

Guia para implementação JWT (API e Front)

Site: [São Paulo Tech School](#)
Curso: 2CC0A - Programação WEB - Backend 2025/1
Livro: Guia para implementação JWT (API e Front)

Impresso por: RAUL GOMES REIS .
Data: segunda-feira, 2 jun. 2025, 09:09

Índice

1. O que é JWT?

2. Configuração

2.1. Camadas

2.2. Entity e Repository

2.3. DTOs e Mapper

2.4. Service

2.5. Configurações Spring Security - Parte 1

2.6. Configurações Spring Security - Parte 2

2.7. Configurações Spring Security - Parte 3

2.8. Controller

2.9. Swagger

3. Demonstração

3.1. Requisição Sem Autenticação

3.2. Se Autenticando e Conferindo JWT

3.3. Requisição Com Autenticação

4. Bônus

1. O que é JWT?

JWT (JSON Web Token) é um padrão aberto definido pela especificação RFC 7519, utilizado para transmitir informações de forma segura entre clientes e servidores. É amplamente adotado em aplicações web e mobile para **autenticação e autorização de usuários**.

O token é **assinado digitalmente**, utilizando uma chave secreta com o algoritmo HMAC ou um par de chaves pública e privada (**RSA** ou **ECDSA**), garantindo a **integridade e autenticidade** dos dados transmitidos.

A principal função do JWT é permitir que o sistema identifique **quem é o usuário** que está acessando a aplicação e **quais recursos ele está autorizado a utilizar**, sem a necessidade de manter sessões ativas no servidor.

Um JWT é composto por **três partes**:

1. **Header (Cabeçalho)**: Contém informações sobre o tipo do token (JWT) e o algoritmo de assinatura;
2. **Payload (Corpo)**: Carrega os dados – chamados de claims – como o ID do usuário, nome, permissões e tempo de expiração;
3. **Signature (Assinatura)**: Garante que o token não foi alterado, validando sua integridade.

Pontos Importantes

- **Token translúcido:**

O JWT é considerado um token **translúcido**, o que significa que seu conteúdo pode ser lido por qualquer pessoa que o decodifique – embora não possa ser alterado sem invalidar a assinatura. Por isso, **evite armazenar dados sensíveis** (como senhas ou informações pessoais confidenciais).

Isso difere dos **tokens opacos**, cujos dados internos não são visíveis nem acessíveis, sendo úteis quando se deseja mais sigilo.

- **Token stateless (sem estado):**

O JWT é um token **stateless**, ou seja, ele **não mantém estado no servidor**. Toda requisição à um endpoint protegido precisa incluir o token (geralmente no cabeçalho de autorização). O servidor apenas valida o token recebido, sem precisar armazená-lo ou verificar sessões anteriores.


- **Tempo de expiração:**

Tokens JWT possuem um **tempo de expiração configurável**, definido no momento em que são gerados. Após o vencimento, o token deixa de ser válido para autenticação ou autorização, sendo necessário gerar um **novo token** para continuar acessando a aplicação de forma segura.

2. Configuração

Iremos criar um projeto básico para ilustrar a autenticação com Spring Security e JWT. Caso já possua um projeto criado, é possível seguir para o próximo tópico. Lembre-se de incluir as dependências necessárias.

(Se a visualização da imagem estiver ruim para leitura, abra ela em uma nova guia)

 **spring** initializr

Project

☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ Java ☐ Kotlin ☐ Groovy

☒ Maven

Spring Boot

☐ 3.5.0 (SNAPSHOT) ☐ 3.5.0 (M3) ☐ 3.4.5 (SNAPSHOT) ☒ 3.4.4

☐ 3.3.11 (SNAPSHOT) ☐ 3.3.10

Project Metadata

Group

school.sptech

Artifact

exemplo-jwt

Name

exemplo-jwt

Description

Demo project for Spring Boot

Package name

school.sptech.exemplojwt

Packaging

☒ Jar ☐ War

Java

☐ 24 ☒ 21 ☐ 17

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA

SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database

SQL

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Validation

I/O

Bean Validation with Hibernate validator.

Spring Security

SECURITY

Highly customizable authentication and access-control framework for Spring applications.

Spring Boot Actuator

OPS

Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.

Dependências Adicionais:

Algumas dependências que iremos utilizar no projeto não estão presentes no spring inicializr, por isso precisamos colocá-las manualmente, sendo elas:

- **Dependência do Swagger:**

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.5.0</version>
</dependency>
```

- Dependências do JWT:

```
<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>
  <scope>runtime</scope>
</dependency>
```

Configuração do application.properties:

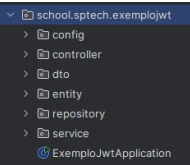
```
server.error.include-message=always
server.error.include-binding-errors=always

spring.datasource.url=jdbc:h2:mem:teste-security
spring.h2.console.enabled=true
spring.datasource.username=admin
spring.datasource.password=admin

spring.jpa.defer-datasource-initialization=true
```

2.1. Camadas

Para organizarmos melhor o nosso código, vamos criar os seguintes pacotes:



- **Pacote config:** terá todas as classes de configuração do nosso projeto;
- **Pacote controller:** terá todos os nossos endpoints para testarmos a aplicação;
- **Pacote dto:** terá as classes destinadas a transferência de dados na aplicação;
- **Pacote entity:** terá classe que representa nossa entidade no banco;
- **Pacote repository:** terá nossa classe que nos permite realizar queries no banco de dados;
- **Pacote service:** terá as classes que contém toda nossa regra de negócio.

2.2. Entity e Repository

No pacote **entity** crie uma classe **Usuario** que será a entidade que iremos utilizar na aplicação:

```
package school.sptech.exemplojwt.entity;

import ...

