

SPTECH

# Spring Security e JWT

Guia para Implementação de Autenticação Baseada em Tokens

SPTECH

23/3/2023

## Sumário

Passo 1 - Configuração inicial (Criação da API) .....	2
Camadas .....	2
Domain .....	3
Repository .....	4
Service .....	5
Data transfer Object (DTO) .....	6
API .....	8
Passo 2 – Spring Security e JWT .....	11
Dependências .....	12
Novas Classes .....	13
Passo 3 - Utilizando configuração .....	24
Service .....	24
API .....	25

## Passo 1 - Configuração inicial (Criação da API)

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 passo 2. Lembre-se de incluir as dependências necessárias.



**Project**

☐ Gradle - Groovy  
☐ Gradle - Kotlin  
☒ Maven

**Language**

☒ Java  
☐ Kotlin  
☐ Groovy

**Spring Boot**

☐ 3.2.0 (SNAPSHOT) ☐ 3.2.0 (M3) ☐ 3.1.5 (SNAPSHOT)  
☒ 3.1.4 ☐ 3.0.12 (SNAPSHOT) ☐ 3.0.11  
☐ 2.7.17 (SNAPSHOT) ☐ 2.7.16

**Project Metadata**

Group

school.sptech

Artifact

exemplo-jwt

Name

exemplo-jwt

Description

Projeto com JWT

Package name

school.sptech.exemplo-jwt

Packaging

☒ Jar ☐ War

Java

☐ 21 ☒ 17 ☐ 11 ☐ 8

**Dependencies**

**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

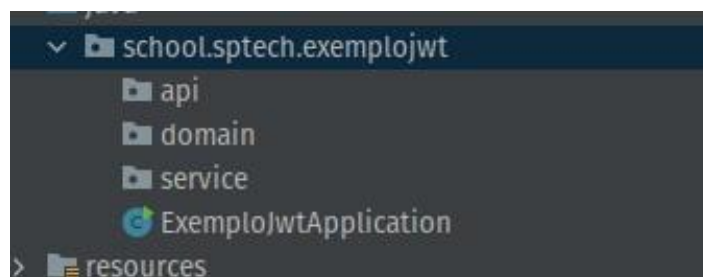
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

**Validation** I/O

Bean Validation with Hibernate validator.

## Camadas

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



Para deixar o nosso código mais organizado, vamos dividir em três pacotes diferentes:

- **Pacote Api:** Aqui ficarão as configurações do projeto e os endpoints.
- **Pacote Domain:** Este pacote terá as entidades do banco de dados.
- **Pacote Service:** Aqui concentraremos toda a lógica da regra de negócio.

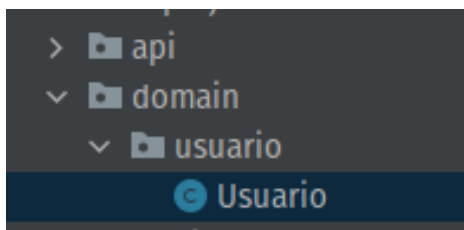
## Domain

A camada de domínio (domain) é responsável por definir as entidades de domínio do aplicativo, mapeá-las para o banco de dados e garantir a integridade dos dados. Além disso, é nessa camada que as regras de negócio são definidas. A separação das entidades em sua própria camada facilita a manutenção do código e a implementação de mudanças nas regras de negócio, já que ela permite uma clara separação de responsabilidades entre as diferentes partes do aplicativo. Em suma, a camada domain é uma parte essencial de qualquer projeto Java Spring Boot e deve ser cuidadosamente planejada e implementada para garantir um aplicativo eficiente e fácil de manter.

Para criar uma entidade de usuário e, conseqüentemente, gerar uma tabela correspondente no banco de dados, siga as etapas abaixo:

1. Clique com o botão direito do mouse no diretório principal do seu projeto.
2. Selecione "New" (Novo) e depois "Package" (Pacote).
3. Dê um nome para o pacote, como "usuario".
4. Clique com o botão direito do mouse no pacote "usuario" criado e selecione "Class" (Classe).
5. Dê um nome para a classe, como "Usuario".
6. Adicione os atributos que deseja que a entidade "Usuario" tenha, como nome, sobrenome, e-mail e senha. Certifique-se de que cada atributo tenha um tipo de dado correspondente.

Exemplo:



```

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Usuario {

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

    private String nome;
    private String email;
    private String senha;

    // Gerar getters e setters
}

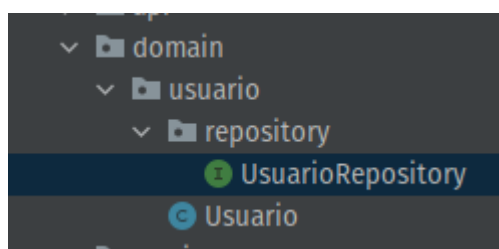
```

## Repository

Para criar uma interface de repositório para a entidade "Usuario", siga as seguintes etapas:

1. Clique com o botão direito do mouse no pacote "domain/usuario".
2. Selecione "New" (Novo) e depois "Package" (Pacote).
3. Dê um nome para o pacote, como "repository".
4. Clique com o botão direito do mouse no pacote "repository" criado e selecione "Interface".
5. Dê um nome para a interface, como "UsuarioRepository".

