



# PW3 - Programação para a WEB III

## Spring Security (parte 2)

Conteúdo 12



Lembrando o finzinho da aula anterior...

Rodando o projeto...



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

/usr/lib/jvm/jdk-20/bin/java ...

```
 .   ----
 \  /   _ - _ - ( )_ -- -- - \ \ \ \
 ( ( )\__| ' | ' | ' | ' | \ / ` | \ \ \ \
 \| \| _ )| | _ )| | | | | | ( | | ) ) )
 ' | _ _ | . _ | _ | _ | _ \_, , | / / /
 ======|_|=====|_|=/_/_/_/
 :: Spring Boot ::           (v3.1.3)
```

```
2023-11-02T17:34:21.299-03:00 INFO 12274 --- [ restartedMain] br.edu.ifsp.pw3.api.ApiApplication      : Starting ApiApplication using Java 20.0.1 with PID 12
2023-11-02T17:34:21.302-03:00 INFO 12274 --- [ restartedMain] br.edu.ifsp.pw3.api.ApiApplication      : No active profile set, falling back to 1 default prof
2023-11-02T17:34:21.436-03:00 INFO 12274 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtoo
2023-11-02T17:34:21.436-03:00 INFO 12274 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting t
2023-11-02T17:34:22.713-03:00 INFO 12274 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT
2023-11-02T17:34:22.816-03:00 INFO 12274 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 81 ms. Fo
2023-11-02T17:34:23.855-03:00 INFO 12274 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
```

■ ■ ■

```
2023-11-02T17:34:24.995-03:00 INFO 12274 --- [ restartedMain] org.hibernate.cfg.Environment      : HHH000406: Using bytecode reflection optimizer
2023-11-02T17:34:25.153-03:00 INFO 12274 --- [ restartedMain] o.h.b.i.BytecodeProviderInitiator    : HHH000021: Bytecode provider name : bytebuddy
2023-11-02T17:34:25.390-03:00 INFO 12274 --- [ restartedMain] o.s.o.j.p.SpringPersistenceUnitInfo  : No LoadTimeWeaver setup: ignoring JPA class transform
2023-11-02T17:34:25.866-03:00 INFO 12274 --- [ restartedMain] o.h.b.i.BytecodeProviderInitiator    : HHH000021: Bytecode provider name : bytebuddy
2023-11-02T17:34:26.462-03:00 INFO 12274 --- [ restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator   : HHH000490: Using JtaPlatform implementation: [org.hib
2023-11-02T17:34:26.464-03:00 INFO 12274 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit
2023-11-02T17:34:26.950-03:00 WARN 12274 --- [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefor
2023-11-02T17:34:27.318-03:00 WARN 12274 --- [ restartedMain] .s.s.UserDetailsServiceAutoConfiguration :
```

Using generated security password: 6b2df463-3839-4ca1-9561-b7e823758a9f

This generated password is for development use only. Your security configuration must be updated before running your application in production.

```
2023-11-02T17:34:27.443-03:00 INFO 12274 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain     : Will secure any request with [org.springframework.sec
2023-11-02T17:34:27.482-03:00 INFO 12274 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2023-11-02T17:34:27.504-03:00 INFO 12274 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context pa
2023-11-02T17:34:27.516-03:00 INFO 12274 --- [ restartedMain] br.edu.ifsp.pw3.api.ApiApplication      : Started ApiApplication in 6.819 seconds (process runn
```

Vamos disparar a requisição que devolve a lista de médicos:

GET lista geral

HTTP PW3 / lista geral

GET localhost:8080/medicos

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies (1) Headers (13) Test Results

Pretty Raw Preview Visualize Text 

1

Status: 401 Unauthorized

GET lista geral

HTTP PW3 / lista geral

GET localhost:8081

Params Authorization Headers

Query Params

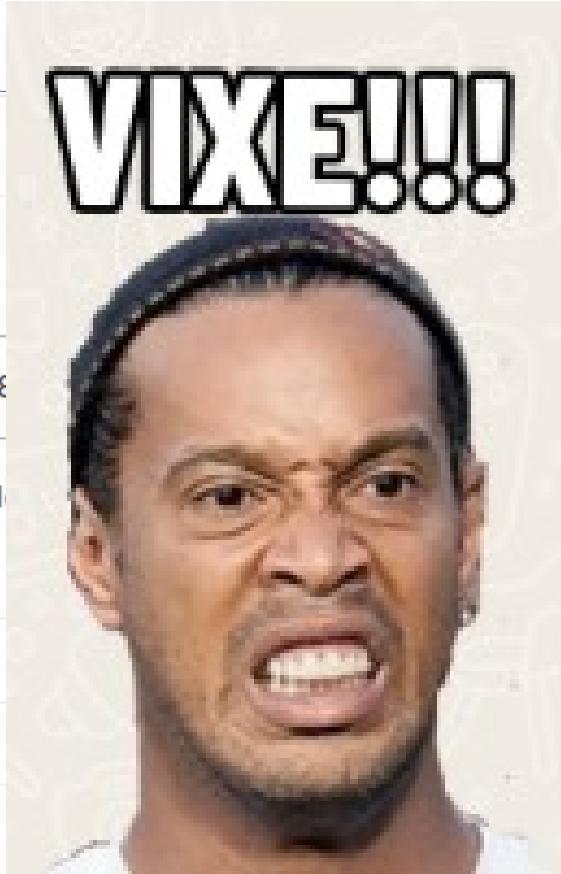
Key	Description
Key	Description

Body Cookies (1) Headers (13) Test Results

Pretty Raw Preview Visualize Text ↻

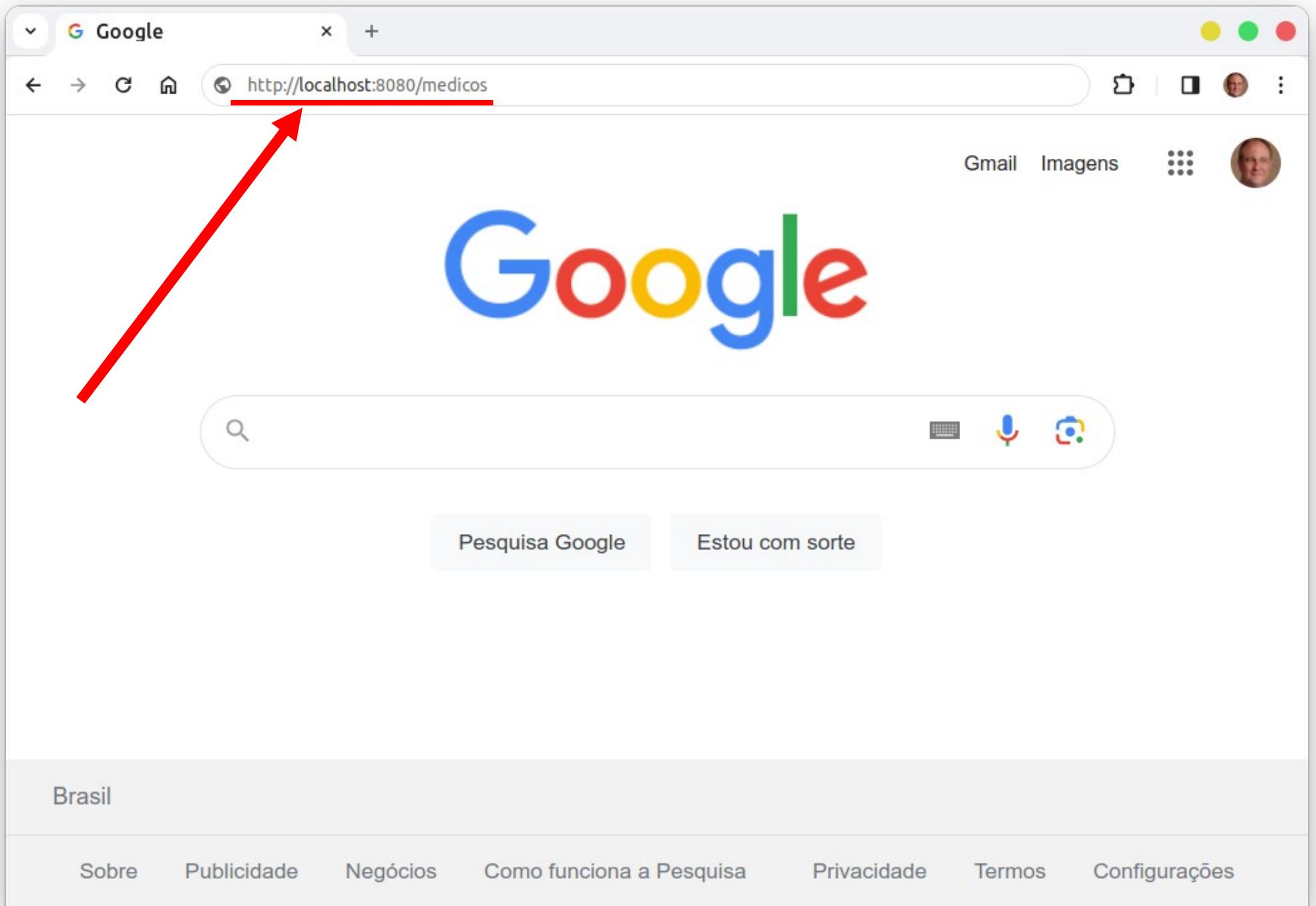
1

Status: 401 Unauthorized



Por padrão, bloqueia tudo!!!!

Vamos tentar no navegador:



Please sign in

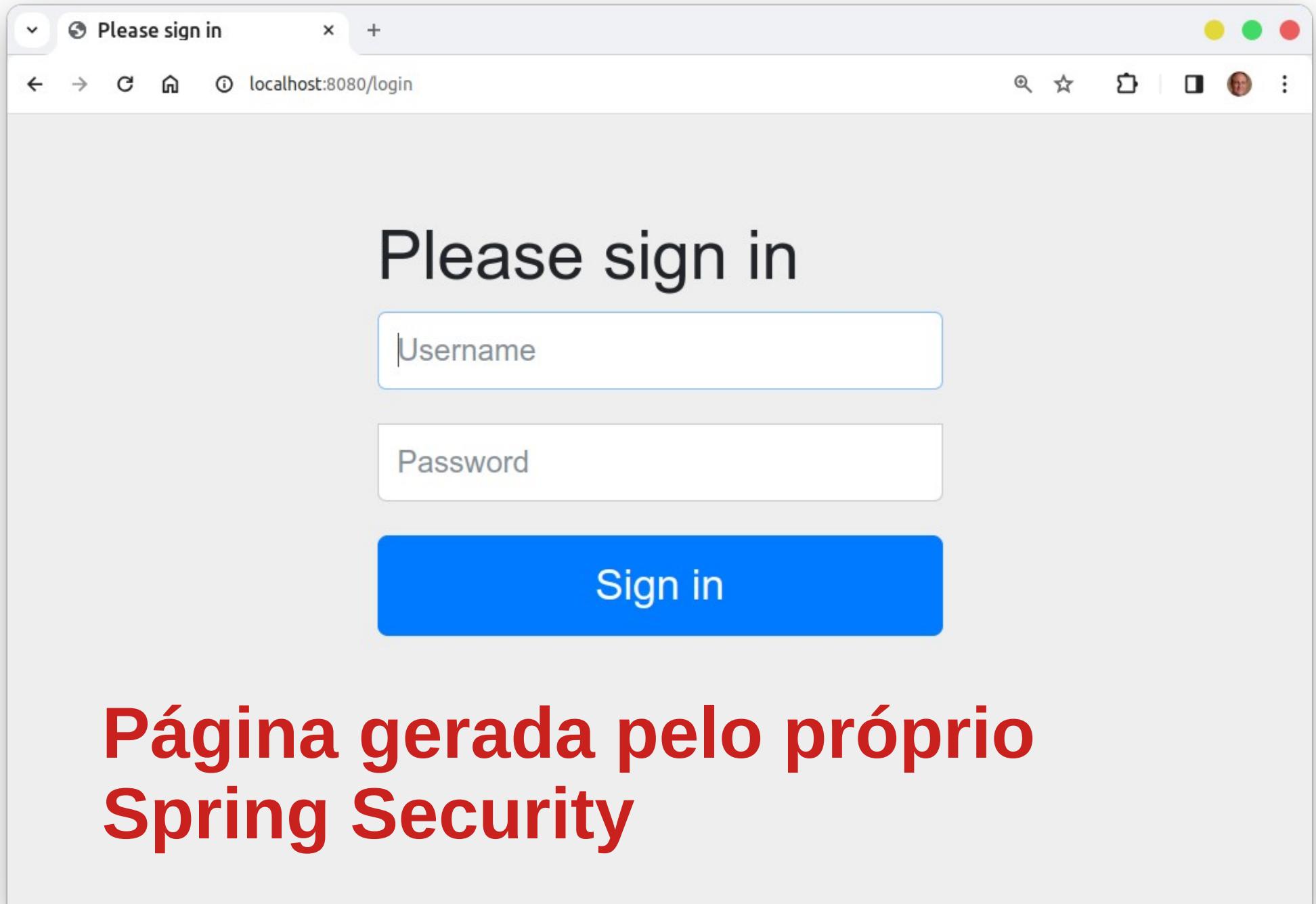
localhost:8080/login

# Please sign in

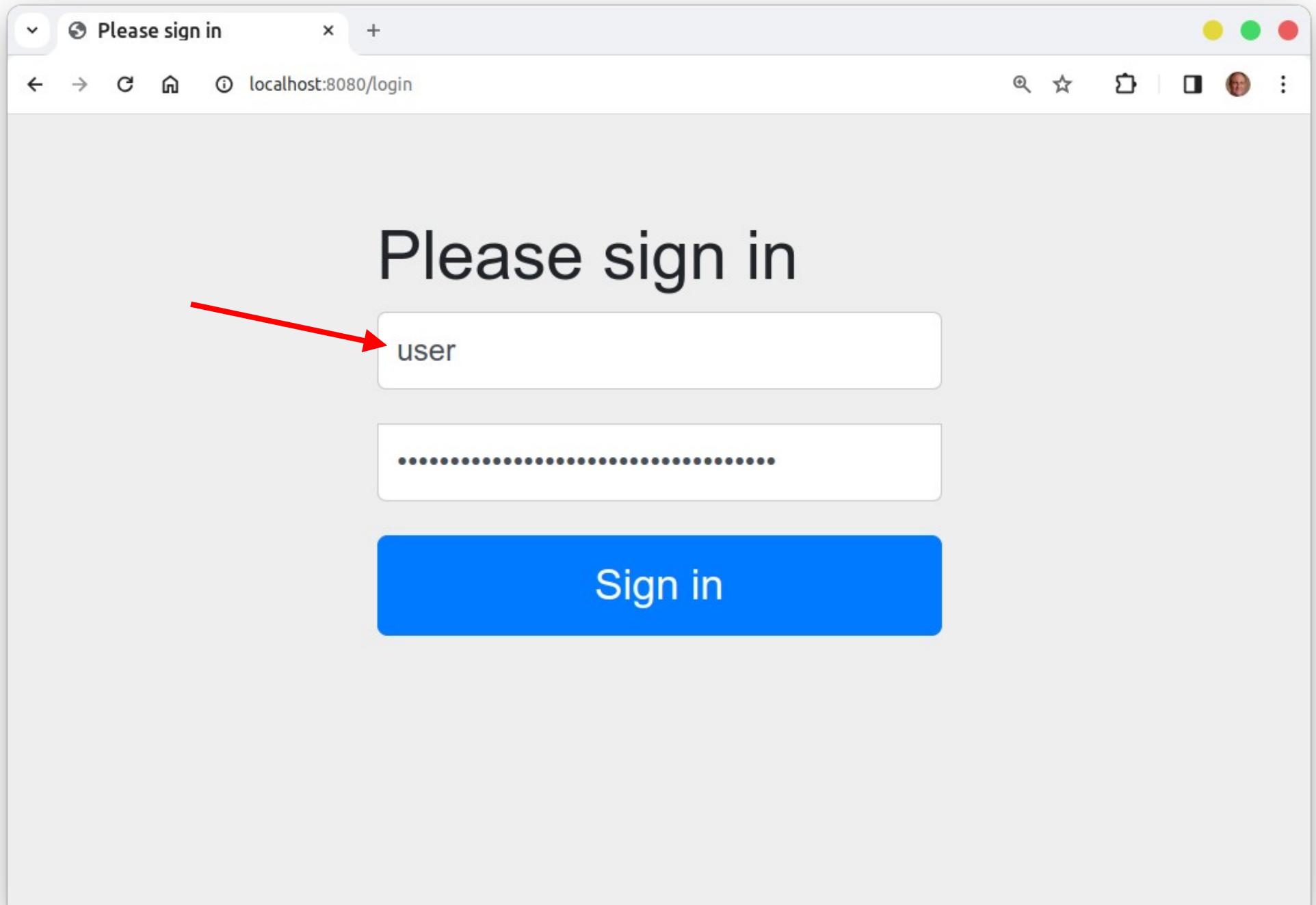
Username

Password

Sign in



Pode usar aquela senha gerada e exibida no log do Spring



localhost:8080/medicos

```
// 20231102180453
// http://localhost:8080/medicos?continue

[

{
    "id": 2,
    "nome": "Lupita Pereira",
    "email": "lupy@gmail.com",
    "telefone": "16992482222",
    "crm": "222222",
    "especialidade": "DERMATOLOGIA",
    "endereco": {
        "logradouro": "RUA DOIS",
        "bairro": "BAIRRO DOIS",
        "cep": "22222222",
        "numero": null,
        "complemento": null,
        "cidade": "Ibaté",
        "uf": "SP"
    },
    "ativo": false
},
{
    "id": 3,
```

Isso é o que o Spring faz por default.  
Mas não é isso que queremos.  
Queremos autenticação Stateless.

Vamos começar a alterar o projeto  
para atingir esse objetivo.

**OBS:**

Diferente do que foi feito em aulas anteriores,  
precisaremos configurar várias  
coisas antes de ver resultados  
práticos!

# ACOMPANHAMENTO DOS PASSOS

- Adicionado o Spring Security no projeto.
  - Por padrão, já bloqueia tudo.

**Como falei na aula passada, é bastante complexo,  
muitos detalhes...**

**Vou fazendo um registro passo-a-passo ao longo do processo,  
pra gente tentar não se perder...**

# Cadastro de usuários (login e senha) no BD

# Criando a classe Usuario

The screenshot shows the IntelliJ IDEA interface with the project structure on the left and the code editor on the right.

**Project Structure:**

- Project: api (~/HD\_PRINCIPAL/Dropbox/intellij)
- src
  - main
    - java
      - br.edu.ifsp.pw3.api
        - controller
        - endereco
        - medico
        - usuario
      - util
    - ApiApplication
  - resources
  - test
  - target
  - .gitignore
  - HELP.md
  - mvnw
  - mvnw.cmd
  - pom.xml
- External Libraries
- Scratches and Consoles

A red arrow points from the "usuario" folder in the project tree to the "Usuario" class in the code editor.

**Code Editor (Usuario.java):**

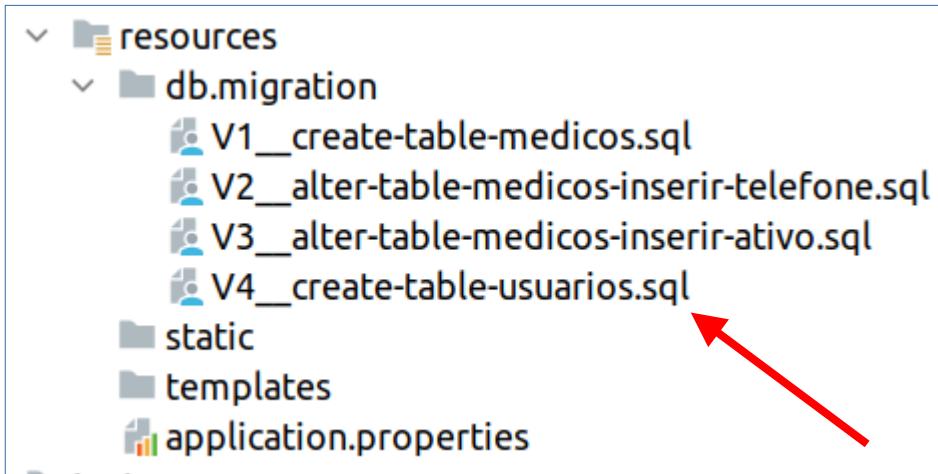
```
1 import jakarta.persistence.*;
2 import lombok.AllArgsConstructor;
3 import lombok.EqualsAndHashCode;
4 import lombok.Getter;
5 import lombok.NoArgsConstructor;
6
7 @Table(name = "usuarios")
8 @Entity(name = "Usuario")
9 @Getter
10 @NoArgsConstructor
11 @AllArgsConstructor
12 @EqualsAndHashCode(of = "id")
13
14 public class Usuario {
15
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private Long id;
19     private String login;
20     private String senha;
21
22 }
23
```

The fields `private Long id;`, `private String login;`, and `private String senha;` are highlighted with a red border.

OBS: Anotações vistas em aulas anteriores.

OBS2: Tá faltando coisa aqui...

# Migration do Banco de Dados, para criar a tabela:

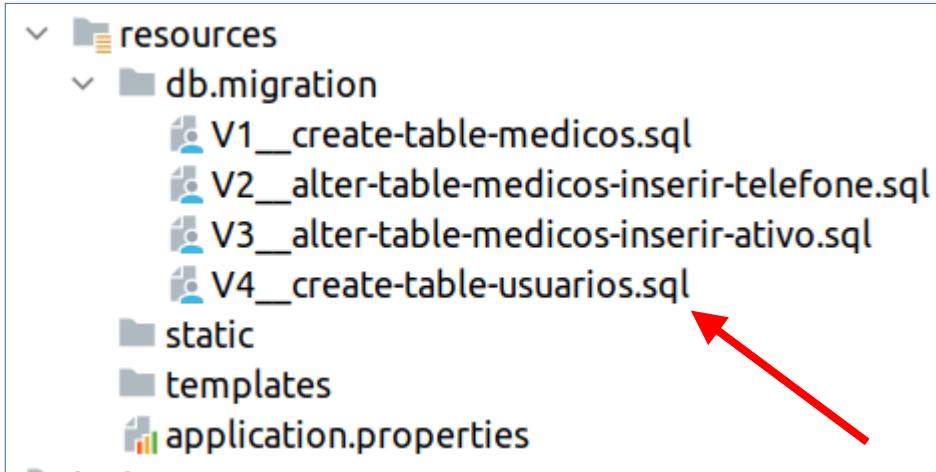


**ATENÇÃO!!!**

**Se você fez a migration para criar o campo UUID, como mostrado na aula anterior, esta será a migration V5.**

```
create table usuarios(  
    id bigint not null auto_increment,  
    login varchar(100) not null,  
    senha varchar(255) not null,  
    primary key(id)  
);
```