@Entity 24 usages 1 relsrb
public class Usuario {

    @Id 2 usages
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nome; 2 usages
    private String email; 2 usages
    private String senha; 2 usages

    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }

    public String getNome() { return nome; }

    public void setNome(String nome) { this.nome = nome; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getSenha() { return senha; }

    public void setSenha(String senha) { this.senha = senha; }
}
```

No pacote **repository** crie a interface **UsuarioRepository** que terá um método de consulta personalizado que irá retornar um **Optional<Usuario>** ao tentar buscar um usuário cadastrado banco com um determinado e-mail:

```
@Repository 4 usages 1 Diego Brito
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

    Optional<Usuario> findByEmail(String email); 2 usages 1 Diego Brito
}
```

2.3. DTOs e Mapper

No pacote de **dto** crie as seguintes classes:

- **UsuarioCriacaoDto** - que terá os atributos necessários para cadastrar um usuário na nossa aplicação, sendo que iremos usar algumas anotações de validações para esses atributos:

```
public class UsuarioCriacaoDto { 4 usages  ▲ reisirb +2

    @Size(min = 3, max = 10) 2 usages
    @Schema(description = "Nome do usuário", example = "John Doe")
    private String nome;

    @Email 2 usages
    @Schema(description = "Email do usuário", example = "john@doe.com")
    private String email;

    @Size(min = 6, max = 20) 2 usages
    @Schema(description = "Senha do usuário", example = "123456")
    private String senha;

    public String getNome() { return nome; }

    public void setNome(String nome) { this.nome = nome; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getSenha() { return senha; }

    public void setSenha(String senha) { this.senha = senha; }
}
```

- **UsuarioListarDto** - que terá os atributos que queremos exibir quando formos listar nossos usuários cadastrados:

```
public class UsuarioListarDto { 8 usages  ▲ henriquePiassi

    @Schema(description = "Id do usuário", example = "1") 2 usages
    private Long id;

    @Schema(description = "Nome do usuário", example = "John Doe") 2 usages
    private String nome;

    @Schema(description = "Email do usuário", example = "john@doe.com")
    private String email;

    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }

    public String getNome() { return nome; }

    public void setNome(String nome) { this.nome = nome; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }
}
```

- **UsuarioLoginDto** - que terá os atributos necessários para que o usuário possa fazer o login na nossa aplicação:

```
public class UsuarioLoginDto { 4 usages  ▲ reisirb +1

    @Schema(description = "E-mail do usuário", example = "john@doe.com")
    private String email;
    @Schema(description = "Senha do usuário", example = "123456") 2 usages
    private String senha;

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getSenha() { return senha; }

    public void setSenha(String senha) { this.senha = senha; }
}
```

- **UsuarioTokenDto** - que terá os atributos que queremos retornar assim que o usuário for autenticado por nossa aplicação (o atributo token será justamente aonde iremos retornar o JWT mais para frente):

```
public class UsuarioTokenDto { 8 usages  ↗ reirrb

    private Long userId; 2 usages
    private String nome; 2 usages
    private String email; 2 usages
    private String token; 2 usages

    public Long getUserId() { return userId; }

    public void setUserId(Long userId) { this.userId = userId; }

    public String getNome() { return nome; }

    public void setNome(String nome) { this.nome = nome; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getToken() { return token; }

    public void setToken(String token) { this.token = token; }
}
```

- **UsuarioDetalhesDto** - essa classe será um pouco diferente das outras, porque nela iremos implementar uma interface do **Spring Security** chamada **UserDetails** que será usada posteriormente na nossa configuração de segurança:
 - Primeiro iremos definir seus atributos, construtor e getter:

```
public class UsuarioDetalhesDto implements UserDetails {

    private final String nome; 2 usages

    private final String email; 2 usages

    private final String senha; 2 usages

    public UsuarioDetalhesDto(Usuario usuario) { 1 usage  ↗ reirrb
        this.nome = usuario.getNome();
        this.email = usuario.getEmail();
        this.senha = usuario.getSenha();
    }

    public String getNome() { return nome; }
```

- Agora iremos reescrever alguns métodos da interface **UserDetails** (caso queira trabalhar com nível de autorização de um usuário, você vai precisar fazer uma coleção, uma lista por exemplo, das autorizações disponíveis no seu sistema e utilizar o método `getAuthorities()`):

```
@Override 2 usages  ↗ reirrb
public Collection< extends GrantedAuthority> getAuthorities() { return null; }

@Override  ↗ reirrb
public String getPassword() { return senha; }

@Override  ↗ reirrb
public String getUsername() { return email; }

@Override no usages  ↗ reirrb
public boolean isAccountNonExpired() { return true; }

@Override no usages  ↗ reirrb
public boolean isAccountNonLocked() { return true; }

@Override no usages  ↗ reirrb
public boolean isCredentialsNonExpired() { return true; }

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

No mesmo pacote de **dto** crie a classe **UsuarioMapper** que utiliza o padrão de projeto **Adapter** para mapear os dados de um tipo de objeto para outro. Nela terá os seguintes métodos:

- Um método que irá mapear o **UsuarioCriacaoDto** para a entidade **Usuario**:

```
public class UsuarioMapper { 6 usages  ↗ reirrb +1

    public static Usuario of(UsuarioCriacaoDto usuarioCriacaoDto) {
        Usuario usuario = new Usuario();

        usuario.setEmail(usuarioCriacaoDto.getEmail());
        usuario.setNome(usuarioCriacaoDto.getNome());
        usuario.setSenha(usuarioCriacaoDto.getSenha());

        return usuario;
    }
}
```

- Um método que irá mapear o **UsuarioLoginDto** para a entidade **Usuario**:

```
public static Usuario of(UsuarioLoginDto usuarioLoginDto) {
    Usuario usuario = new Usuario();

    usuario.setEmail(usuarioLoginDto.getEmail());
    usuario.setSenha(usuarioLoginDto.getSenha());

    return usuario;
}
```