Exemplo:



```
@Repository
public interface UsuarioRepository extends
JpaRepository<Usuario, Long> {

    Optional<Usuario> findByEmail(String email);

}
```

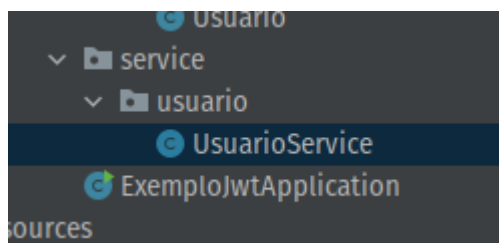
## Service

A camada service em um projeto Java Spring Boot é uma camada intermediária entre a camada de controladores (controllers) e a camada de persistência (repositories). É responsável por implementar a lógica de negócio da aplicação e realizar ações complexas sobre os dados, garantindo que essas operações sigam as regras de negócios definidas pela aplicação.

Essa camada é onde são implementados os serviços do aplicativo, como autenticação de usuário, validação de dados, cálculos complexos e regras de negócio específicas. É também onde as transações de banco de dados são gerenciadas e onde a validação de entrada é realizada. Em resumo, a camada service é responsável por encapsular a lógica de negócio e fornecer uma interface consistente para o restante do aplicativo.

Uma das principais vantagens de usar a camada service é que ela separa as responsabilidades do aplicativo de forma clara e eficiente. Isso torna o código mais organizado e mais fácil de entender e manter, além de permitir que o aplicativo seja escalável e flexível em relação a mudanças nas regras de negócio.

Exemplo:



```
@Service
public class UsuarioService {

}
```

A anotação "@Service" é utilizada no framework Spring para identificar que uma determinada classe é um componente de serviço. Essa prática ajuda a estruturar melhor o código, além de permitir a injeção de dependência de forma mais fácil e segura, já que outros componentes podem acessar as funcionalidades oferecidas por esse serviço.

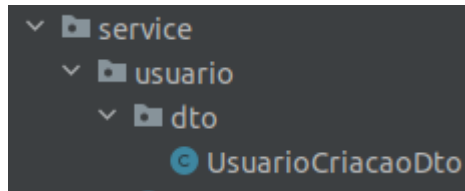
### Data transfer Object (DTO)

O padrão DTO (Data Transfer Object) é utilizado para transferir dados entre diferentes componentes de um sistema de maneira organizada. Sua função principal é encapsular as informações de uma entidade e transferi-las para outro componente sem expor sua estrutura interna.

O uso de DTOs é especialmente útil em aplicações com diferentes camadas, como apresentação, serviço e acesso a dados. Eles ajudam a garantir que as informações sejam transferidas de maneira segura e organizada entre essas camadas.

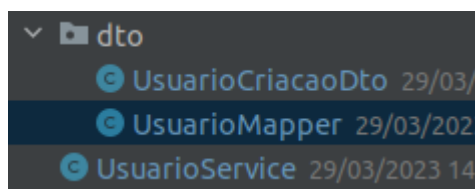
Para criar um DTO, é necessário criar uma classe Java com os campos necessários para a transferência de dados e métodos getters e setters correspondentes. Em seguida, basta usar essa classe para encapsular as informações a serem transferidas entre as diferentes camadas do aplicativo.

Dentro do pacote **"service/usuario"**, crie um pacote chamado **"dto"**. Em seguida, dentro deste novo pacote, crie uma classe Java chamada **"UsuarioCriacaoDto"**. Essa classe será utilizada para transferir os dados necessários para a criação de um novo usuário entre as diferentes camadas do aplicativo com validações.



```
public class UsuarioCriacaoDto {  
  
    @Size(min = 3, max = 10)  
    @Schema(description = "Nome do usuário", example = "Rafael Reis")  
    private String nome;  
  
    @Email  
    @Schema(description = "Email do usuário", example = "rafael.reis@sptech.school")  
    private String email;  
  
    @Size(min = 6, max = 20)  
    @Schema(description = "Senha do usuário", example = "AdminAdmin")  
    private String senha;  
  
    // Getters e Setters  
}
```

Dentro do pacote criado anteriormente, crie uma classe chamada **UsuarioMapper**. Essa classe terá a responsabilidade de realizar o mapeamento e transferência de dados entre objetos, ou seja, ela será responsável por transformar os objetos da classe **UsuarioCriacaoDto** em objetos da classe **Usuario** e vice-versa, quando necessário. Essa prática de separação de responsabilidades ajuda a manter o código mais organizado e modular, facilitando a manutenção e a evolução do sistema.





```

public class UsuarioMapper {

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

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

        return usuario;
    }
}

```

Dentro da classe `UsuarioService`, é possível escrever um método chamado "criar" sem retorno que recebe uma DTO (Data Transfer Object), converte-a em uma entidade e a registra no banco de dados.

```

@Service
public class UsuarioService {

    @Autowired
    private UsuarioRepository usuarioRepository;

    public void criar(UsuarioCriacaoDto usuarioCriacaoDto){
        final Usuario novoUsuario = UsuarioMapper.of(usuarioCriacaoDto);
        this.usuarioRepository.save(novoUsuario);
    }
}

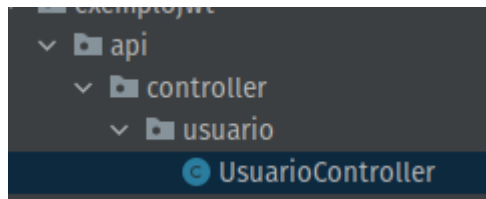
```