# Migration do Banco de Dados, para criar a tabela:



**ATENÇÃO!!!**  
Ao trabalhar com migrations,  
pare a execução do projeto antes!

```
create table usuarios(  
    id bigint not null auto_increment,  
    login varchar(100) not null,  
    senha varchar(255) not null,  
    primary key(id)  
);
```

# Repository para trabalhar com usuários:

```
import org.springframework.data.jpa.repository.JpaRepository;  
  
public interface UsuarioRepository extends JpaRepository <Usuario, Long> {  
}
```



# ACOMPANHAMENTO DOS PASSOS

- Adicionado o Spring Security no projeto.
  - Por padrão, já bloqueia tudo.
- Criada a classe/entidade que vai representar um usuário de nossa API
  - Classe Usuario / Migration do BD / Repository

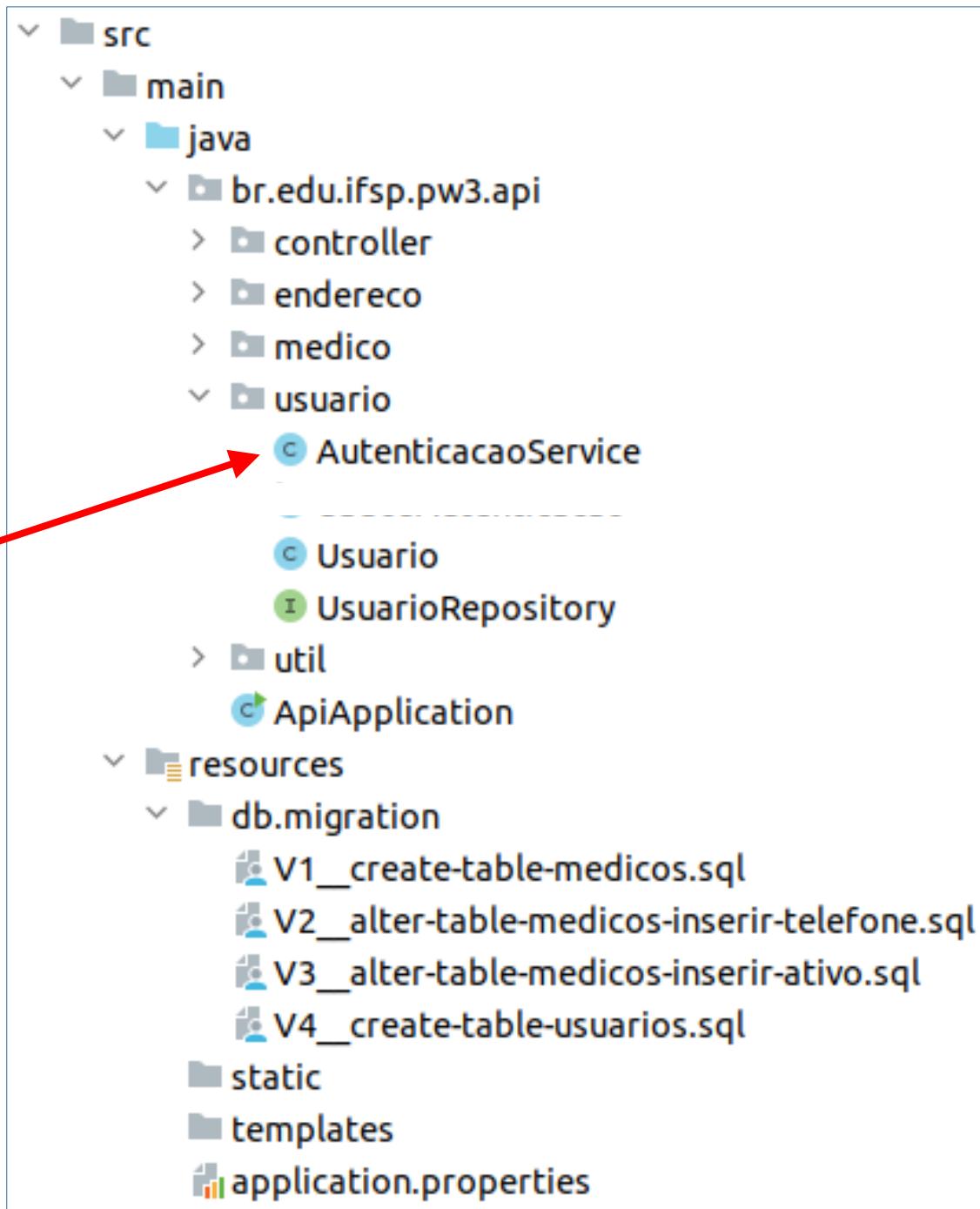


Classe AutenticacaoService:

Vai ter a lógica de autenticação do projeto.

OBS:

Coloquei no package do usuário  
(a organização dos packages  
não é exatamente a melhor possível...)



```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class AutenticacaoService implements UserDetailsService {

    @Autowired
    private UsuarioRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {

        return repository.findByLogin(username);

    }

}
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service ←
public class AutenticacaoService implements UserDetailsService {

    @Autowired
    private UsuarioRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {

        return repository.findByLogin(username);

    }

}
```

# Antes de @Service...

## Beans



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

# O que são Beans no Spring Framework

Beans são objetos gerenciados pelo contêiner Spring IoC (Inversion of Control). Eles são os componentes fundamentais de uma aplicação Spring e possuem características específicas:

## Características principais dos Beans:

- 1. Objetos gerenciados:** São instanciados, configurados e gerenciados pelo contêiner Spring, não pelo seu código diretamente.
- 2. Ciclo de vida controlado:** O Spring controla quando um bean é criado, inicializado, utilizado e destruído.
- 3. Escopo definido:** Cada bean tem um escopo específico que determina seu tempo de vida e visibilidade (singleton, prototype, request, etc.).
- 4. Injeção de dependências:** Os beans podem ter suas dependências injetadas automaticamente pelo Spring, facilitando o desacoplamento entre componentes.
- 5. Configuração centralizada:** São definidos de forma centralizada, seja por anotações, XML ou código Java.



# Explicação da Anotação @Service no Spring Framework

A anotação `@Service` que você está usando em sua classe `AutenticacaoService` é uma das anotações estereotipadas do Spring Framework, que faz parte do mecanismo de injeção de dependências do Spring.

## O que a anotação @Service faz:

- 1. Marca a classe como um componente de serviço:** Indica que a classe pertence à camada de serviço da aplicação, responsável por conter lógica de negócios.
- 2. Habilita detecção automática de componentes:** Quando o Spring escaneia os pacotes da aplicação, ele identifica automaticamente classes marcadas com `@Service` e as registra como beans no contexto da aplicação.
- 3. Permite injeção:** Uma vez registrada como um bean, a classe `AutenticacaoService` pode ser injetada em outras classes usando `@Autowired`, assim como você está injetando o `UsuarioRepository` na sua classe.



## Arquitetura em camadas:

O Spring Framework promove uma arquitetura em camadas, onde cada anotação estereotipada indica a função específica do componente:

- `@Controller`: Componentes da camada de apresentação (controladores)
- `@Service`: Componentes da camada de serviço (lógica de negócio)
- `@Repository`: Componentes da camada de persistência (acesso a dados)
- `@Component`: Componentes genéricos

No seu caso, a classe `AutenticacaoService` implementa a interface `UserDetailsService` do Spring Security, fornecendo o método `loadUserByUsername` que busca um usuário pelo login, sendo assim corretamente classificada como um serviço.

Tecnicamente, todas essas anotações estereotipadas são especializações de `@Component` e funcionam de maneira semelhante para o registro de beans, mas usar a anotação específica `@Service` ajuda a manter uma boa organização de código e expressa a intenção da classe claramente.



```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class AutenticacaoService implements UserDetailsService {
    @Autowired
    private UsuarioRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        return repository.findByLogin(username);
    }
}
```



# Interface UserDetailsService

- A interface UserDetailsService é usada para **carregar detalhes de um usuário durante o processo de autenticação**.

Ela define um único método:

- UserDetails **loadUserByUsername(String username)**  
throws UsernameNotFoundException;
- Propósito e uso da **interface UserDetailsService**:
  - **Carregar detalhes do usuário**: O principal objetivo da interface UserDetailsService é fornecer um método, **loadUserByUsername**, que carrega detalhes do usuário com base no nome de usuário (username) fornecido como parâmetro.
    - **Esses detalhes geralmente incluem informações como nome de usuário, senha, papéis (roles) e outras informações relevantes para a autenticação e autorização.**
  - **Tratamento de exceções**: O método loadUserByUsername também lança uma exceção do tipo UsernameNotFoundException se o usuário com o nome de usuário especificado não for encontrado.
    - Isso permite ao Spring Security tratar casos em que o usuário não existe na base de dados ou em algum mecanismo de autenticação.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class AutenticacaoService implements UserDetailsService {

    @Autowired
    private UsuarioRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {

        return repository.findByLogin(username);
    }
}
```

“Repository” injetado aqui automaticamente,  
usado para acessar a tabela de usuários no banco de dados.  
(Lembrando o comentário que fizemos na aula passada...)

```
@Service
public class AutenticacaoService implements UserDetailsService {
    @Autowired
    private UsuarioRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        return repository.findByLogin(username);
    }
}
```

A classe `UserDetails` faz parte do Spring Security e é usada para representar os detalhes de um usuário durante o processo de autenticação e autorização. Ela é uma interface que define os métodos e atributos comuns associados a um usuário. A classe `UserDetails` geralmente é implementada por outras classes que representam um usuário específico no contexto de uma aplicação.

No seu código, a classe `AutenticacaoService` usa um repositório de usuários (`UsuarioRepository`) para buscar um usuário com base no nome de usuário fornecido e, em seguida, retorna os detalhes desse usuário encapsulados em um objeto que implementa a interface `UserDetails`. Esses detalhes do usuário, como nome de usuário, senha e autorizações, serão usados pelo Spring Security para autenticar o usuário e determinar suas permissões na aplicação.

```
@Service
public class AutenticacaoService implements UserDetailsService {

    @Autowired
    private UsuarioRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        return repository.findByLogin(username);
    }
}
```

A classe `UserDetails` faz parte do Spring Security e é usada para representar os detalhes de um usuário durante o processo de autenticação e autorização. Ela é uma interface que define os métodos e atributos comuns associados a um usuário. A classe `UserDetails` geralmente é implementada por outras classes que representam um usuário específico no contexto de uma aplicação.

**ISTO AINDA  
NÃO ESTÁ FEITO!  
Nossa classe  
Usuário tem que  
implementar  
UserDetails.  
Veremos a frente!**

No seu código, a classe `AutenticacaoService` usa um repositório de usuários (`UsuarioRepository`) para buscar um usuário com base no nome de usuário fornecido e, em seguida, retorna os detalhes desse usuário encapsulados em um objeto que implementa a interface `UserDetails`. Esses detalhes do usuário, como nome de usuário, senha e autorizações, serão usados pelo Spring Security para autenticar o usuário e determinar suas permissões na aplicação.