- Um método que irá mapear a entidade **Usuario** para o **UsuarioTokenDto**, além de também colocar o valor do **token** no dto:


```
public static UsuarioTokenDto of(Usuario usuario, String token) {  
    UsuarioTokenDto usuarioTokenDto = new UsuarioTokenDto();  
  
    usuarioTokenDto.setUserId(usuario.getId());  
    usuarioTokenDto.setEmail(usuario.getEmail());  
    usuarioTokenDto.setNome(usuario.getNome());  
    usuarioTokenDto.setToken(token);  
  
    return usuarioTokenDto;  
}
```

- Um método que irá mapear a entidade **Usuario** para o **UsuarioListarDto**:

```
public static UsuarioListarDto of(Usuario usuario) { 1 usage  
    UsuarioListarDto usuarioListarDto = new UsuarioListarDto();  
  
    usuarioListarDto.setId(usuario.getId());  
    usuarioListarDto.setEmail(usuario.getEmail());  
    usuarioListarDto.setNome(usuario.getNome());  
  
    return usuarioListarDto;  
}
```

2.4. Service

No pacote **service** crie a classe **UsuarioService** que conterá toda a regra de negócio relacionada a entidade **Usuario**, para isso:

- Primeiro vamos definir a anotação da classe e suas dependências (a classe **GerenciadorTokenJwt** ainda será criada, não se preocupe com o erro que a IDE indicar por ora):

```
@Service 2 usages 1 reisrb +1
public class UsuarioService {

    @Autowired 1 usage
    private PasswordEncoder passwordEncoder;

    @Autowired 3 usages
    private UsuarioRepository usuarioRepository;

    @Autowired 1 usage
    private GerenciadorTokenJwt gerenciadorTokenJwt;

    @Autowired 1 usage
    private AuthenticationManager authenticationManager;
```

- Primeiro método da classe será para cadastrar o usuário no banco de dados, mas nele iremos utilizar o método **encode** do **PasswordEncoder** para **criptografar a senha do usuário**, fazendo com que qualquer um que acesse o banco nunca saberá qual a senha de determinado usuário, assim **protegendo um dado muito sensível**:

```
public void criar(Usuario novoUsuario) { 1 usage 1 reisrb +1

    String senhaCriptografada = passwordEncoder.encode(novoUsuario.getSenha());
    novoUsuario.setSenha(senhaCriptografada);

    this.usuarioRepository.save(novoUsuario);
}
```

- Segundo método será para realizar a autenticação do usuário, utilizando a classe **AuthenticationManager** para verificar as credenciais, depois buscar no banco o usuário pelo seu e-mail, definindo no contexto do Spring Security a autenticação desse usuário e por fim, utiliza ela para gerar o token JWT (com a classe que ainda iremos criar):

```
public UsuarioTokenDto autenticar(Usuario usuario) { 1 usage 1 reisrb +1

    final UsernamePasswordAuthenticationToken credentials = new UsernamePasswordAuthenticationToken(
        usuario.getEmail(), usuario.getSenha());

    final Authentication authentication = this.authenticationManager.authenticate(credentials);

    Usuario usuarioAutenticado =
        usuarioRepository.findByEmail(usuario.getEmail())
            .orElseThrow(
                () -> new RuntimeException(404, "Email do usuário não cadastrado", null)
            );

    SecurityContextHolder.getContext().setAuthentication(authentication);

    final String token = gerenciadorTokenJwt.generateToken(authentication);

    return UsuarioMapper.of(usuarioAutenticado, token);
}
```

- Por fim iremos criar um método que irá listar todos os usuários cadastrados no banco, mapeando eles para o DTO que criamos anteriormente:

```
public List<UsuarioListarDto> listarTodos() { 1 usage 1 HenriquePiasini

    List<Usuario> usuariosEncontrados = usuarioRepository.findAll();
    return usuariosEncontrados.stream().map(UsuarioMapper::of).toList();
}
```

Ainda no pacote **service** crie a classe **AutenticacaoService**, que irá implementar a interface **UserDetailsService** do **Spring Security**, nela iremos implementar o método **loadUserByUsername** que recebe o username do usuário (que no nosso caso é o e-mail dele) e com esse dado iremos procurar ele no banco, retornando que ele não foi encontrado ou o DTO **UsuarioDetalhesDto** que criamos anteriormente:

```
@Service 3 usages 1 reisrb
public class AutenticacaoService implements UserDetailsService {

    @Autowired 1 usage
    private UsuarioRepository usuarioRepository;

    // Método da interface implementada
    @Override 2 usages 1 reisrb
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        Optional<Usuario> usuarioOpt = usuarioRepository.findByEmail(username);

        if (usuarioOpt.isEmpty()) {

            throw new UsernameNotFoundException(String.format("usuário: %s não encontrado", username));
        }

        return new UsuarioDetalhesDto(usuarioOpt.get());
    }
}
```

2.5. Configurações Spring Security - Parte 1

Agora iremos começar a criar nossas classes de configurações do **Spring Security** e para o **JWT**, criaremos todas elas no pacote **config**:

- Crie a classe **AutenticacaoEntryPoint** que irá implementar a interface **AuthenticationEntryPoint**, o método que iremos implementar serve para definir como a aplicação vai reagir com as requisições não autenticadas, ou seja, quando o usuário tentar acessar um endpoint protegido sem estar autenticado:

```
@Component 3 usages 1 reisrb +2
public class AutenticacaoEntryPoint implements AuthenticationEntryPoint {

    @Override no usages 1 Diego Brito +2
    public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException) throws IOException, ServletException {
        if (authException.getClass().equals(BadCredentialsException.class) || authException.getClass().equals(InsufficientAuthenticationException.class)) {
            response.sendError(HttpServletResponse.SC_UNAUTHORIZED);
        } else {
            response.sendError(HttpServletResponse.SC_FORBIDDEN);
        }
    }
}
```