## API

Crie uma classe chamada `UserController` dentro do pacote **api/controller/usuario**, siga os seguintes passos:

1. Clique com o botão direito do mouse no projeto no qual deseja criar a classe e selecione a opção "New" no menu que aparece.
2. Selecione a opção "Package" para criar um novo pacote e digite "api/controller" como nome do pacote.
3. Clique novamente com o botão direito do mouse no pacote "api/controller" e selecione a opção "New" no menu que aparece.
4. Selecione a opção "Package" novamente para criar um novo pacote dentro do pacote "api/controller" e digite "usuario" como nome do pacote.
5. Clique novamente com o botão direito do mouse no pacote "api/controller/usuario" e selecione a opção "New" no menu que aparece.

6. Selecione a opção "Class" para criar uma classe e digite "UserController" como nome da classe.



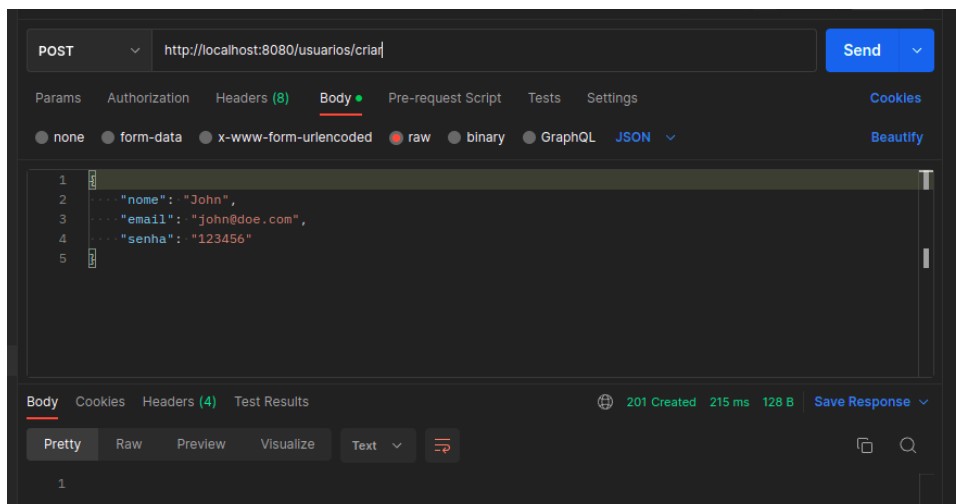
Agora você tem uma nova classe chamada UserController dentro do pacote api/controller/usuario. Você pode implementar os métodos desejados nesta classe para lidar com as requisições relacionadas ao usuário.

```
@RestController
@RequestMapping("/usuarios")
public class UserController {

    @Autowired
    private UsuarioService usuarioService;

    @PostMapping
    @SecurityRequirement(name = "Bearer")
    public ResponseEntity<Void> criar(@RequestBody @Valid UsuarioCriacaoDto usuarioCriacaoDto) {
        this.usuarioService.criar(usuarioCriacaoDto);
        return ResponseEntity.status(201).build();
    }
}
```

Chame o endpoint fornecendo um objeto para realizar o cadastro. Se a operação for concluída com sucesso, a resposta será um status 201.



Após o processo de cadastro, vamos realizar uma verificação no banco de dados H2 para confirmar se o registro foi cadastrado com sucesso.

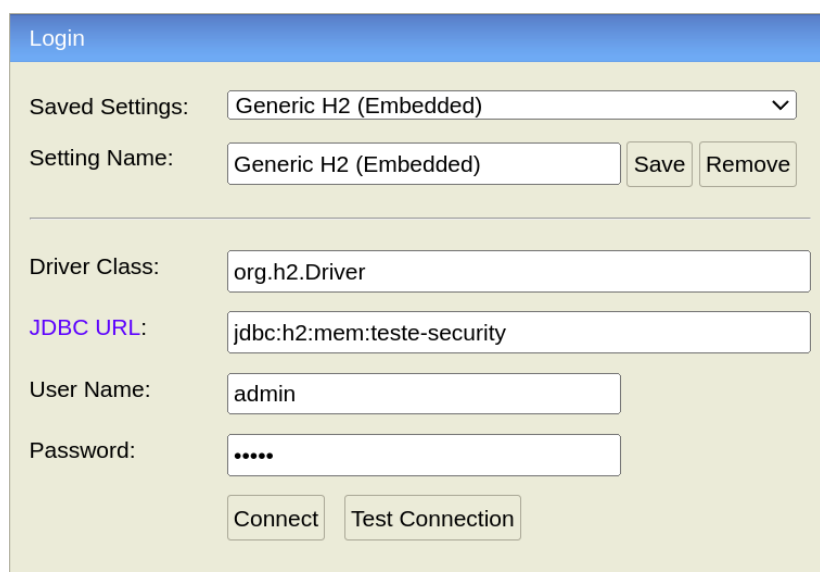
Para definir as configurações do banco de dados H2 no projeto, é preciso editar o arquivo **"application.properties"**. Nesse arquivo, é possível configurar diversas propriedades, mas no exemplo em questão, serão necessárias apenas algumas delas (inclusive validações):

```
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
```

Para acessar o console H2, basta digitar a URL <http://localhost:{porta}/h2-console> em seu navegador de preferência, onde **{porta}** representa a porta em que a sua aplicação está sendo executada. Ao acessar essa URL, você será direcionado para a tela inicial do console H2, como

mostrado no exemplo abaixo:



Login

Saved Settings: Generic H2 (Embedded) ▼

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:teste-security

User Name: admin

Password: .....