```
@Service
public class AutenticacaoService implements UserDetailsService {

    @Autowired
    private UsuarioRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {

        return repository.findByLogin(username);
    }
}
```



Precisamos criar esse método em **UsuarioRepository**.

## Resumo do método:

- Ele é responsável por **carregar informações de usuários** a partir de uma fonte de dados (como um banco de dados) e **retornar um objeto** do Spring Security chamado **UserDetails**.
- UserDetailsService: A interface UserDetailsService é parte do Spring Security e define um único **método chamado loadUserByUsername**. Este método é responsável por **carregar um usuário com base no nome de usuário** (geralmente o nome de usuário fornecido durante a tentativa de login). O método loadUserByUsername deve **retornar um objeto** que implementa a interface **UserDetails**.
- **UserDetails**: A interface UserDetails é outra parte do Spring Security e **representa as informações de um usuário autenticado**. Ela contém detalhes sobre o usuário, como nome de usuário, senha, papéis (ou funções) do usuário e outras informações relacionadas à autenticação e autorização.

## Resumo da classe:

- A principal função da classe AutenticacaoService é implementar o método loadUserByUsername. **Quando um usuário tenta fazer login em sua aplicação, o Spring Security chama esse método, passando o nome de usuário como argumento.** Dentro desse método, você normalmente faz uma consulta ao banco de dados ou outra fonte de dados para recuperar as informações do usuário com base no nome de usuário fornecido.
- Em seguida, você **cria e retorna um objeto UserDetails** que contém as informações do usuário.
- O Spring Security utiliza essas informações para verificar a senha do usuário, determinar os papéis do usuário (para controle de autorização) e realizar outras operações de segurança.
- A anotação @Service é usada para marcar a classe como um componente gerenciado pelo Spring (um bean), o que permite que o Spring a injete onde for necessário, como em configurações de segurança.

Precisamos inserir o método necessário  
no Repository:

# Repository para trabalhar com usuários:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.security.core.userdetails.UserDetails;

public interface UsuarioRepository extends JpaRepository <Usuario, Long> {

    UserDetails findByLogin(String login);

}
```

O Spring Data JPA fornece uma funcionalidade chamada "consulta automática" (automatic query creation) que permite que você crie consultas simples de busca de registros com base nos nomes dos métodos definidos em uma interface de repositório, desde que sigam uma convenção específica de nomenclatura.

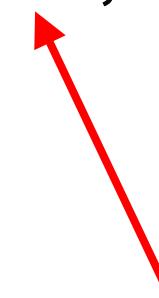
No seu caso, o método `**findByLogin**` na interface `**UsuarioRepository**` segue a convenção de nomenclatura do Spring Data JPA. O Spring Data JPA analisará o nome do método (`**findByLogin**`), entenderá que você deseja encontrar um registro com base no campo `**login**` da entidade `**Usuario**`, e gerará automaticamente a consulta SQL correspondente.

Portanto, sim, o método `**findByLogin**` é gerado automaticamente pelo Spring Data JPA e realizará a busca de um registro com base no valor fornecido no campo `**login**`. Esta é uma maneira poderosa e conveniente de criar consultas simples em repositórios do Spring Data JPA, economizando tempo e esforço na escrita de consultas SQL manualmente.



# Repository para trabalhar com usuários:

```
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.security.core.userdetails.UserDetails;  
  
public interface UsuarioRepository extends JpaRepository <Usuario, Long> {  
    UserDetails findByLogin(String login);  
}
```

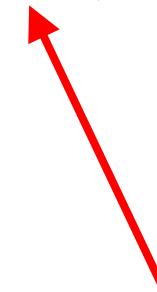


Pq retorna um UserDetails ? Não deveria retornar um Usuario ?



# Repository para trabalhar com usuários:

```
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.security.core.userdetails.UserDetails;  
  
public interface UsuarioRepository extends JpaRepository <Usuario, Long> {  
    UserDetails findByLogin(String login);  
}
```



Pq retorna um UserDetails ? Não deveria retornar um Usuario ?

Sim, deveria... e vai.

Como já falei, mais a frente vamos fazer nossa classe Usuario implementar a interface UserDetails.



# ACOMPANHAMENTO DOS PASSOS

- Adicionado o Spring Security no projeto.
  - Por padrão, já bloqueia tudo.
- Criada a classe/entidade que vai representar um usuário de nossa API
  - Classe Usuario / Migration do BD / UsuarioRepository
    - Adicionamos o método **UserDetails findByLogin(String login)** no UsuarioRepository.
- **Criada a classe que representa o “serviço” de autenticação de usuários**
  - Usa o repository criado acima
  - Implementa método loadUserByUsername, que retorna um UserDetails.



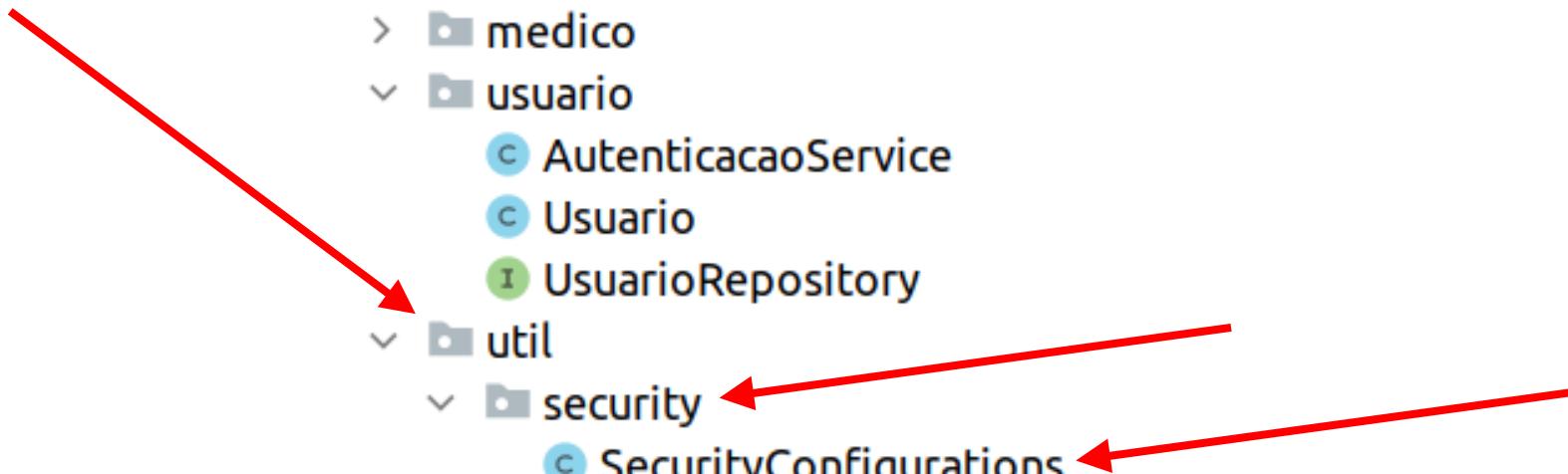
# Configuração para tornar o processo “StateLess”

Criação de classe para realizar configurações relacionadas com autenticação.

## SecurityConfigurations



```
src
└── main
    ├── java
    │   └── br.edu.ifsp.pw3.api
    │       ├── controller
    │       ├── endereco
    │       ├── medico
    │       └── usuario
    │           ├── AutenticacaoService
    │           ├── Usuario
    │           └── UsuarioRepository
    └── util
        └── security
            ├── SecurityConfigurations
            ├── TratadorDeErros
            └── ApiApplication
    └── resources
        ├── db.migration
        ├── static
        ├── templates
        └── application.properties
```



```
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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf(csrf -> csrf.disable())
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
    }
}
```

```
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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf(csrf -> csrf.disable())
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
    }
}
```

# Explicação da Anotação @Bean no Spring Framework

A anotação `@Bean` é um componente fundamental do Spring Framework para configuração baseada em Java.

## O que é a anotação @Bean:

A anotação `@Bean` é um marcador a nível de método que indica que o método produz um objeto que deve ser registrado como um bean no contêiner Spring. Diferente das anotações estereotipadas como `@Service` ou `@Repository` que são aplicadas em classes, `@Bean` é usada em métodos dentro de classes marcadas com `@Configuration`.



```
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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean ←
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        return http.csrf(csrf -> csrf.disable())
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
    }
}
```

O objetivo específico da anotação `@Bean` no método `securityFilterChain` é:

- 1. Registrar a cadeia de filtros:** Criar e registrar um bean `SecurityFilterChain` no contêiner Spring, que define como a segurança da aplicação web deve funcionar.
- 2. Personalizar a configuração de segurança:** Fornecer uma configuração customizada para o Spring Security, em vez de usar a configuração padrão.
- 3. Configuração programática:** Permitir definir regras de segurança de forma programática usando a API fluente do `HttpSecurity`.
- 4. Injeção automática do HttpSecurity:** O parâmetro `HttpSecurity http` é automaticamente injetado pelo Spring quando o método é chamado.

# Como criar beans ?

- **@Component:**
  - **Marca uma classe** como um componente genérico gerenciado pelo Spring.
- **@Service:**
  - Uma especialização de **@Component**. Usada para **marcar classes** que representam serviços na camada de serviço.
- **@Repository:**
  - Uma especialização de **@Component**. Usada para **marcar classes** que representam repositórios de dados.
- **@Controller:**
  - Uma especialização de **@Component**. Usada para **marcar classes** que controlam o fluxo de requisições em uma aplicação web.
- **Essas anotações são detectadas automaticamente** pelo Spring IoC Container durante a varredura de componentes (component scanning) e **são utilizadas para criar e gerenciar beans** no contexto da aplicação.

# Como criar beans ?

- **@Component:**

- **Marca uma classe** como um componente genérico gerenciado pelo Spring.

- **@Service:**

- Uma especialização de **@Component**. Usada para **marcar classes** que representam serviços na camada de serviço.

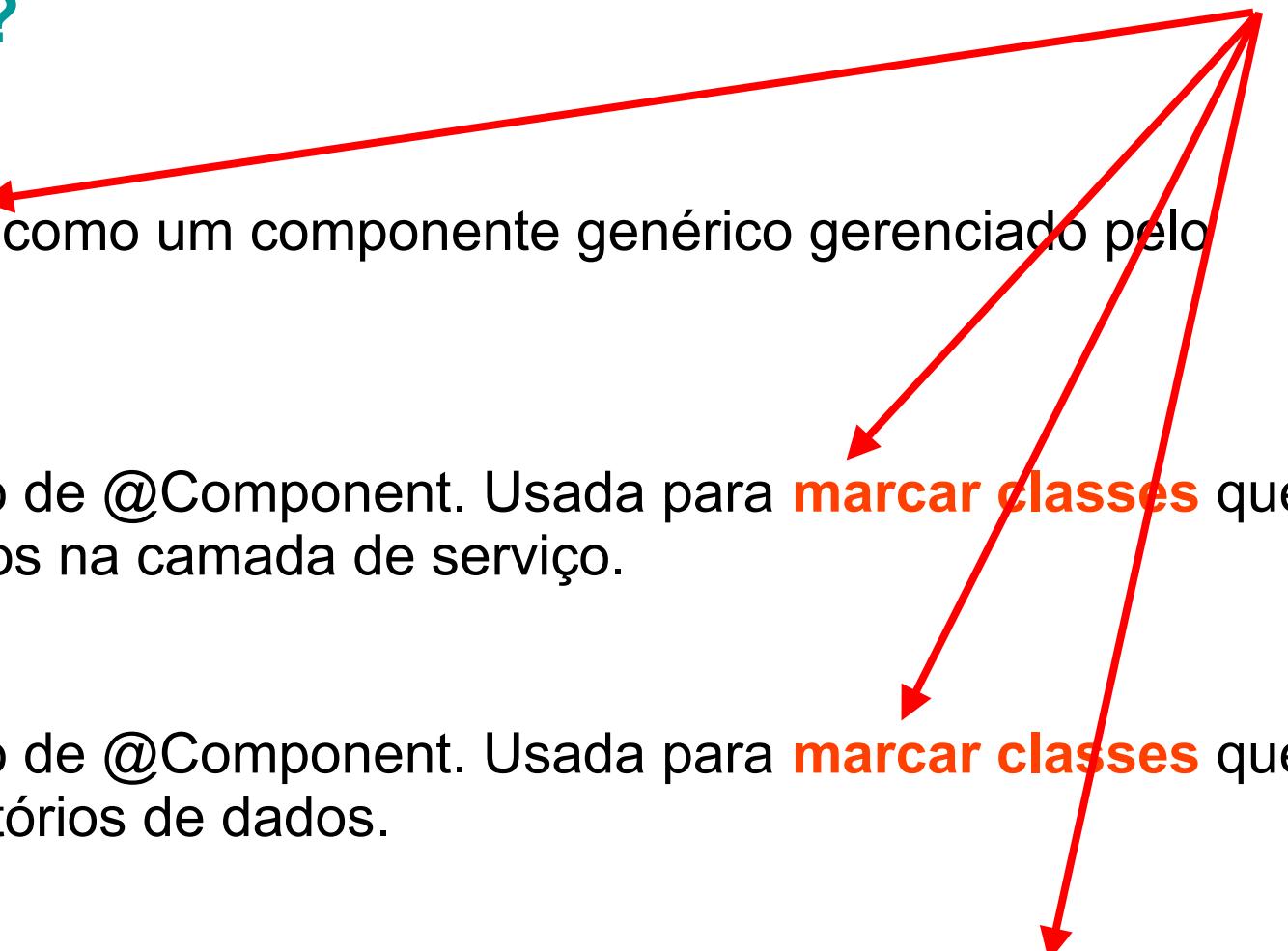
- **@Repository:**

- Uma especialização de **@Component**. Usada para **marcar classes** que representam repositórios de dados.