- Crie a classe **AutenticacaoProvider** que irá implementar a interface **AuthenticationProvider**, os métodos que iremos implementar servem para definir a lógica de autenticação do usuário, sendo responsável por processar as informações fornecidas e verificar se as credenciais são válidas:

```
public class AutenticacaoProvider implements AuthenticationProvider { 1 usage 1 reisrb

    private final AutenticacaoService usuarioAutorizacaoService; 2 usages
    private final PasswordEncoder passwordEncoder; 2 usages

    public AutenticacaoProvider(AutenticacaoService usuarioAutorizacaoService, PasswordEncoder passwordEncoder) { 1 usage
        this.usuarioAutorizacaoService = usuarioAutorizacaoService;
        this.passwordEncoder = passwordEncoder;
    }

    @Override 1 reisrb
    public Authentication authenticate(final Authentication authentication) throws AuthenticationException {

        final String username = authentication.getName();
        final String password = authentication.getCredentials().toString();

        UserDetails userDetails = this.usuarioAutorizacaoService.loadUserByUsername(username);

        if (this.passwordEncoder.matches(password, userDetails.getPassword())) {
            return new UsernamePasswordAuthenticationToken(userDetails, credentials: null, userDetails.getAuthorities());
        } else {
            throw new BadCredentialsException("Usuário ou Senha inválidos");
        }
    }

    @Override 1 reisrb
    public boolean supports(final Class<?> authentication) {
        return authentication.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

2.6. Configurações Spring Security - Parte 2

Continuando no pacote **config**:

- Crie a classe **GerenciadorTokenJwt** que vai ser a responsável por gerar e validar os tokens JWT:
 - Os métodos da classe irão codificar as informações do usuário em uma sequência de caracteres criptografados e adicionar uma assinatura digital para garantir que o token não tenha sido alterado durante a transmissão:

```
public class GerenciadorTokenJwt { 6 usages 1 relsrb +1

    @Value("${jwt.secret}") 1 usage
    private String secret;

    @Value("${jwt.validity}") 1 usage
    private long jwtTokenValidity;

    public String getUsernameFromToken(String token) { return getClaimForToken(token, Claims::getSubject); }

    public Date getExpirationDateFromToken(String token) { return getClaimForToken(token, Claims::getExpiration); }

    public String generateToken(final Authentication authentication) { 1 usage 1 relsrb +1

        // Para verificacoes de permissões;
        final String authorities = authentication.getAuthorities().stream().map(GrantedAuthority::getAuthority)
            .collect(Collectors.joining(" ", ""));

        return Jwts.builder().setSubject(authentication.getName())
            .signWith(parseSecret()).setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + jwtTokenValidity * 1_000)).compact();
    }

    public <T> T getClaimForToken(String token, Function<Claims, T> claimsResolver) { 2 usages 1 relsrb
        Claims claims = getAllClaimsFromToken(token);
        return claimsResolver.apply(claims);
    }
}
```

```
public boolean validateToken(String token, UserDetails userDetails) { 1 usage 1 relsrb
    String username = getUsernameFromToken(token);
    return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
}

private boolean isTokenExpired(String token) { 1 usage 1 relsrb
    Date expirationDate = getExpirationDateFromToken(token);
    return expirationDate.before(new Date(System.currentTimeMillis()));
}

private Claims getAllClaimsFromToken(String token) { 1 usage 1 Diego Brito +1
    return Jwts.parserBuilder().jwtParserBuilder()
        .setSigningKey(parseSecret())
        .build().jwtParser()
        .parseClaimsJws(token).getBody();
}

private SecretKey parseSecret() { return Keys.hmacShaKeyFor(this.secret.getBytes(StandardCharsets.UTF_8)); }
}
```

- Nos atributos dessa classe usamos a anotação **@Value** própria do Spring, na String que colocamos nela estamos falando para o Spring buscar no **application.properties** os valores que definimos no caminho especificado e assim alocar eles nos atributos, mas atualmente nosso **applicaton.properties** não tem esses caminhos e valores definidos, então vamos fazer isso:

```
# tempo de expiração do token em milissegundos (esse valor indica que ele vai expirar em 1 hora)
jwt.validity=3600000

# palavra passe do token (segredo) necessita de no mínimo 32 caracteres e serve para assinar e verificar os tokens, sendo recomendável usar uma chave complexa e difícil de adivinhar para aumentar a
segurança da aplicação
jwt.secret=RXhpc3RlIHVtYSB6ZW9yaWdgcXVlIGRpeiBxdWUsIHNIHVtIGRpySBhbgd1Gw9ZGVzY291cmlyIGV4YXRobWudGUGcGFyYSBxdWUgc2VydmUgbyBvbml2ZXJzbyB1IHhvc1BxdWUgZWxlIGVzd0EgYXF1aSwgZWxlIGRlc2FwYXJlY2VY4SBpbm90YW50
```

- Crie a classe **AutenticacaoFilter**, que vai estender a classe abstrata **OncePerRequestFilter**, sendo responsável por processar as solicitações de autenticação do usuário e realizar a validação das credenciais fornecidas pelo usuário, então para cada solicitação HTTP recebida ela extrai o token do cabeçalho da requisição, checa se está expirado ou se é válido e, se passar nessa validação, identifica quem é o usuário e autoriza o acesso dele na aplicação:

```
public class AutenticacaoFilter extends OncePerRequestFilter { 3 usages 1 relsrb

    private static final Logger LOGGER = LoggerFactory.getLogger(AutenticacaoFilter.class); 2 usages

    private final AutenticacaoService autenticacaoService; 2 usages

    private final GerenciadorTokenJwt jwtTokenManager; 3 usages

    public AutenticacaoFilter(AutenticacaoService autenticacaoService, GerenciadorTokenJwt jwtTokenManager) {
        this.autenticacaoService = autenticacaoService;
        this.jwtTokenManager = jwtTokenManager;
    }
}
```

```
@Override no usages 1 reirsb
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
    String username = null;
    String jwtToken = null;

    String requestTokenHeader = request.getHeader("Authorization");

    if (Objects.nonNull(requestTokenHeader) && requestTokenHeader.startsWith("Bearer ")) {
        jwtToken = requestTokenHeader.substring(beginIndex: 7);

        try {
            username = jwtTokenManager.getUsernameFromToken(jwtToken);
        } catch (ExpiredJwtException exception) {

            LOGGER.info("[FALHA AUTENTICACAO] - Token expirado, usuario: {} - {}",
                exception.getClaims().getSubject(), exception.getMessage());

            LOGGER.trace("[FALHA AUTENTICACAO] - stack trace: %s", exception);

            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        }
    }

    if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
        addUsernameInContext(request, username, jwtToken);
    }

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

```
private void addUsernameInContext(HttpServletRequest request, String username, String jwtToken) { 1 usage 1 reirsb

    UserDetails userDetails = autenticacaoService.loadUserByUsername(username);

    if (jwtTokenManager.validateToken(jwtToken, userDetails)) {

        UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(
            userDetails, credentials: null, userDetails.getAuthorities());

        usernamePasswordAuthenticationToken
            .setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

        SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
    }
}
```

2.7. Configurações Spring Security - Parte 3

Ainda no pacote `config` crie a classe `SecurityConfiguracao`, ela vai ser responsável por proteger as rotas da aplicação, controlando **quem pode acessar o quê**. Ela define quais endpoints estão **abertos ao público** — como o login, a [documentação Swagger](#) e o console do banco H2 — e quais precisam de **autenticação** com um token JWT válido.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfiguracao {

    @Autowired
    private AutenticacaoService autenticacaoService;

    @Autowired
    private AutenticacaoEntryPoint autenticacaoJwtEntryPoint;

    private static final AntPathRequestMatcher[] URLS_PERMITIDAS = {
        new AntPathRequestMatcher(pattern: "/swagger-ui/**"),
        new AntPathRequestMatcher(pattern: "/swagger-ui.html"),
        new AntPathRequestMatcher(pattern: "/swagger-resources"),
        new AntPathRequestMatcher(pattern: "/swagger-resources/**"),
        new AntPathRequestMatcher(pattern: "/configuration/ui"),
        new AntPathRequestMatcher(pattern: "/configuration/security"),
        new AntPathRequestMatcher(pattern: "/api/public/**"),
        new AntPathRequestMatcher(pattern: "/api/public/authenticate"),
        new AntPathRequestMatcher(pattern: "/webjars/**"),
        new AntPathRequestMatcher(pattern: "/v3/api-docs/**"),
        new AntPathRequestMatcher(pattern: "/actuator/*"),
        new AntPathRequestMatcher(pattern: "/usuarios/login/**"),
        new AntPathRequestMatcher(pattern: "/h2-console/**"),
        new AntPathRequestMatcher(pattern: "/h2-console/**/**"),
        new AntPathRequestMatcher(pattern: "/error/**")
    };
};
```

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .headers(HeadersConfigurer<HttpSecurity> headers -> headers
            .frameOptions(HeadersConfigurer.FrameOptionsConfig::disable))
        .cors(Customizer.withDefaults())
        .csrf(CsrfConfigurer<HttpSecurity>::disable)
        .authorizeHttpRequests(AuthorizationManagerRequestMat... authorize -> authorize.requestMatchers(URLS_PERMITIDAS).AuthorizedUri
            .permitAll() AuthorizationManagerRequestMat...
            .anyRequest().AuthorizedUri
            .authenticated()
        )
        .exceptionHandling(ExceptionHandlingConfigurer<HttpSecurity> handling -> handling
            .authenticationEntryPoint(autenticacaoJwtEntryPoint))
        .sessionManagement(SessionManagementConfigurer<HttpSecurity> management -> management
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    http.addFilterBefore(jwtAuthenticationFilterBean(), UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

```
@Bean
public AuthenticationManager authManager(HttpSecurity http) throws Exception {
    AuthenticationManagerBuilder authenticationManagerBuilder =
        http.getSharedObject(AuthenticationManagerBuilder.class);
    authenticationManagerBuilder.authenticationProvider(new AutenticacaoProvider(autenticacaoService, passwordEncoder()));
    return authenticationManagerBuilder.build();
}

@Bean
public AutenticacaoEntryPoint jwtAuthenticationEntryPointBean() { return new AutenticacaoEntryPoint(); }

@Bean
public AutenticacaoFilter jwtAuthenticationFilterBean() {
    return new AutenticacaoFilter(autenticacaoService, jwtAuthenticationUtilBean());
}

@Bean
public GerenciadorTokenJwt jwtAuthenticationUtilBean() { return new GerenciadorTokenJwt(); }

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

```
@Bean no usages ▲ Rafael Reis +1
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuracao = new CorsConfiguration();
    configuracao.applyPermitDefaultValues();
    configuracao.setAllowedMethods(
        Arrays.asList(
            HttpMethod.GET.name(),
            HttpMethod.POST.name(),
            HttpMethod.PUT.name(),
            HttpMethod.PATCH.name(),
            HttpMethod.DELETE.name(),
            HttpMethod.OPTIONS.name(),
            HttpMethod.HEAD.name(),
            HttpMethod.TRACE.name());

    configuracao.setExposedHeaders(List.of(HttpHeaders.CONTENT_DISPOSITION));

    UrlBasedCorsConfigurationSource origem = new UrlBasedCorsConfigurationSource();
    origem.registerCorsConfiguration(pattern: "**", configuracao);

    return origem;
}
```

O método **filterChain** é o centro da configuração. Nele, são definidos:

- O **CORS** (que permite que outras origens façam requisições para a API);
- A desativação do **CSRF** (proteção contra ataques de falsificação de requisições);
- A **autorização** para que apenas usuários autenticados possam acessar certos recursos;
- A **política de sessão do servidor** como **stateless** (tópico explicado no primeiro capítulo).