Connect Test Connection

Ao acessar o console H2, é necessário utilizar o usuário e a senha configurados no arquivo "application.properties". Caso não tenha especificado um usuário ou senha, será utilizada a senha padrão do H2, na qual o usuário é "sa" e a senha está vazia (em branco).

Após o acesso ao console H2, é possível realizar uma consulta para validar se o registro foi inserido com sucesso no banco de dados. Segue abaixo um exemplo:

RunRun SelectedAuto completeClear

select \* from usuario;

select \* from usuario;

ID	EMAIL	NOME	SENHA
1	john@doe.com	John	123456

(1 row, 2 ms)

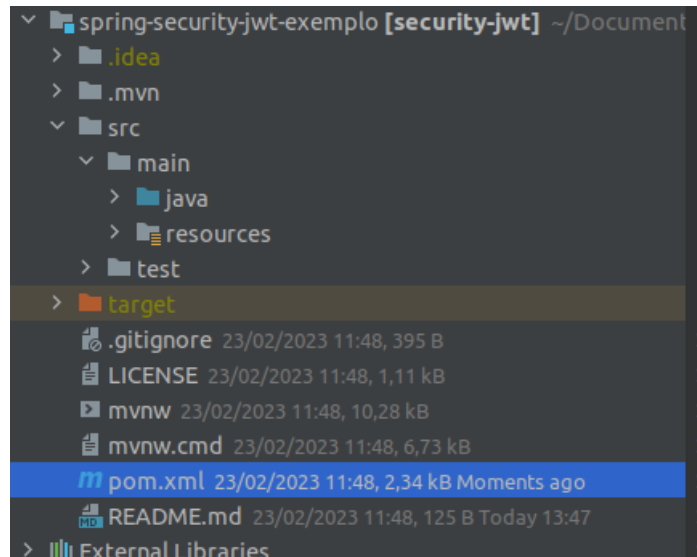
## Passo 2 – Spring Security e JWT

O Spring Security é um framework de segurança para aplicações Java que se baseia no Spring Framework. Ele fornece um conjunto de recursos que incluem autenticação, autorização e proteção contra ataques, além de suporte a protocolos de segurança populares como OAuth e JWT. O Spring Security é altamente personalizável e pode ser configurado de acordo com as necessidades específicas do usuário. Devido à sua eficácia, é amplamente utilizado em aplicações web empresariais para proteger recursos e dados sensíveis.

O JWT, por sua vez, é um formato de token de segurança que tem ganhado popularidade na web. Ele é usado para autenticar e autorizar usuários em sistemas web e é composto por três partes separadas codificadas em Base64 - cabeçalho, payload e assinatura digital. O cabeçalho inclui informações sobre o algoritmo de criptografia, enquanto o payload armazena informações sobre o usuário e suas permissões. A assinatura é gerada a partir das duas primeiras partes e serve para validar a autenticidade do token.

## Dependências

Para iniciar o desenvolvimento da parte de autenticação, é preciso adicionar novas dependências ao projeto. Para isso, será necessário abrir o arquivo pom.xml localizado na raiz do projeto.



Abra o arquivo e inclua as dependências abaixo:

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

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-test</artifactId>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
</dependency>

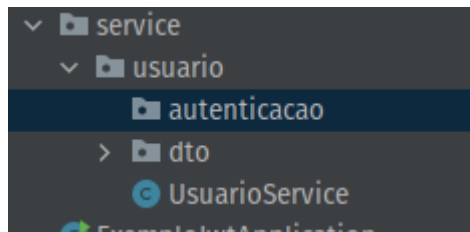
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.2</version>
</dependency>

<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.11.1</version>
  <scope>runtime</scope>
</dependency>

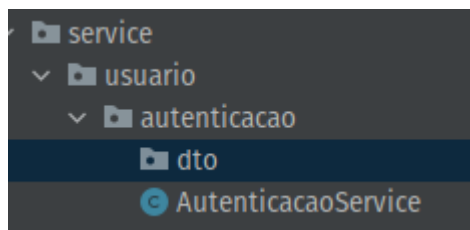
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
  <version>3.0.5</version>
</dependency>
```

## Novas Classes

Iremos criar um pacote chamado "autenticacao" dentro do diretório "service/usuario".



Além disso, será necessário criar um pacote chamado "dto" dentro do diretório recém-criado "autenticacao".



Crie 3 classes dentro desse pacote

1. O "UsuarioDetalhesDto" agora contém apenas o método getName. Todos os outros métodos são implementados da classe "UserDetails" do Spring Security, com nossos atributos implementados como retorno.

Lembre-se de alterar o "getPassword" e "getUsername" de null para "senha" e "email".

Altere também todas as booleanas de "false" para "true" no retorno dos métodos.