- **@Controller:**

- Uma especialização de **@Component**. Usada para **marcar classes** que controlam o fluxo de requisições em uma aplicação web.

- **Essas anotações são detectadas automaticamente** pelo Spring IoC Container durante a varredura de componentes (component scanning) e **são utilizadas para criar e gerenciar beans** no contexto da aplicação.



## @Bean

- Embora as anotações anteriores sejam convenientes e funcionem, elas são anotações de **nível de classe**.
- **E se você precisar criar um bean a partir de uma chamada de método?**
- A anotação **@Bean** é usada para declarar um bean no Spring.
- **Quando aplicada a um método**, essa anotação especifica que o método retorna um bean que deve ser gerenciado pelo contêiner Spring.
- **Métodos anotados com @Bean são declarados dentro de classes anotadas com @Configuration.**



# Diferença entre @Service e @Bean no Spring Framework

As anotações `@Service` e `@Bean` são fundamentais no Spring, mas servem a propósitos diferentes e são aplicadas de maneiras distintas.

## Diferenças principais

Característica	<code>@Service</code>	<code>@Bean</code>
<b>Nível de aplicação</b>	Anotação a nível de <b>classe</b>	Anotação a nível de <b>método</b>
<b>Classe alvo</b>	Aplicada em classes de serviço	Aplicada em métodos dentro de classes <code>@Configuration</code>
<b>Finalidade</b>	Marca uma classe como componente da camada de serviço	Define explicitamente um método que produz um bean
<b>Detecção</b>	Detectada pelo component-scan automático	Processada durante a inicialização da classe de configuração
<b>Controle</b>	A classe inteira se torna um bean	Permite controle preciso sobre a criação e configuração do objeto
<b>Escopo</b>	Define a própria classe como bean	Define o valor retornado pelo método como bean



```
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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration ←
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf(csrf -> csrf.disable())
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
    }
}
```

A anotação `@Configuration` é usada no Spring Framework para indicar que uma classe é uma classe de configuração. Isso significa que a classe contém definições de beans e configurações que serão gerenciadas pelo contêiner Spring. Quando você marca uma classe com `@Configuration`, o Spring Framework a reconhece como uma fonte de configuração e a utiliza para construir e inicializar os componentes do seu aplicativo.

No contexto do seu código, a classe `SecurityConfigurations` é marcada com `@Configuration`, o que significa que ela é responsável por configurar a segurança da sua aplicação. Ela define um bean chamado `securityFilterChain` que é um componente de segurança responsável por configurar aspectos relacionados à segurança do aplicativo.

A anotação `@Configuration` é frequentemente usada em conjunto com outras anotações, como `@Bean`, para definir e configurar beans específicos que são usados na aplicação. Neste caso, a classe `SecurityConfigurations` configura a segurança da aplicação usando o Spring Security, e o método `securityFilterChain` define a configuração do filtro 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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf(csrf -> csrf.disable())
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
    }
}
```



A anotação `@EnableWebSecurity` é uma anotação fornecida pelo Spring Security, que é uma parte do Spring Framework usada para configurar a segurança em aplicativos web. Quando você anota uma classe de configuração com `@EnableWebSecurity`, está sinalizando ao Spring Framework que essa classe é responsável por configurar a segurança da sua aplicação web usando o Spring Security.

Em sua classe `SecurityConfigurations`, ao usar `@EnableWebSecurity`, você está indicando que essa classe é onde você deseja definir as configurações de segurança específicas para a sua API REST. As configurações de segurança podem incluir regras de autorização, autenticação, configurações de sessão, proteção contra CSRF (Cross-Site Request Forgery) e muito mais.

A anotação `@EnableWebSecurity` é uma parte fundamental da configuração do Spring Security e é geralmente usada no início da classe de configuração da segurança. Ela ativa o suporte do Spring Security e permite que você defina as configurações necessárias para proteger sua aplicação web de maneira adequada.



```
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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf(csrf -> csrf.disable())
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
    }
}
```



A anotação `@Bean` é usada no contexto do Spring Framework para indicar que um método em uma classe de configuração retorna um objeto que deve ser gerenciado e configurado como um bean pelo contêiner Spring. Essa anotação permite que você defina manualmente objetos específicos como beans e os disponibilize para serem usados em outras partes de sua aplicação.

No seu código, o método `securityFilterChain` é anotado com `@Bean`. Isso significa que o método retorna um objeto que será configurado e gerenciado como um bean pelo Spring Framework. Nesse caso, o objeto retornado é um `SecurityFilterChain`, que é usado para definir a configuração de segurança no contexto do Spring Security.

A anotação `@Bean` é frequentemente usada em classes de configuração, onde você pode definir os beans necessários para configurar diferentes partes do seu aplicativo. Esses beans podem ser componentes, serviços, gerenciadores de segurança, entre outros. O Spring cuidará de criar, configurar e injetar esses beans em outras partes do seu aplicativo, conforme necessário.



```
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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf(csrf -> csrf.disable())
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
    }
}
```

A classe '**SecurityFilterChain**' representa uma configuração de filtro de segurança em uma aplicação Spring Security. Ela é usada para definir a configuração de como os filtros de segurança devem se comportar ao processar solicitações HTTP em um aplicativo web protegido pelo Spring Security.



```
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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf(csrf -> csrf.disable())
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
    }
}
```

O parâmetro `HttpSecurity` é um objeto que você pode usar para configurar a segurança de sua aplicação Spring usando o Spring Security. Ele é passado como um parâmetro para o método `securityFilterChain` na classe de configuração `SecurityConfigurations`, conforme mostrado em seu código.

O objeto `HttpSecurity` é uma parte central da configuração do Spring Security. Ele fornece uma interface fluente que permite que você defina regras de segurança, configure permissões de acesso, lide com autenticação e autorização, proteja rotas e recursos e muito mais.

```
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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf(csrf -> csrf.disable())
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
    }
}
```



```

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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf(csrf -> csrf.disable()
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build());
    }
}

```

Uma expressão lambda é uma função anônima (sem nome) que permite escrever código mais conciso. Ela tem três partes:

1. Parâmetro(s) -> os valores de entrada
2. Seta -> (operador lambda)
3. Corpo -> o código a ser executado

`csrf -> csrf.disable()` é uma expressão lambda onde:

- `csrf ->` é o parâmetro da lambda (um objeto que configura CSRF)
- `csrf.disable()` é o corpo da lambda que desativa a proteção CSRF



# CSRF

CSRF (Cross-Site Request Forgery), em português, Falsificação de Solicitação entre Sites, é um tipo de ataque de segurança em que um invasor engana um usuário para realizar ações indesejadas em um aplicativo no qual o usuário já está autenticado. Isso é feito aproveitando a confiança que o aplicativo tem no navegador do usuário.

Aqui está um exemplo simplificado de como um ataque CSRF funciona:

1. Um usuário autenticado em um aplicativo (como um banco online) possui um cookie de sessão válido.
2. O usuário visita uma página maliciosa ou é direcionado para ela, possivelmente através de um link em um e-mail ou uma página da web comprometida.
3. A página maliciosa contém código HTML ou JavaScript que emite uma solicitação HTTP para o aplicativo (por exemplo, transferir dinheiro para a conta do atacante) em segundo plano, sem o conhecimento do usuário.
4. O navegador do usuário, que inclui automaticamente os cookies de sessão válidos para o domínio de destino, envia a solicitação ao aplicativo, executando a ação maliciosa em nome do usuário autenticado.
5. O ataque é bem-sucedido, e o dinheiro é transferido para a conta do atacante, por exemplo, sem que o usuário perceba.

```
return http.csrf(csrf -> csrf.disable())
    .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
    .build();
```

O trecho "http.csrf(csrf -> csrf.disable())" é uma parte da configuração de segurança do Spring Security que desabilita a proteção contra CSRF (Cross-Site Request Forgery) para sua aplicação. Vamos analisar em detalhes o que cada parte desse código faz:

1. `http.csrf(...)`: `http` é uma instância de `HttpSecurity`, que permite a configuração de segurança em sua aplicação. A chamada `http.csrf(...)` inicia a configuração das medidas de proteção contra CSRF.
2. `csrf -> csrf.disable()`: Este é um lambda ou função anônima que desabilita a proteção CSRF. O objeto `csrf` é uma instância de `HttpSecurity.CsrfConfigurer`, que fornece métodos para configurar as opções de CSRF. No código, ele chama o método `disable()` para desativar a proteção CSRF.

A proteção contra CSRF é uma medida de segurança que protege sua aplicação contra ataques nos quais um invasor tenta forçar um usuário a executar ações não autorizadas em seu nome, aproveitando as sessões autenticadas do usuário. Desativar o CSRF é apropriado em casos em que sua aplicação é uma API REST e não depende de sessões ou formulários da web para autenticação e autorização, e em que você usa tokens (como tokens JWT) para autenticar as solicitações em vez de sessões.

```
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.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf(csrf -> csrf.disable()
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build());
    }
}
```

O trecho `.sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))` no seu código está configurando a política de gerenciamento de sessão para a segurança da sua aplicação Spring. Especificamente, está definindo a política de criação de sessão como `'SessionCreationPolicy.STATELESS'`.

Quando a política de criação de sessão é definida como `'SessionCreationPolicy.STATELESS'`, isso significa que sua aplicação não manterá sessões de usuário no servidor. Isso é comum em aplicações RESTful, onde a autenticação é baseada em tokens, como tokens JWT (JSON Web Tokens), em vez de sessões de usuário.