Nessa classe também temos todas as nossas classes de configurações sendo definidas no contexto do Spring, com cada uma ficando responsável por etapas importantes da autenticação, conforme visto anteriormente.

Tudo isso junto forma a camada de segurança da aplicação, garantindo que apenas usuários autorizados possam acessar informações protegidas.

2.8. Controller

No pacote **controller** crie a classe **UsuarioController** onde iremos definir os endpoints da nossa aplicação para poder testá-la, os endpoints serão:

- Um **POST** para cadastrar o usuário (precisa estar autenticado):

```
@RestController no usages 1 henriquePiassi
@RequestMapping("/usuarios")
public class UsuarioController {

    @Autowired 3 usages
    private UsuarioService usuarioService;

    @PostMapping no usages 1 henriquePiassi
    @SecurityRequirement(name = "Bearer")
    public ResponseEntity<Void> criar(@RequestBody @Valid UsuarioCriacaoDto usuarioCriacaoDto) {

        final Usuario novoUsuario = UsuarioMapper.of(usuarioCriacaoDto);
        this.usuarioService.criar(novoUsuario);
        return ResponseEntity.status(201).build();
    }
}
```

- Um **POST** para o usuário poder realizar o login (é liberado para o público, apenas lembre-se de cadastrar um usuário no banco via insert para fazer o login pela primeira vez):

```
@PostMapping("/login") no usages 1 henriquePiassi
public ResponseEntity<UsuarioTokenDto> login(@RequestBody UsuarioLoginDto usuarioLoginDto) {

    final Usuario usuario = UsuarioMapper.of(usuarioLoginDto);
    UsuarioTokenDto usuarioTokenDto = this.usuarioService.autenticar(usuario);

    return ResponseEntity.status(200).body(usuarioTokenDto);
}
```

- Um **GET** para listar todos os usuários cadastrados (precisa estar autenticado):

```
@GetMapping no usages 1 henriquePiassi
@SecurityRequirement(name = "Bearer")
public ResponseEntity<List<UsuarioListarDto>> listarTodos() {

    List<UsuarioListarDto> usuariosEncontrados = this.usuarioService.listarTodos();

    if (usuariosEncontrados.isEmpty()){
        return ResponseEntity.status(204).build();
    }
    return ResponseEntity.status(200).body(usuariosEncontrados);
}
```



3. Demonstração

Agora vamos testar nossa aplicação e ver como funciona a validação do JWT na prática.

Para isso, acesse a página **HTML do Swagger** da aplicação, onde terá listado todos os endpoints da aplicação e poderemos fazer requisições neles por lá.

A URL de acesso é: **http://localhost:8080/swagger-ui/index.html#/**

É esperado a tela aparecer assim:

 **Swagger**
Supports OpenAPI 3.0

/v3/api-docs

Explore

Projeto Usuários

1.0.0OAS 3.0

/v3/api-docs

Exemplo de implementação de JWT com Spring Security

[Diego - Website](#)
[Send email to Diego](#)
UNLICENSED

Servers

http://localhost:8080 - Generated server url

Authorize

usuario-controller

GET /usuarios

POST /usuarios

POST /usuarios/login

- Atente-se:**
- Em várias das classes criadas para esse projeto existe anotações do Swagger nelas, caso o seu Swagger esteja muito diferente do print acima ou dos próximos prints, revise os prints dos códigos da aplicação para conferir se a documentação está correta!
 - Na nossa aplicação o cadastro de usuário precisa do JWT para ser executado, então caso não tenha nenhum usuário já cadastrado no seu banco para realizar o login, faça isso via insert:
 - URL para acessar o console do H2: localhost:8080/h2-console (lembre-se que no **application.properties** foi configurado para o usuário e a senha do H2 ser admin).

3.1. Requisição Sem Autenticação

Primeiro vamos descobrir o que acontece quando fazemos uma requisição de um endpoint protegido sem estar autenticado. Para isso, execute o endpoint de GET de `/usuarios` para tentar listar os usuários cadastrados:

usuario-controller

GET /usuarios

Try it out

Parameters

No parameters

Responses

| Code | Description | Links |
|------|---|----------|
| 200 | OK <div>Media type */*</div> <div>Controls Accept header.</div> <div>Example Value Schema</div> <div><pre>[{ "id": 1, "nome": "John Doe", "email": "john@doe.com" }]</pre></div> | No links |

GET /usuarios

Cancel

Parameters

No parameters

Execute

Responses

| Code | Description | Links |
|------|---|----------|
| 200 | OK <div>Media type */*</div> <div>Controls Accept header.</div> <div>Example Value Schema</div> <div><pre>[{ "id": 1, "nome": "John Doe", "email": "john@doe.com" }]</pre></div> | No links |

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:8080/usuarios' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8080/usuarios
```

Server response

| Code | Details |
|------|-------------------------------|
| 401 | Error: response status is 401 |

Response body

```
{
  "timestamp": "2025-04-11T12:48:45.775+00:00",
  "status": 401,
  "error": "Unauthorized",
  "message": "No message available",
  "path": "/usuarios"
}
```

Response headers