```

public class UsuarioDetalhesDto implements UserDetails {

    private final String nome;

    private final String email;

    private final String senha;

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

    public String getNome() {
        return nome;
    }

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

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

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

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

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

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

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

```

## 2. "UsuarioLoginDto"

```

public class UsuarioLoginDto {

    private String email;
    private String senha;

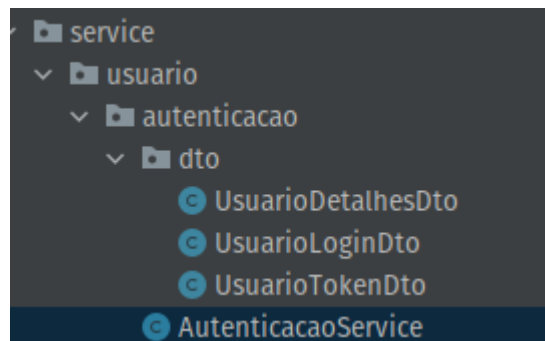
    // Criar getters e setters
}

```

### 3. "UsuarioTokenDto"

```
public class UsuarioTokenDto {  
  
    private Long userId;  
    private String nome;  
    private String email;  
    private String token;  
  
    // Criar getter e setters  
}
```

Dentro do pacote "autenticacao", defina uma classe intitulada "AutenticacaoService".



Na classe de serviço, estamos realizando a implementação da interface UserDetailsService do Spring Security.

```
@Service  
public class AutenticacaoService implements UserDetailsService {  
  
    @Autowired  
    private UsuarioRepository usuarioRepository;  
  
    // Método da interface implementada  
    @Override  
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
  
        Optional<Usuario> usuarioOpt = usuarioRepository.findByEmail(username);  
  
        if (usuarioOpt.isEmpty()) {  
            throw new UsernameNotFoundException(String.format("usuario: %s nao encontrado", username));  
        }  
  
        return new UsuarioDetalhesDto(usuarioOpt.get());  
    }  
}
```

**UserDetailsService** é uma interface do Spring Security que fornece um método para carregar informações do usuário com base em seu nome de usuário. Essas informações incluem as credenciais do usuário (como senha) e as autorizações associadas a ele. O método **loadUserByUsername()** é responsável por retornar um objeto **UserDetails**, que contém essas informações e é usado pelo Spring Security para autenticar e autorizar usuários em um



sistema. A implementação da interface **UserDetailsService** é geralmente personalizada para atender às necessidades específicas do aplicativo, como recuperar informações do usuário de um banco de dados ou de um serviço de autenticação externo.

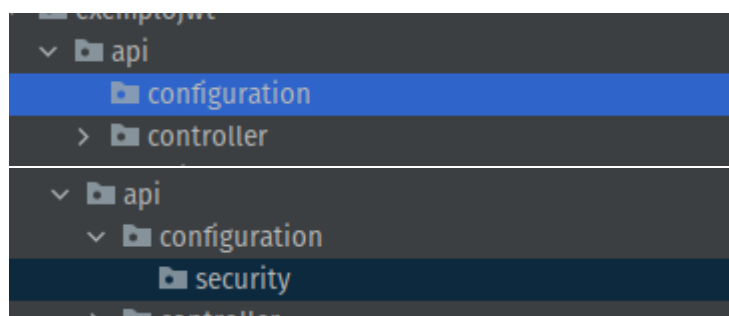
Retorne para a classe `UsuarioMapper` e inclua o método a seguir.

```
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;
}
```

Crie um pacote denominado **"configuration"** dentro de **"api"** e, em seguida, crie um pacote adicional chamado **"security"** dentro de **"configuration"**.



Crie a classe **"AutenticacaoEntryPoint"** dentro do pacote **"security"** usando o seguinte código:

```
@Component
public class AutenticacaoEntryPoint implements AuthenticationEntryPoint {

    @Override
    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);
        }
    }
}
```

**"AuthenticationEntryPoint"** é uma interface do Spring Security que define como a aplicação deve lidar com as requisições não autenticadas. Quando o usuário tenta acessar um endpoint protegido sem estar autenticado, o Spring Security redireciona o usuário para um ponto de entrada de autenticação.

A implementação dessa interface permite que a aplicação defina como deve ser tratado esse redirecionamento. Por exemplo, é possível retornar uma mensagem de erro personalizada ou redirecionar o usuário para uma página de login.

- **HttpServletResponse.SC\_UNAUTHORIZED (401)** - Informa para quem realizou a requisição que não está autorizado a utilizar o sistema.
- **HttpServletResponse.SC\_FORBIDDEN (403)** - Informa para quem realizou a requisição que não possui permissões suficientes para acessar um recurso específico.

Crie a classe "**AutenticacaoProvider**" dentro do pacote "**security**" usando o seguinte código:

```
public class AutenticacaoProvider implements AuthenticationProvider {

    private final AutenticacaoService usuarioAutorizacaoService;
    private final PasswordEncoder passwordEncoder;

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

    @Override
    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);

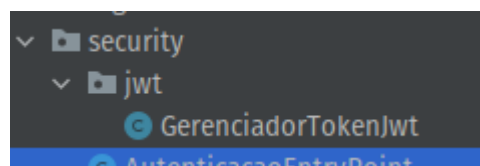
        if (this.passwordEncoder.matches(password, userDetails.getPassword())) {
            return new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
        } else {
            throw new BadCredentialsException("Usuário ou Senha inválidos");
        }
    }

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

"**AuthenticationProvider**" é uma interface do Spring Security que define a lógica para autenticar usuários. Essa interface é responsável por processar as informações de autenticação fornecidas pelo usuário e verificar se as credenciais são válidas.