```
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
    return http.csrf(csrf -> csrf.disable())  
        .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))  
        .build();  
}
```

O método `.build()` no final do trecho de código que você forneceu é usado para concluir a configuração de segurança definida com `HttpSecurity` e construir um objeto `SecurityFilterChain`.

Em termos simples, o `SecurityFilterChain` representa a configuração de segurança para a aplicação Spring Security. Ele é composto por uma série de filtros de segurança que definem como as solicitações HTTP devem ser tratadas em termos de autenticação, autorização e outras medidas de segurança.

O método `.build()` é usado para finalizar a configuração do `HttpSecurity` e criar um `SecurityFilterChain` a partir dessa configuração. Em outras palavras, ele encapsula a configuração que você definiu com `HttpSecurity` e a transforma em um objeto que pode ser usado pela aplicação Spring Security para proteger as solicitações HTTP.

Então, no seu código, o `.build()` está retornando o objeto `SecurityFilterChain` que contém a configuração de segurança especificada, incluindo a desativação do CSRF e a definição da política de criação de sessão como `STATELESS`. Essa configuração será usada para proteger as solicitações HTTP em sua aplicação.

# ACOMPANHAMENTO DOS PASSOS

- Adicionado o Spring Security no projeto.
  - Por padrão, já bloqueia tudo.
- Criada a classe/entidade que vai representar um usuário de nossa API
  - Classe Usuario / Migration do BD / UsuarioRepository
    - Adicionamos o método **UserDetails findByLogin(String login)** no UsuarioRepository.
- Criada a classe que representa o “serviço” de autenticação de usuários
  - Usa o repository criado acima
  - Implementa método loadUserByUsername, que retorna um UserDetails.
- **Criada a classe que define configurações de segurança.**
  - util | security | SecurityConfigurations
  - desligou csrf | definiu autenticação stateless



Vamos criar um usuário no BD,  
para testes.

Porém não é boa prática de segurança  
armazenar a senha diretamente  
no BD.

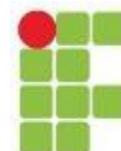
Vamos utilizar um algoritmo de hashing de senhas.

Vamos usar o algoritmo Bcrypt.



# Algoritmo BCrypt

O algoritmo Bcrypt é um algoritmo de hash de senhas que é amplamente usado para armazenar senhas de forma segura. Ele aplica uma função de hash lenta e segura, projetada para ser resistente a ataques de força bruta. O Bcrypt utiliza uma técnica chamada "salting" (adicionar um valor aleatório à senha antes do hash) para tornar as senhas mais seguras. Além disso, ele é capaz de ajustar automaticamente o custo do algoritmo, tornando-o mais resistente ao longo do tempo à medida que os computadores ficam mais poderosos. Por causa de sua segurança e eficácia, o Bcrypt é uma escolha comum para proteger senhas em aplicativos e sistemas.



# Convertendo a senha '123456'

The screenshot shows a web browser window for 'Bcrypt Generator - Online Hash Generator & Checker' at [bcrypt-generator.com](https://bcrypt-generator.com). The page is titled 'Bcrypt Hash Generator' and describes it as a simple tool to generate and verify bcrypt hashes, processing in the browser for security.

**Generate Hash:** A form where users input text to hash and select rounds (cost factor). The input field contains '123456' and the slider is set to 12 rounds, described as 'High security - suitable for production'. The generated bcrypt hash is '\$2a\$12\$n72204vH2KRv0co0Ivbvs.jExERlWrTwEb/LsNwVe7lWrTwEb/LsNwVe7hBYfaMEMEeu', noted as being generated with 12 rounds of hashing.

**Verify Hash:** A form where users input the bcrypt hash and the original text. The hash input field contains '\$2a\$12\$n72204vH2KRv0co0Ivbvs.jExERlWrTwEb/LsNwVe7lWrTwEb/LsNwVe7hBYfaMEMEeu' and the original text input field contains '123456'. A red arrow points from the original text input field to the 'Verify Hash' button. The result is a green box stating '✓ Hash matches the text'.

Lembrando...  
(matéria de Estrutura de Dados geralmente...)  
o algoritmo de hash é unidirecional,  
isto é,  
vai da senha para o hash,  
e não o contrário.

Por isso podemos colocar o hash no BD,  
pois saber ele,  
não tem como saber a senha original que o gerou.

Você só consegue CONFERIR  
que a senha está correta, baseada no hash  
armazenado no BD.

Vamos criar um usuário na tabela de usuários:

login: asdrubal@gmail.com  
senha: 123456 (**encriptada**)

Como estou usando o H2,  
vou tentar usar o h2-console:

English

Preferences Tools Help

## Login

Saved Settings:

Setting Name:

---

Driver Class:

JDBC URL:

User Name:

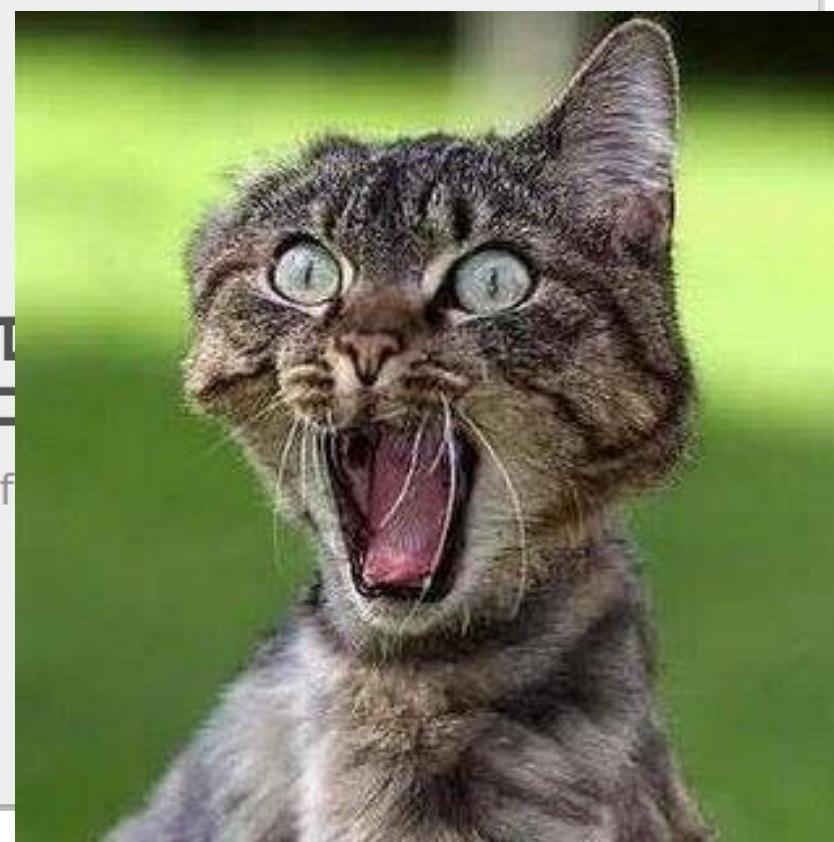
Password:



**localhost** refused to connect.



localhost ref



Como ativamos o Spring Security  
no nosso projeto,  
e o h2-console é acessado através  
do mesmo Tomcat do Spring,  
o Spring Security bloqueou o acesso  
ao h2-console.

## Soluções possíveis:

- autorizar h2-console na classe de configuração que acabamos de fazer (muitos comandos para um objetivo muito específico... não vou gastar tempo com isso)
- criar uma migration para inserir o registro no banco de dados.
- usar um programa externo para inserir registros em tabelas.

## OBS:

Isso tudo pq estou usando o H2.  
Se você está usando outro banco  
(mysql, postgress, oracle)  
use a interface de administração  
do seu banco.

No início da disciplina,  
utilizamos um console JAR  
para visualizar dados no banco H2.

Poderia ser usado pra inserir o usuário na tabela.

Existem outros programas externos para  
manipular bancos H2:



## Menu

[Home](#)  
[Download and Installation](#)  
[Plugins](#)  
[Translations](#)  
[Introduction, Screenshots](#)  
[Programming](#)  
[FAQ](#)  
[Old Versions](#)

## On SourceForge

[Project Home](#)  
[Downloads](#)  
[News](#)  
[Mailing Lists](#)  
[Bugs](#)  
[Feature Requests](#)  
[GIT](#)

## On GitHub

[Project Home](#)  
[Stable downloads](#)  
[Snapshot downloads](#)  
[Bugs/Feature requests](#)  
[GIT](#)

[New Features](#) [Latest News](#) [Overview](#) [Download and Installation](#) [Introduction, Screenshots](#) [Programming](#) [Mailing Lists](#) [GIT](#)

## Feature highlights of 4.6.0:

- Data compare function
- Search function for preferences/configurations
- Enhanced BLOB and CLOB loading configuration
- Export of multiple SQL results
- Java 20 compatibility
- Many more features and bug fixes

All new features and bug fixes of the 4.6.0 release can be found in our change log at [SourceForge](#) or [GitHub](#).

[Download SQuirreL SQL Client](#)

Support us:



Support this  
project

Some people recently experienced that the support button links to the PayPal page in German.  
This button now definitely links to the English PayPal page:

[Donate](#)

[Review SQuirreL on Sourceforge](#)



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos



# DBeaver Community

Free Universal Database Tool

[Home](#) [About](#) [Download](#) [Documentation](#) [News](#) [Support](#) [DBeaver PRO](#) [CloudBeaver](#) [DBeaver Merch](#)

## Universal Database Tool

DBeaver Community is a free cross-platform database tool for developers, database administrators, analysts, and everyone working with data. It supports all popular SQL databases like MySQL, MariaDB, PostgreSQL, SQLite, Apache Family, and more.

The screenshot shows the DBeaver interface with several panes:

- Left pane (Database Browser):** Shows a tree view of database objects. Nodes include: **System Info**, **PostgreSQL - Chinook** (with tables like Address, City, State, Country, Value), **Tables** (with sub-nodes like Address, Customer, Employee, etc.), **Views**, **Indices**, **Regressions**, **Table Triggers**, and **Procedures**.
- Center pane (Query Editor):** Displays a table titled "Customer" with columns: Address, City, State, Country, Value. The table contains 80 rows of data. A specific row (customer 23) is highlighted in yellow.
- Right pane (Properties):** Shows detailed information for the selected row (customer 23), including columns: Level, # Type, Title Name, and more.

## DBeaver Community

Open-source version

## DBeaver PRO

Commercial versions

Como falei anteriormente, pode usar o DataGrip,  
~~com sua licença acadêmica.~~

Gratuito pra uso acadêmico!!



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

Vamos usar a solução via migration:  
(bem mais fácil... 😊 )



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

# Solução via migration:

## V6\_\_insere\_asdrubal.sql

A screenshot of a code editor showing a migration file named `V6__insere_asdrubal.sql`. The file contains the following SQL code:

```
1 INSERT INTO usuarios (login, senha)
2 VALUES ('asdrubal@gmail.com', '$2a$12$n72204vH2KRv0co0Ivbvs.jExERlWrTwEb/LsNwVe7hBYfaMEMEeu');
3
```

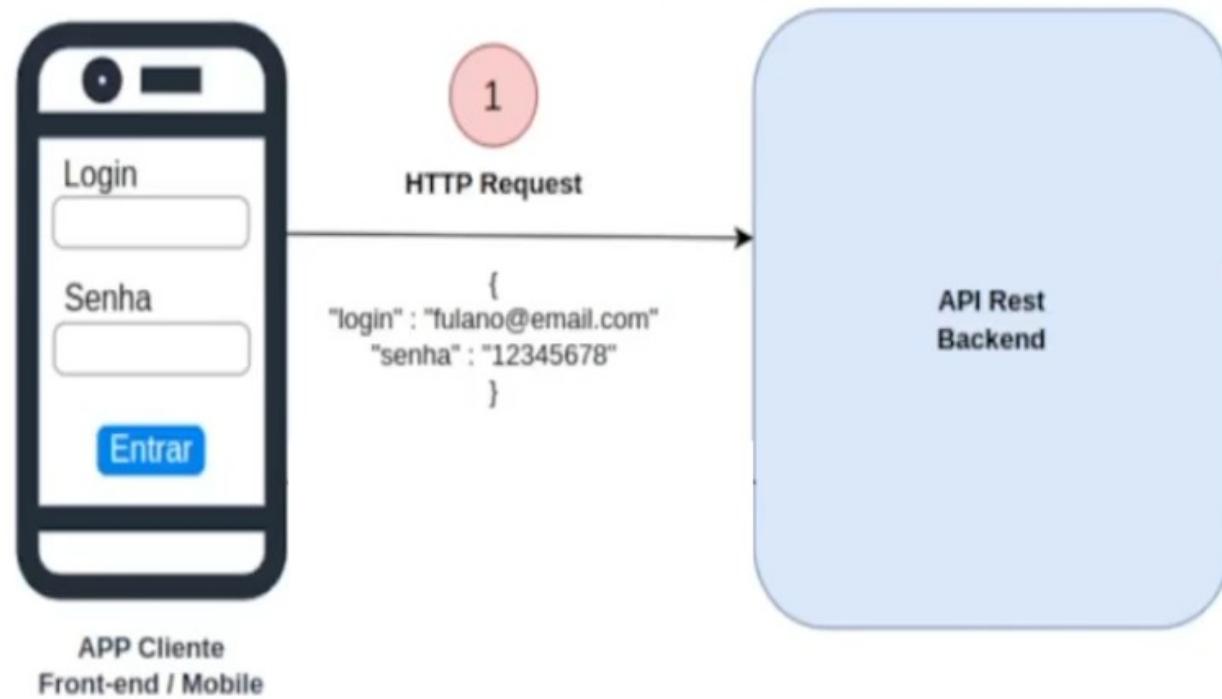
```
INSERT INTO usuarios (login, senha)
VALUES ('asdrubal@gmail.com', '$2a$12$n72204vH2KRv0co0Ivbvs.jExERlWrTwEb/LsNwVe7hBYfaMEMEeu');
```

# ACOMPANHAMENTO DOS PASSOS

- Adicionado o Spring Security no projeto.
  - Por padrão, já bloqueia tudo.
- Criada a classe/entidade que vai representar um usuário de nossa API
  - Classe Usuario / Migration do BD / UsuarioRepository
    - Adicionamos o método **UserDetails findByLogin(String login)** no **UsuarioRepository**.
- Criada a classe que representa o “serviço” de autenticação de usuários
  - Usa o repository criado acima
  - Implementa método loadUserByUsername, que retorna um UserDetails.
- Criada a classe que define configurações de segurança.
  - util | security | SecurityConfigurations
  - desligou csrf | definiu autenticação stateless
- **Criado usuário na tabela de usuários**
  - senha encriptada por BCrypt
  - usando programa externo ou migration (h2 console foi bloqueado).

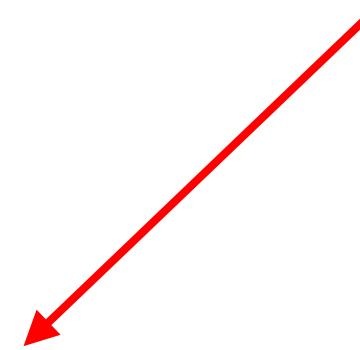
Lembrando como é feita  
a requisição de login:

# Relembrando o processo de autenticação:



Precisamos criar o controller para receber esta requisição.

```
src
└── main
    └── java
        └── br.edu.ifsp.pw3.api
            └── controller
                ├── AutenticacaoController
                ├── HelloController
                └── MedicoController
```



# Iniciando este controller....

// Já vimos essas anotações em códigos anteriores:

```
@RestController  
@RequestMapping("/login")  
public class AutenticacaoController {  
  
    @PostMapping  
    public ResponseEntity efetuarLogin(@RequestBody @Valid dadosAutenticacao dados) {  
  
    }  
}
```

Precisamos criar este DTO...

São os dados que serão  
enviados para fazer login  
(email e senha)

# Iniciando este controller....

// Já vimos essas anotações em códigos anteriores:

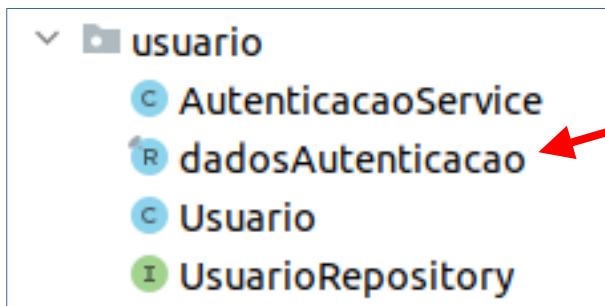
```
@RestController  
@RequestMapping("/login")  
public class AutenticacaoController {  
  
    @PostMapping  
    public ResponseEntity efetuarLogin(@RequestBody @Valid dadosAutenticacao dados) {  
  
    }  
}
```

Precisamos criar este DTO...

São os dados que serão enviados para fazer login (email e senha)

```
13  
14     public class Usuario {  
15  
16         @Id  
17         @GeneratedValue(strategy = GenerationType.IDENTITY)  
18         private Long id;  
19         private String login;  
20         private String senha;  
21     }
```

# DTO (Record): dadosAutenticacao



```
import jakarta.validation.constraints.NotBlank;
```

```
public record dadosAutenticacao(
```

```
    @NotBlank String login,
```

```
    @NotBlank String senha) {
```

```
}
```

Voltando para o controller...

```
@RestController
@RequestMapping("/login")
public class AutenticacaoController {

    @PostMapping
    public ResponseEntity efetuarLogin(@RequestBody @Valid dadosAutenticacao dados) {

        // Estamos recebendo os dados para autenticação em 'dados'.
        //
        // Precisamos consultar este usuário no Banco de Dados.
        //
        // Já temos um método para isso, loadUserByUsername(),
        // lá na classe AutenticacaoService.
        //
        // Mas não somos nós que chamamos diretamente a classe AutenticacaoService.
        //
        // Nós precisamos usar uma classe do próprio Spring,
        // chamada AuthenticationManager, e é esta classe
        // que inicia o processo de autenticação.
        //
        // Vamos colocar um objeto dela como atributo (injetado)
        // aqui nesta classe.

    }

}
```

```
@RestController
@RequestMapping("/login")
public class AutenticacaoController {

    @Autowired
    private AuthenticationManager manager; // Objeto AuthenticationManager,
                                         // que será injetado aqui:

    @PostMapping
    public ResponseEntity efetuarLogin(@RequestBody @Valid dadosAutenticacao dados) {

        var token = new UsernamePasswordAuthenticationToken( dados.login(), dados.senha() );

        var authentication = manager.authenticate(token);

        return ResponseEntity.ok().build();
    }
}
```

Aqui está o que a classe `AuthenticationManager` faz e para que serve:

- 1. Autenticação de Usuários:** O principal propósito do `AuthenticationManager` é autenticar os usuários. Ele recebe uma solicitação de autenticação, que normalmente inclui credenciais como nome de usuário e senha, e verifica se essas credenciais são válidas.
- 2. Validação de Credenciais:** O `AuthenticationManager` realiza a validação das credenciais do usuário, verificando se elas correspondem a um usuário registrado no sistema e se estão corretas.

```
...
@PostMapping
public ResponseEntity efetuarLogin(@RequestBody @Valid dadosAutenticacao dados) {
    var token = new UsernamePasswordAuthenticationToken( dados.login(), dados.senha() );
    var authentication = manager.authenticate(token);
    return ResponseEntity.ok().build();
}
```

Aqui está o que a classe `UsernamePasswordAuthenticationToken` faz e para que serve:

- 1. Representa Credenciais de Autenticação:** A classe `UsernamePasswordAuthenticationToken` é usada para encapsular as credenciais de autenticação de um usuário, que normalmente incluem um nome de usuário (ou identificação) e uma senha.
- 2. Token de Autenticação:** Ela é usada para criar um token que pode ser submetido ao AuthenticationManager do Spring Security para verificar a autenticidade das credenciais.
- 3. Uso em Processo de Autenticação:** Em um cenário típico de autenticação, como o que você apresentou em seu código, um objeto `UsernamePasswordAuthenticationToken` é criado com o nome de usuário e senha fornecidos nos dados de autenticação. Em seguida, esse token é submetido ao `AuthenticationManager` para verificar a autenticidade das credenciais. Se a autenticação for bem-sucedida, o `AuthenticationManager` retorna um objeto de autenticação válido.

```
...
@PostMapping
public ResponseEntity efetuarLogin(@RequestBody @Valid dadosAutenticacao dados) {

    var token = new UsernamePasswordAuthenticationToken( dados.login(), dados.senha() );

    var authentication = manager.authenticate(token);

    return ResponseEntity.ok().build();

}
```

Aqui está o que acontece quando você chama `manager.authenticate(token)`:

1. O `AuthenticationManager`, que foi injetado na classe `AutenticacaoController`, é responsável por coordenar o processo de autenticação. O `AuthenticationManager` é uma parte fundamental do Spring Security e é usado para autenticar usuários.
2. O `manager.authenticate(token)` processa o token de autenticação, que contém as credenciais do usuário.
3. O `AuthenticationManager` verifica as credenciais do usuário, normalmente consultando um repositório de usuários (como um banco de dados) para encontrar o usuário com as credenciais correspondentes.
4. Se as credenciais do usuário forem válidas e a autenticação for bem-sucedida, o `AuthenticationManager` retorna um objeto de autenticação válido. Esse objeto de autenticação inclui informações sobre o usuário autenticado, bem como quaisquer autorizações associadas a esse usuário.
5. Se as credenciais forem inválidas ou a autenticação falhar por qualquer motivo, o `AuthenticationManager` pode lançar uma exceção indicando a falha na autenticação.

```
...
@PostMapping
public ResponseEntity efetuarLogin(@RequestBody @Valid dadosAutenticacao dados) {
    var token = new UsernamePasswordAuthenticationToken( dados.login(), dados.senha() );
    var authentication = manager.authenticate(token);
    return ResponseEntity.ok().build();
}
```

O método `manager.authenticate(token)` retorna um objeto de autenticação, que é uma instância de uma classe que implementa a interface `Authentication` do Spring Security. A classe concreta exata que será retornada depende do processo de autenticação e da configuração da sua aplicação. No entanto, normalmente, o retorno será uma instância de `UsernamePasswordAuthenticationToken` se a autenticação for bem-sucedida.

O objeto de autenticação retornado contém informações sobre o usuário autenticado, como o nome de usuário, as credenciais, as autorizações, entre outros detalhes relacionados à autenticação. Você pode acessar essas informações por meio do objeto de autenticação para tomar decisões sobre o acesso do usuário a recursos protegidos ou realizar ações relacionadas à sessão do usuário.

```
...  
@PostMapping  
public ResponseEntity efetuarLogin(@RequestBody @Valid dadosAutenticacao dados) {  
  
    var token = new UsernamePasswordAuthenticationToken( dados.login(), dados.senha() );  
  
    var authentication = manager.authenticate(token);  
  
    return ResponseEntity.ok().build();  
}  
}
```

Teremos mais coisa por aqui depois....

Um detalhe sobre o atributo

**private AuthenticationManager manager;**

desta classe...