```
cache-control: no-cache,no-store,max-age=0,must-revalidate
connection: keep-alive
content-type: application/json
date: Fri, 11 Apr 2025 12:48:45 GMT
expires: 0
keep-alive: timeout=60
pragma: no-cache
transfer-encoding: chunked
vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
x-content-type-options: nosniff
x-xss-protection: 0
```

Responses

A resposta que o endpoint retornará por causa do usuário não estar autenticado será **401Unauthorized** (Não autorizado).

3.2. Se Autenticando e Conferindo JWT

Agora vamos nos logar na aplicação e pegar o token de acesso JWT, para assim podermos nos autenticar e executar os outros endpoints:

- No endpoint POST `/usuarios/login` realize o login passando no body da requisição suas credenciais (do usuário cadastrado previamente no banco):

POST

/usuarios/login

Parameters

No parameters

Request body required

application/json

```
{
  "email": "john@doe.com",
  "senha": "123456"
}
```

Execute

Curl

```
curl -X 'POST' \
  'http://localhost:8080/usuarios/login' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "email": "john@doe.com",
    "senha": "123456"
  }'
```

Request URL

http://localhost:8080/usuarios/login

Server response

| Code | Details |
|------|---|
| 200 | <div><div>Response body</div><div><pre>{ "userId": 1, "nome": "John Doe", "email": "john@doe.com", "token": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJqb2huQGRvZS5jb20iLCJpYXQiOiJlNDQzNzYxNTYsImV4cCI6MTc0Nzk3NjE1Nm0uZm90Ym90Ym90PQSNuU35_X39GM00Qp6sCLBAX0YBjNeij1WBQoDwh_4khknni15h9uY1B0XTF6dQ_WKcFBQ56Q" }</pre></div><div>Download</div></div> <div><div>Response headers</div><div><pre>cache-control: no-cache,no-store,max-age=0,must-revalidate connection: keep-alive content-type: application/json date: Fri, 11 Apr 2025 12:55:56 GMT expires: 0 keep-alive: timeout=60 pragma: no-cache transfer-encoding: chunked vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers x-content-type-options: nosniff x-xss-protection: 0</pre></div></div> |

- Na resposta do endpoint terá os atributos que estabelecemos no DTO da nossa aplicação, sendo que o atributo `"token"` é o valor do **token JWT**:

Server response

| Code | Details |
|------|--|
| 200 | <div><div>Response body</div><div><pre>{ "userId": 1, "nome": "John Doe", "email": "john@doe.com", "token": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJqb2huQGRvZS5jb20iLCJpYXQiOiJlNDQzNzYxNTYsImV4cCI6MTc0Nzk3NjE1Nm0uZm90Ym90Ym90PQSNuU35_X39GM00Qp6sCLBAX0YBjNeij1WBQoDwh_4khknni15h9uY1B0XTF6dQ_WKcFBQ56Q" }</pre></div><div>Download</div></div> |

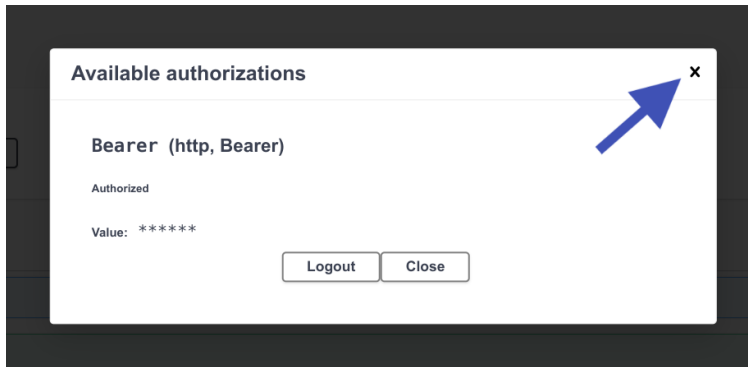
- Entre no site <https://jwt.io/> e cole nele o valor do JWT que retornou no login, como o token é **translúcido**, nesse site é possível ver todas as informações presentes nele:

- Mas... ele vai dar erro de **assinatura inválida**. Isso ocorre porque o site inicia com uma chave secreta genérica, para corrigir isso, copie a chave do **application.properties** e cole no site, porque assim ele vai validar se a chave secreta bate com a criptografia do seu token e claro, o site também valida se o JWT está em um formato válido:



- Agora que temos certeza que o JWT está totalmente correto, vamos salvar seu valor no Swagger para ele nos autenticar na chamada dos outros endpoints:





Com esse passo a passo você estará autenticado no Swagger e agora vai poder executar os outros endpoints com sucesso.

3.3. Requisição Com Autenticação

Com sua autenticação realizada, execute os outros endpoints e veja se todos irão voltar os status code de sucesso:

- Endpoint de POST /usuarios:

POST /usuarios

Parameters

No parameters

Request body required

application/json

```
{
  "nome": "Diego Lima",
  "email": "diego@email.com",
  "senha": "123456"
}
```

Execute

Clear

Responses

Curl

```
curl -X POST \
  http://localhost:8080/usuarios \
  -H 'accept: */*' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzIuOiJpZ2h0dGRvZS5jb291LCJpYXQiOiJlbnR0QzRzYnNTYsImV4cCI6MTc0MzIzNjE1In0.foH0nyUtyfa0PQ5naUJ5_X39CM00p6sCLBAX0YB3neiJlMBQoDwb_4khkgjI5t9' \
  -d '{
    "nome": "Diego Lima",
    "email": "diego@email.com",
    "senha": "123456"
  }'
```

Request URL

http://localhost:8080/usuarios

Server response

Code

Details

201

Undocumented

Response headers