Uma implementação do "**AuthenticationProvider**" pode verificar as credenciais do usuário em um banco de dados, um serviço de autenticação externo ou outro meio de armazenamento de credenciais.

Crie um pacote chamado "**jwt**" dentro do pacote "**security**":



Crie a classe **"GerenciadorTokenJwt"** dentro do pacote **"jwt"** usando o seguinte código:

```
public class GerenciadorTokenJwt {

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

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

        // 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) {
        Claims claims = getAllClaimsFromToken(token);
        return claimsResolver.apply(claims);
    }

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

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

    private Claims getAllClaimsFromToken(String token) {
        return Jwts.parserBuilder()
            .setSigningKey(parseSecret())
            .build()
            .parseClaimsJws(token).getBody();
    }

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

O **"GerenciadorTokenJwt"** é uma classe responsável por gerar e validar tokens JWT (JSON Web Tokens) usados para autenticar usuários em um sistema seguro.

Ao gerar um token JWT, a classe **"GerenciadorTokenJwt"** codifica as informações do usuário em uma sequência de caracteres criptografada e adiciona uma assinatura digital para garantir que o token não tenha sido alterado durante a transmissão.

Para utilizar o token JWT, é necessário configurar duas variáveis no projeto. Para isso, siga os passos abaixo:

1. Abra o arquivo "**application.properties**" do seu projeto.
2. Adicione as seguintes linhas no final do arquivo:

```
# validade do token

jwt.validity=3600000

# palavra passe do token (segredo) necessita de no mínimo 32 caracteres

jwt.secret=RXhpc3RlIHVtYSB0ZW9yaWEgcXVlIGRpeiBxdWUsIHNIHVtIGRpYSBhbGd16W0gZGVzY29icmlyIGV4YXRhbWVudGUgcGFyYSBxdWUgc2VydmUgbyBVbml2ZXJzbyBliHBvc iBxdWUgZWxlIGVzd0EgYXF1aSwgZWxlIGRlc2FwYXJlY2Vy4SBpbmN0YW50YW5lYW1lbnRlIGUgc2Vy4SBzdWJzdGl0de1kbyBwb3IgYWxnbyBhaw5kYSBtYWlzIGVzdHJhbmhvIGUgaW5leHBsaWPhdmVsLiBFeGlzdGUgdW1hIHNLZ3VuZGEgdGVvcmlhIHFI1ZSBkaXogcXVlIGlzc28gauEgYWVnbmRlY2V1Li4u
```

A primeira linha define o tempo de expiração do token em milissegundos. Nesse exemplo, o token expirará após uma hora (3600000 milissegundos).

A segunda linha define a chave secreta usada para assinar e verificar os tokens JWT. É recomendável usar uma chave complexa e difícil de adivinhar para garantir a segurança do sistema.

3. Salve o arquivo e reinicie o servidor para que as alterações tenham efeito.

Com essas variáveis configuradas, você pode usar a classe "**GerenciadorTokenJwt**" para gerar e validar os tokens JWT em seu sistema.

Crie a classe **"AutenticacaoFilter"** na raiz do pacote **"security"** usando o seguinte código:

```
public class AutenticacaoFilter extends OncePerRequestFilter {

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

    private final AutenticacaoService autenticacaoService;

    private final GerenciadorTokenJwt jwtTokenManager;

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

    @Override
    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(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) {

        UserDetails userDetails = autenticacaoService.loadUserByUsername(username);

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

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

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

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

**"AutenticacaoFilter"** é uma classe do Spring Security responsável por processar as solicitações de autenticação do usuário e realizar a validação das credenciais fornecidas pelo usuário.

O **"AutenticacaoFilter"** é uma implementação do **"AbstractAuthenticationProcessingFilter"**, que é a classe base fornecida pelo Spring Security para processar as solicitações de autenticação. O **"AutenticacaoFilter"** é responsável por extrair as informações de autenticação da solicitação HTTP recebida, criar uma instância de **"Authentication"** e enviar essa instância para o **"AuthenticationManager"** realizar a autenticação.

Após a autenticação bem-sucedida, o **"AutenticacaoFilter"** é responsável por gerar o token de acesso e retorná-lo ao cliente. O token é geralmente armazenado pelo cliente e enviado em cada solicitação subsequente para validar a autenticação do usuário.

