordAuthenticationToken(usuario, null,
```

```
                usuario.getAuthorities());
```

```
                user.setDetails(new
```

```
                WebAuthenticationDetailsSource().buildDetails(request));
```

```
                SecurityContextHolder.getContext().setAuthentication(user);
```

```
            }
```

```
        }
```

```
        filterChain.doFilter(request, response);
```

```
    }
```

```
}
```


Spring Security e JWT

- Adicionar no pom.xml:

```
<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>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.11.5</version>
</dependency>
```
- spring-boot-starter-security: simplifica a configuração e o uso do Spring Security em aplicativos Spring Boot;
- jjwt-api: Esta é a API principal da biblioteca Java JWT. **Contém as interfaces e classes que definem o núcleo da funcionalidade JWT.** Geralmente, é necessário incluir essa dependência em seus projetos para usar as funcionalidades principais relacionadas a JWT, como **criação, verificação e manipulação de tokens.**
- jjwt-impl: Esta dependência fornece a **implementação concreta das interfaces definidas na API (jjwt-api).** Inclui as **classes que realizam as operações reais de manipulação de tokens.** Normalmente, esta dependência é adicionada como escopo de tempo de execução (runtime), pois é necessária apenas durante a execução do aplicativo.
- jjwt-jackson: Esta dependência **adiciona suporte para processamento de objetos JSON usando a biblioteca Jackson.** Jackson é uma biblioteca popular para trabalhar com JSON em Java. A inclusão desta dependência permite que você use objetos Java anotados com Jackson para representar e manipular os componentes do JWT de forma mais conveniente.

SecurityConfig

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
@EnableMethodSecurity

public class SecurityConfig {
    private final JwtAuthFilter jwtAuthfilter;
    private final AuthenticationProvider authenticationProvider;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(auth -> auth
                .requestMatchers(AntPathRequestMatcher.antMatcher("/api/auth/**"),
                    AntPathRequestMatcher.antMatcher("/h2-console/**"))
                .permitAll()
                .anyRequest().authenticated())
            .sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authenticationProvider(authenticationProvider)
            .addFilterBefore(jwtAuthfilter, UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }
}
```

Criar método de autenticação

- Criar DTO de autenticação:

@Data

@NoArgsConstructor

```
public class AutenticacaoDTO {  
    private String email;  
    private String senha;  
}
```

- Criar DTO de retorno da autenticação:

@Data

@NoArgsConstructor

@AllArgsConstructor

```
public class TokenDTO {  
    private String email;  
    private String token;  
}
```

Criar classe AuthController

- AuthController:

```
@RestController
@RequestMapping("/api/auth")
public class AutenticarController {
    private UsuarioService usuarioService;
    private JwtService jwtService;

    public AutenticarController(UsuarioService usuarioService, JwtService jwtService) {
        this.usuarioService = usuarioService;
        this.jwtService = jwtService;
    }
}
```

```
@PostMapping()
public TokenDTO autenticar(@RequestBody AutenticacaoDTO
autenticacao) {
    try {
        Usuario usuario = new Usuario(0, "",
autenticacao.getEmail(), autenticacao.getSenha(), "");

        UserDetails usuarioAutenticado =
usuarioService.autenticar(usuario);

        String token = jwtService.gerarToken(usuario);

        return new TokenDTO(usuario.getEmail(), token);

    } catch (UsernameNotFoundException |
RegraNegocioException ex) {

        throw new
ResponseStatusException(HttpStatus.UNAUTHORIZED,
ex.getMessage());

    }
}
```

Adicionar o Spring Security e JWT

- Adicionando esse starter a aplicação já está implementando a segurança.
- Se tentar executar os métodos com postman retornará o erro: ERROR 401 (Unauthorized).
- Se tentar executar pelo navegador abrirá uma tela de login.
 - Usuário: user
 - Senha: Será um token que aparecerá no console

Criar classe Usuario

```
@Entity
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor

public class Usuario implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String nome;

    private String email;

    private String senha;

    private String perfil;

    @Override

    public Collection<? extends GrantedAuthority> getAuthorities() {

        return null;
    }

    @Override

    public String getPassword() {

        return senha;
    }

    @Override

    public String getUsername() {

        return nome;
    }
}
```

```
@Override

    public boolean isAccountNonExpired() {

        return true;
    }

    @Override

    public boolean isAccountNonLocked() {

        return true;
    }

    @Override

    public boolean isCredentialsNonExpired() {

        return true;
    }

    @Override

    public boolean isEnabled() {

        return true;
    }
}
```

Criar classe UsuarioDTO

@Data

@Builder

@NoArgsConstructor

@AllArgsConstructor

public class UsuarioDTO {

private String nome;

private String email;

private String senha;

private String perfil;

}

UsuarioRepository

```
public interface UsuarioRepository extends JpaRepository<Usuario,  
Integer> {  
    Usuario findByEmail(String email);  
}
```


UsuarioService

```
public interface UsuarioService {  
    Usuario salvar(UsuarioDTO dto);  
  
    UsuarioDTO obterUsuarioPorId(Integer id);  
  
    List<UsuarioDTO> obterUsuarios();  
  
    UserDetails autenticar(Usuario usuario);  
}
```

UsuarioServiceImpl

@Service

@RequiredArgsConstructor

```
public class UsuarioServiceImpl implements UsuarioService, UserDetailsService {
```

```
    private final UsuarioRepository usuarioRepository;
```

```
    private final PasswordEncoder passwordEncoder;
```