```
cache-control: no-cache,no-store,max-age=0,must-revalidate
connection: keep-alive
content-length: 0
date: Fri, 11 Apr 2025 13:31:42 GMT
expires: 0
keep-alive: timeout=60
pragma: no-cache
vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
x-content-type-options: nosniff
x-xss-protection: 0
```

- Endpoint de GET /usuarios:

GET /usuarios

Parameters

No parameters

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  http://localhost:8080/usuarios \
  -H 'accept: */*' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzIuOiJpZ2h0dGRvZS5jb291LCJpYXQiOiJlbnR0QzRzYnNTYsImV4cCI6MTc0MzIzNjE1In0.foH0nyUtyfa0PQ5naUJ5_X39CM00p6sCLBAX0YB3neiJlMBQoDwb_4khkgjI5t9'
```

Request URL

http://localhost:8080/usuarios

Server response

Code

Details

200

Response body

```
{
  "id": 1,
  "nome": "John Doe",
  "email": "john@doe.com"
},
{
  "id": 2,
  "nome": "Diego Lima",
  "email": "diego@email.com"
}
```

Response headers

```
cache-control: no-cache,no-store,max-age=0,must-revalidate
content-type: application/json
date: Fri, 11 Apr 2025 13:34:29 GMT
expires: 0
keep-alive: timeout=60
pragma: no-cache
transfer-encoding: chunked
vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
x-content-type-options: nosniff
x-xss-protection: 0
```

LEMBRE-SE:

O token possui um tempo de expiração (que na nossa aplicação está configurado para durar 1 hora), após a validade dele passar, é necessário **gerar um novo token**, se não esse token expirado não vai conseguir te autenticar no sistema.

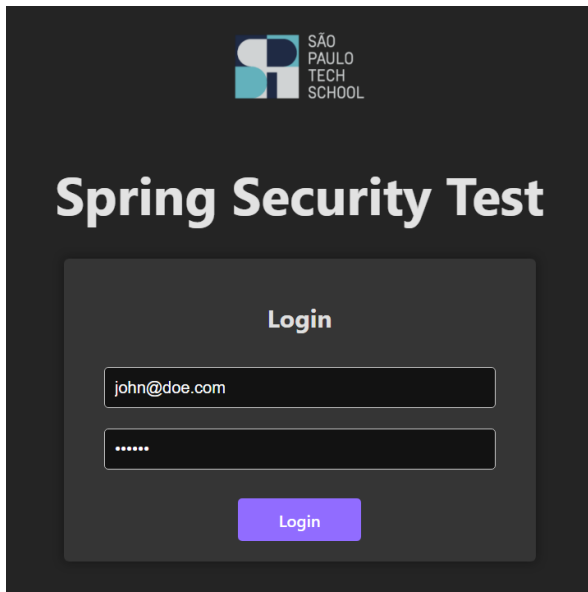
4. Bônus

Ao longo do livro você viu vários prints de uma API com Spring Boot, certo? Segue o link do repositório que possui todo o código demonstrado aqui aplicando a autenticação via JWT:

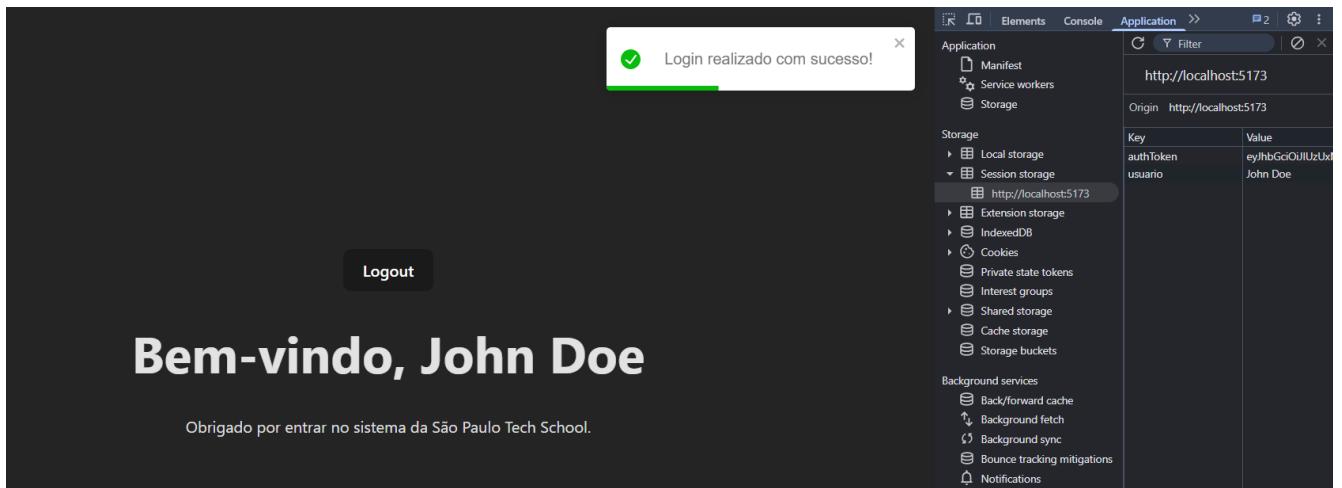
<https://github.com/BandTec/spring-security-jwt-exemplo>

Mas como seria se autenticar no front-end?

- Primeiro, certifique-se que seu back-end esteja rodando e então inicie sua aplicação front-end, na tela de login iremos passar as mesmas credenciais de usuário que passamos no Swagger:



- E assim que realizarmos o login, iremos entrar na nossa aplicação e o token ficará salvo no **Session storage**, podendo ser visualizado nas ferramentas de desenvolvedor:



Para mais detalhes de como funciona a implementação do JWT no front-end, segue o link de um repositório de exemplo feito com React:

<https://github.com/BandTec/spring-security-jwt-exemplo-react>