Novamente dentro do pacote **"security"**, crie a classe **"SecurityConfiguracao"** usando o seguinte código:

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfiguracao {
    private static final String ORIGENS_PERMITIDAS = "*";

    @Autowired
    private AutenticacaoService autenticacaoService;

    @Autowired
    private AutenticacaoEntryPoint autenticacaoJwtEntryPoint;

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

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .cors(Customizer.withDefaults())
            .csrf(CsrfConfigurer<HttpSecurity>::disable)
            .authorizeHttpRequests(authorize -> authorize.requestMatchers(URLS_PERMITIDAS)
                .permitAll()
                .anyRequest()
                .authenticated()
            )
            .exceptionHandling(handling -> handling
                .authenticationEntryPoint(autenticacaoJwtEntryPoint))
            .sessionManagement(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
    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("/**", configuracao);

        return origem;
    }
}
```



A classe "**SecurityConfiguracao**" é responsável por configurar e filtrar os endpoints para permitir apenas o acesso autenticado. O método "**public SecurityFilterChain filterChain(HttpSecurity http)**" é responsável por configurar o CORS e os endpoints que não exigem autenticação, incluindo o endpoint de login.

SecurityFilterChain é uma interface do Spring Security que define um filtro de segurança para uma solicitação HTTP. Ela é responsável por garantir que as solicitações sejam autorizadas e autenticadas corretamente antes de permitir o acesso ao endpoint. A interface define um único método, "**doFilter**", que processa a solicitação e a encaminha para o próximo filtro na cadeia. A configuração da cadeia de filtros é feita por meio do método "**filterChain**" em uma classe que implementa a interface "**WebSecurityConfigurerAdapter**".

### Versão com o Spring 3.0.5

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.headers()
        .frameOptions().disable()
        .and()
        .cors()
        .configurationSource(request -> buildCorsConfiguration())
        .and()
        .csrf()
        .disable()
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers(URLS_PERMITIDAS)
            .permitAll()
            .anyRequest()
            .authenticated()
        )
        .exceptionHandling()
        .authenticationEntryPoint(authenticacaoJwtEntryPoint)
        .and()
        .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS);

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

    return http.build();
}
```

### Versão com o Spring 3.1.4

```

@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .cors(Customizer.withDefaults())
        .csrf(CsrfConfigurer<HttpSecurity>::disable)
        .authorizeHttpRequests(authorize -> authorize.requestMatchers(URLS_PERMITIDAS)
            .permitAll()
            .anyRequest()
            .authenticated()
        )
        .exceptionHandling(handling -> handling
            .authenticationEntryPoint(authenticacaoJwtEntryPoint))
        .sessionManagement(management -> management
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS));

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

    return http.build();
}

```

Todas as URLs que serão permitidas sem a necessidade de autenticação por token.

```

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

```

```

        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers(URLS_PERMITIDAS)
            .permitAll()
            .anyRequest()
            .authenticated()
        )

```

O método **requestMatchers()** permite especificar os padrões de URL e as regras de autorização que se aplicam a esses padrões. No nosso caso, não queremos restringir o



acesso para os endpoint de **/usuarios/login** e **/h2-console/** com tudo que vier depois do **/** por exemplo.

## Passo 3 - Utilizando configuração

### Service

Vá até a classe "**UsuarioService**" e no método "**criar**" da classe "**UsuarioService**", adicione a criptografia da senha. Para isso, você pode adicionar as seguintes linhas abaixo:

Injete o PasswordEncoder na service.

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

```
@Autowired
private PasswordEncoder passwordEncoder;
```

Código do método por completo:

```

@Autowired
private PasswordEncoder passwordEncoder;

@Autowired
private UsuarioRepository usuarioRepository;

public void criar(UsuarioCriacaoDto usuarioCriacaoDto) {
    final Usuario novoUsuario = UsuarioMapper.of(usuarioCriacaoDto);

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

    this.usuarioRepository.save(novoUsuario);
}

```

Vamos criar um método na classe "**UsuarioService**" para a autenticação do usuário.

Adicione mais duas classes por injeção de dependência.

```

@Autowired
private GerenciadorTokenJwt gerenciadorTokenJwt;

@Autowired
private AuthenticationManager authenticationManager;

```

```

public UsuarioTokenDto autenticar(UsuarioLoginDto usuarioLoginDto) {

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

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

    Usuario usuarioAutenticado =
        usuarioRepository.findByEmail(usuarioLoginDto.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);
}

```

## API

Vamos implementar o endpoint para o login na classe "**UsuarioController**" utilizando o novo método criado na service.

```
@PostMapping("/login")
public ResponseEntity<UsuarioTokenDto> login(@RequestBody UsuarioLoginDto usuarioLoginDto){
    UsuarioTokenDto usuarioToken = this.usuarioService.autenticar(usuarioLoginDto);
    return ResponseEntity.status(200).body(usuarioToken);
}
```

Para iniciar a aplicação com um usuário cadastrado, basta incluir a seguinte linha de código no arquivo **application.properties**.