@Override

@Transactional

```
public Usuario salvar(UsuarioDTO dto) {
```

```
    Usuario usuario = new Usuario();
```

```
    usuario.setEmail(dto.getEmail());
```

```
    usuario.setSenha(passwordEncoder.encode(dto.getSenha()));
```

```
    usuario.setPerfil(dto.getPerfil());
```

```
    return usuarioRepository.save(usuario);
```

```
}
```

@Override

```
public UsuarioDTO obterUsuarioPorId(Integer id) {
```

```
    return usuarioRepository.findById(id).map(u -> {
```

```
        return UsuarioDTO
```

```
            .builder()
```

```
            .email(u.getEmail())
```

```
            .perfil(u.getPerfil())
```

```
            .build();
```

```
    })
```

```
    .orElseThrow(() -> new RegraNegocioException("Usuário não encontrado"));
```

```
}
```

@Override

```
public List<UsuarioDTO> obterUsuarios() {
```

```
    List<UsuarioDTO> dados = usuarioRepository.findAll().stream().map(u -> {
```

```
        return UsuarioDTO
```

```
            .builder()
```

```
            .email(u.getEmail())
```

```
            .perfil(u.getPerfil())
```

```
            .build();
```

```
    }).toList();
```

```
    return dados;
```

```
}
```

UsuarioServiceImpl

@Override

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

```
    Usuario usuario =
    usuarioRepository.findByEmail(username);
```

```
    String[] roles = usuario.getPerfil() == "Administrador" ? new
String[] { "ADMIN", "USER" }
```

```
        : new String[] { "USER" };
```

```
    return User.builder()
```

```
        .username(usuario.getEmail())
```

```
        .password(usuario.getSenha())
```

```
        .roles(roles)
```

```
        .build();
```

```
}
```

@Override

```
public UserDetails autenticar(Usuario usuario) {
```

```
    UserDetails user =
loadUserByUsername(usuario.getEmail());
```

```
    boolean senhaOK =
passwordEncoder.matches(usuario.getSenha(),
user.getPassword());
```

```
    if (senhaOK) {
```

```
        return user;
```

```
    }
```

```
    throw new RegraNegocioException("Senha inválida");
```

```
}
```

Exercícios

1-) Exercício: Implementando Autenticação e Autorização com Spring Security

Cenário:

Você está desenvolvendo uma aplicação web de gerenciamento de tarefas. Existem dois perfis de usuários: **ADMIN** e **USER**.

- **ADMIN** pode acessar todas as rotas, incluindo a rota de administração (/admin).
- **USER** pode acessar apenas as rotas de usuário, como /tasks.

Exercício

Desafio:

Implemente segurança na aplicação usando **Spring Security**. As regras são:

1. Configure autenticação em memória com dois usuários:
 - Usuário: admin / Senha: admin123 / Role: ADMIN
 - Usuário: user / Senha: user123 / Role: USER
2. Defina as permissões:
 - A rota /admin deve ser acessível **apenas para ADMIN**.
 - A rota /tasks deve ser acessível para **USER e ADMIN**.
 - As demais rotas devem exigir autenticação.
3. Crie uma página de login personalizada.

Exercício

Estrutura sugerida:

•Controladores:

- /admin → retorna "Área de Administração"
- /tasks → retorna "Área de Tarefas"
- / → retorna "Bem-vindo"

•Classes principais:

- SecurityConfig → configuração do Spring Security.

2-) Implementação de Autenticação e Autorização com Spring Security e JWT.

Sua missão é construir uma API REST para gerenciar produtos. Essa API deve ser segura, utilizando **Spring Security com JWT** para autenticação e autorização.

Requisitos Funcionais:

1. Autenticação com JWT:

- Crie uma rota /auth/login que receba username e password.
- Caso as credenciais estejam corretas, gere e retorne um JWT contendo:
 - O username.
 - O role do usuário (ROLE_USER ou ROLE_ADMIN).
 - Validade do token (ex.: 1 hora).

2. Controle de Acesso:

- Existem dois perfis de usuários:
 - . **USER**: Pode listar e visualizar detalhes dos produtos.
 - . **ADMIN**: Pode listar, visualizar, cadastrar, atualizar e remover produtos.

3. Rotas da API:

- GET /products — Listar todos os produtos (**USER, ADMIN**).
- GET /products/{id} — Detalhar um produto (**USER, ADMIN**).
- POST /products — Criar novo produto (**ADMIN**).
- PUT /products/{id} — Atualizar um produto existente (**ADMIN**).
- DELETE /products/{id} — Excluir um produto (**ADMIN**).

Requisitos de Segurança

- Toda a API deve ser protegida com Spring Security.
- O login deve ser realizado via JWT, sem sessões (stateless).
- As rotas /auth/login e /auth/register (se implementada) devem ser públicas.
- Todas as demais rotas exigem token JWT válido no header Authorization no formato:

Authorization: Bearer <token>

- O token deve ser validado em todas as requisições protegidas.
- Se o token estiver inválido, expirado ou ausente, a API deve retornar HTTP 401 (Unauthorized).
- Se o usuário não tiver permissão suficiente, retorne HTTP 403 (Forbidden).

- Estrutura Sugerida do Projeto:

- Spring Boot
- Spring Security
- Spring Web
- Spring Data JPA (H2, PostgreSQL ou outro)
- Biblioteca JWT (jjwt ou io.jsonwebtoken)