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

Com isso, ao adicionar o arquivo **"data.sql"** dentro de resources no projeto, conseguimos inserir scripts sql, exemplo:

```
data.sql
1 insert into usuario
2   (nome, email, senha)
3 values
4   ('John Doe', 'john@doe.com', '$2a$10$0/TKTGxdREbWwJWYhwF6e9P1fP0AMMNqEnZg0G95jnSkHSfkkIrC');
```

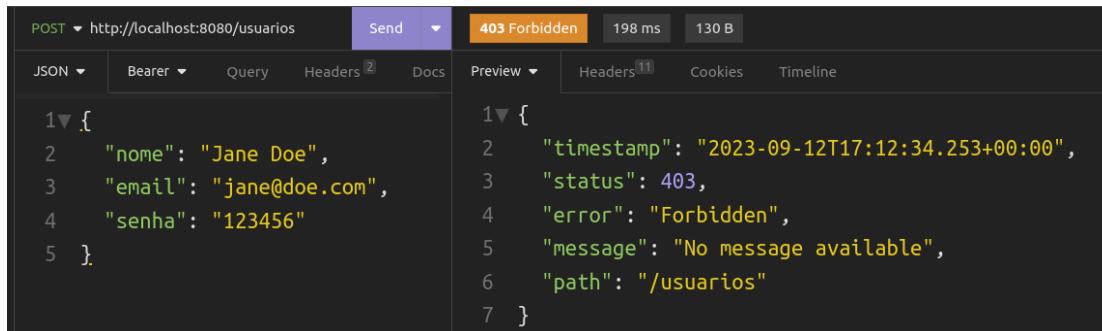
Logo após configurado, todas as chamadas que não estão no **antMatchers()** devem receber o token copiado que vem da resposta.

## Versão Insomnia

Para isso, abra o Insomnia e acesse o endpoint **/usuarios/login** com esse corpo para logar. Ao realizar o login com sucesso, retornará o objeto com um token, copie o token.

```
POST http://localhost:8080/usuarios/login 200 OK 158 ms 247 B 5 Minutes Ago
JSON Auth Query Headers Docs Preview Headers Cookies Timeline
1 {
2   "email": "john@doe.com",
3   "senha": "123456"
4 }
5 "token":
  "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJqb2huQGRvZS5jb20iLCJpYXQiOiJlE20TQ1MjksMTQsImV4cCI6MTY5ODk0OTExNH0udjgBdE0aFdkmjdJ8-3IwmuwiObYr1D4Wv3unWs_rbdT9UbIucW62-z3i63aRmLsVFYkC8hTSxid3UHSwRSF3w"
6 }
```

Acesse o endpoint **POST - /usuarios** para criar um novo usuário com esse objeto. Caso não passe token, o endpoint está bloqueado nas configurações e retornará 403 - não permitido acessar.



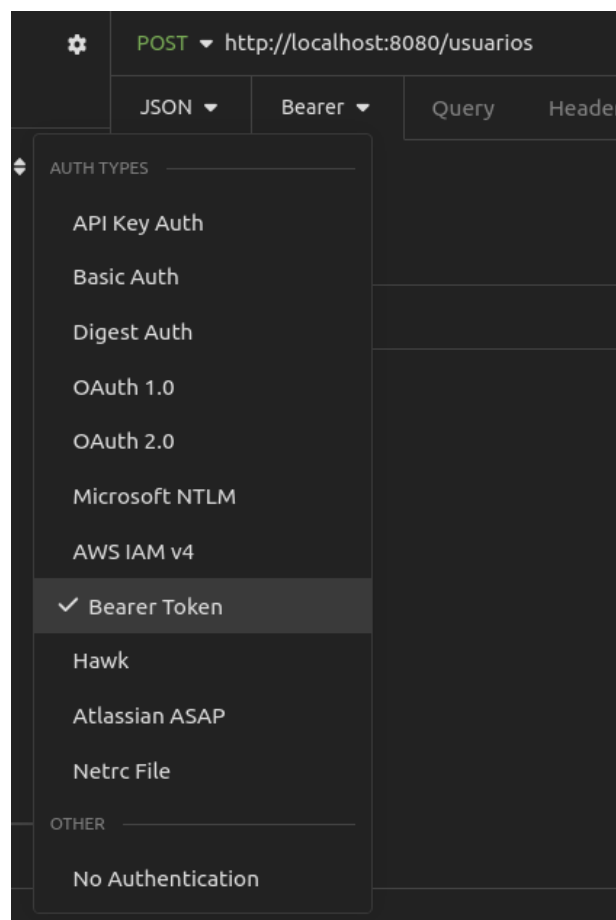
```
POST http://localhost:8080/usuarios 403 Forbidden 198 ms 130 B

JSON Bearer Query Headers Docs Preview Headers Cookies Timeline

1 {
2   "nome": "Jane Doe",
3   "email": "jane@doe.com",
4   "senha": "123456"
5 }

1 {
2   "timestamp": "2023-09-12T17:12:34.253+00:00",
3   "status": 403,
4   "error": "Forbidden",
5   "message": "No message available",
6   "path": "/usuarios"
7 }
```

Para mandar o token pela requisição, devemos passar um **headers** de **Authorizatio**, para isso, basta acessar o Auth e selecionar **Bearer Token**.



Agora, basta colar o token na área de token e fazer a requisição de cadastro (caso não passe token, o endpoint está bloqueado nas configurações e retornará 403 - não permitido acessar).



Para passar o token, vá em Authorization, acesse Bearer Token e cole na área de token.