```
@RestController
@RequestMapping("/login")
public class AutenticacaoController {

    @Autowired
    private AuthenticationManager manager; // Objeto AuthenticationManager,
                                         // que será injetado aqui:

    @PostMapping
    public ResponseEntity efetuarLogin(@RequestBody @Valid dadosAutenticacao dados) {

        var token = new UsernamePasswordAuthenticationToken( dados.login(), dados.senha() );

        var authentication = manager.authenticate(token);

        return ResponseEntity.ok().build();
    }
}
```

**ATENÇÃO**



```
@RestController
@RequestMapping("/login")
public class AutenticacaoController {

    @Autowired
    private AuthenticationManager manager; // Objeto AuthenticationManager,
                                         // que será injetado aqui:

    @PostMapping
    public ResponseEntity efetuarLogin(@RequestBody @Valid dadosAutenticacao dados) {

        var token = new UsernamePasswordAuthenticationToken( dados.login(), dados.senha() );

        var authentication = manager.authenticate(token);

        return ResponseEntity.ok().build();
    }
}
```

Apesar desta ser uma classe do próprio Spring,  
ele não vai saber que precisa criar o objeto para injetar aqui.

Precisamos criar um método que fará isso, e exporte  
o objeto como um Bean (como já vimos).

Faremos isso na classe **securityConfigurations**.



# Classe SecurityConfigurations:

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfigurations {  
  
    @Bean  
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
  
        return http.csrf(csrf -> csrf.disable())  
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))  
            .build();  
    }  
  
    @Bean  
    public AuthenticationManager authenticationManager(AuthenticationConfiguration configuration)  
        throws Exception {  
  
        return configuration.getAuthenticationManager();  
    }  
}
```

Cria e devolve o objeto AuthenticationManager, como um Bean, para que possa ser injetado e usado lá no Controller.

# ACOMPANHAMENTO DOS PASSOS

- Adicionado o Spring Security no projeto.
- Criada a classe/entidade que vai representar um usuário de nossa API
  - Classe Usuario / Migration do BD / UsuarioRepository
    - Adicionamos o método **UserDetails findByLogin(String login)** no UsuarioRepository.
- Criada a classe que representa o “serviço” de autenticação de usuários
  - Usa o repository criado acima
  - Implementa método loadUserByUsername, que retorna um UserDetails.
- Criada a classe que define configurações de segurança.
  - util | security | SecurityConfigurations
  - **criado método authenticationManager, que devolve um objeto ‘AuthenticationManager’, como um Bean (para poder ser injetado no controller).**
- Criado usuário na tabela de usuários
  - senha encriptada por BCrypt
- **Criada classe controller para permitir o login: AutenticationController**
  - método efetuarLogin: '/login'
    - criado DTO (Record) para receber login/senha : dadosAutenticacao
  - mas esse método não acessa o BD diretamente, usa um objeto ‘AuthenticationManager’ que é o responsável por ‘disparar’ o serviço que criamos anteriormente, e que realiza esta ação.

# Codificação da Senha

- Mais cedo nesta aula, geramos o hash da senha '123456' usando o algoritmo BCrypt.
- Precisamos indicar ao Spring que vamos usar este algoritmo para (de)codificar as senhas.
- Como se trata de uma configuração de segurança, vamos implementar na classe SecurityConfigurations.



```
@Configuration  
@EnableWebSecurity  
public class SecurityConfigurations {  
  
    @Bean  
    public SecurityFilterChain securityFilterChain(HttpSecurity http) ...  
    ...  
}  
  
@Bean  
public AuthenticationManager authenticationManager( ...  
    ...  
}  
  
@Bean  
public PasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}  
}
```



```
@Configuration  
@EnableWebSecurity  
public class SecurityConfigurations {  
  
    @Bean  
    public SecurityFilterChain securityFilterChain(HttpSecurity http) ...  
    ...  
}  
  
@Bean  
public AuthenticationManager authenticationManager( ...  
    ...  
}  
  
@Bean  
public PasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}  
}
```



No contexto do Spring Framework, a interface `'PasswordEncoder'` é parte do módulo de segurança (`'spring-security-core'`). Essa interface é usada para lidar com a codificação e verificação de senhas, fornecendo uma maneira segura de armazenar senhas em um sistema.

A ideia principal por trás do `'PasswordEncoder'` é evitar o armazenamento direto de senhas em texto simples no banco de dados, aumentando a segurança do sistema. Em vez disso, as senhas são armazenadas em uma forma codificada e, ao verificar a autenticidade durante o login, a senha fornecida pelo usuário é codificada da mesma forma e comparada com a versão armazenada.

O Spring Security fornece várias implementações de `'PasswordEncoder'`, sendo a mais comum a `'BCryptPasswordEncoder'`. O BCrypt é um algoritmo de hashing projetado para ser lento e resistente a ataques de força bruta, tornando as senhas mais seguras.

Ao usar o `'PasswordEncoder'`, você pode garantir que as senhas dos usuários são tratadas com segurança no seu aplicativo Spring. Isso ajuda a proteger as informações confidenciais e a evitar vulnerabilidades relacionadas a senhas.



```
@Configuration  
@EnableWebSecurity  
public class SecurityConfigurations {  
  
    @Bean  
    public SecurityFilterChain securityFilterChain(HttpSecurity http) ...  
  
    @Bean  
    public AuthenticationManager authenticationManager( ...  
  
        @Bean  
        public PasswordEncoder passwordEncoder() {  
            return new BCryptPasswordEncoder();  
        }  
    }  
}
```

A classe `BCryptPasswordEncoder` é uma implementação da interface `PasswordEncoder` no Spring Security que utiliza o algoritmo de hash bcrypt para codificar senhas. O bcrypt é uma função de hash projetada para ser lenta e computacionalmente intensiva, o que a torna uma escolha sólida para armazenar senhas com segurança.

A principal característica do bcrypt é a sua capacidade de ajustar o custo computacional do algoritmo, conhecido como "fator de trabalho" ou "custo". Quanto maior o custo, mais lento é o processo de hashing, tornando mais difícil para os atacantes realizar ataques de força bruta.

# Algoritmo hash unidirecional

Aqui está uma visão geral do processo:

1. **Usuário insere a senha:** Quando um usuário cria ou altera uma senha, a senha é fornecida e passada por um algoritmo de hash.
2. **Geração do hash:** O algoritmo de hash pega a senha como entrada e gera um hash, que é uma sequência de caracteres aparentemente aleatórios de comprimento fixo.
3. **Armazenamento no banco de dados:** Em vez de armazenar a senha real, o sistema armazena apenas o hash no banco de dados, associado ao usuário correspondente.
4. **Verificação durante o login:** Quando o usuário tenta fazer login, o sistema pega a senha fornecida, a submete ao mesmo algoritmo de hash e compara o hash gerado com o hash armazenado no banco de dados.

A força dessa abordagem está na natureza unidirecional dos algoritmos de hash. Mesmo que um atacante obtenha o hash, não deveria ser possível determinar a senha original, especialmente se o algoritmo for robusto e se uma técnica chamada "salting" for aplicada.

# ACOMPANHAMENTO DOS PASSOS

- Adicionado o Spring Security no projeto.
- Criada a classe/entidade que vai representar um usuário de nossa API
- Criada a classe que representa o “serviço” de autenticação de usuários
- Criada a classe que define configurações de segurança.
  - util | security | SecurityConfigurations
  - **criado método para gerar um objeto (@Bean) responsável pela (de)codificação BCrypt**
- Criado usuário na tabela de usuários
  - senha encriptada por BCrypt (hash unidirecional)
- Criada classe controller para permitir o login: AuthenticationController (/login)
- **Configurar o projeto indicando que a senha está codificada com BCrypt**



# Modificação necessária na classe Usuario



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos



vale  
a pena  
ver  
de novo

```
@Service
public class AutenticacaoService implements UserDetailsService {

    @Autowired
    private UsuarioRepository repository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        return repository.findByLogin(username);
    }
}
```

A classe `UserDetails` faz parte do Spring Security e é usada para representar os detalhes de um usuário durante o processo de autenticação e autorização. Ela é uma interface que define os métodos e atributos comuns associados a um usuário. A classe `UserDetails` geralmente é implementada por outras classes que representam um usuário específico no contexto de uma aplicação.

No seu código, a classe `AutenticacaoService` usa um repositório de usuários (`UsuarioRepository`) para buscar um usuário com base no nome de usuário fornecido e, em seguida, retorna os detalhes desse usuário encapsulados em um objeto que implementa a interface `UserDetails`. Esses detalhes do usuário, como nome de usuário, senha e autorizações, serão usados pelo Spring Security para autenticar o usuário e determinar suas permissões na aplicação.

Nossa classe Usuario  
precisa implementar a interface  
UserDetails



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

A interface '**UserDetails**' no contexto do Spring Security fornece um contrato que as classes de usuário devem seguir para representar informações sobre um usuário no sistema. Ao implementar a interface '**UserDetails**', uma classe de usuário está fornecendo informações essenciais para o Spring Security gerenciar a autenticação e autorização de usuários.

A interface '**UserDetails**' define vários métodos que precisam ser implementados para fornecer detalhes específicos do usuário. Alguns desses métodos incluem:

1. **getUsername()**: Retorna o nome de usuário do usuário.
2. **getPassword()**: Retorna a senha do usuário.
3. **getAuthorities()**: Retorna uma coleção de objetos '**GrantedAuthority**', que representam os papéis (roles) do usuário.
4. **isEnabled()**: Indica se o usuário está habilitado ou desabilitado.
5. **isAccountNonExpired()**, **isAccountNonLocked()**, **isCredentialsNonExpired()**: Métodos que indicam se a conta, bloqueio e credenciais do usuário estão expirados ou não.

Ao implementar a interface '**UserDetails**', o desenvolvedor está fornecendo ao Spring Security uma maneira padronizada de acessar e entender as informações do usuário. Isso permite que o Spring Security integre-se de maneira eficaz com diferentes fontes de dados de usuário, como bancos de dados, serviços LDAP, ou outros provedores personalizados.

# Classe Usuario: era...

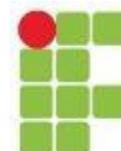
```
public class Usuario {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
}
```



# Classe Usuario: fica...

```
public class Usuario implements UserDetails {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
}
```

Como vimos no slide anterior, temos que implementar alguns métodos desta interface.



```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
  
}  
|
```

control + i



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

```
public class Usuario implements UserDetails {
```

```
@Id
```

```
@GeneratedValue(s
```

```
private Long id;
```

```
private String lo
```

```
private String se
```

```
}
```

### Select Methods to Implement

- org.springframework.security.core.userdetails.UserDe
  - m g getAuthorities():Collection<? extends GrantedAut
  - m g getPassword():String
  - m g getUsername():String
  - m g isAccountNonExpired():boolean
  - m g isAccountNonLocked():boolean
  - m g isCredentialsNonExpired():boolean
  - m g isEnabled():boolean



control + i

Copy JavaDoc

Generate missed JavaDoc

Insert @Override

OK

Cancel



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return null;  
    }  
  
    @Override  
    public String getPassword() { return null; }  
  
    @Override  
    public String getUsername() { return null; }  
  
    @Override  
    public boolean isAccountNonExpired() { return false; }  
  
    @Override  
    public boolean isAccountNonLocked() { return false; }  
  
    @Override  
    public boolean isCredentialsNonExpired() { return false; }  
  
    @Override  
    public boolean isEnabled() { return false; }  
}
```

Neste semestre está gerando com outros retornos.

Como vamos mudar,  
não atualizei o slide.

```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
  
    @Override  
    public Collection<?> getRoles() {  
        return null;  
    }  
  
    @Override  
    public String getPassword() { return null; }  
  
    @Override  
    public String getUsername() { return null; }  
  
    @Override  
    public boolean isAccountNonExpired() { return false; }  
  
    @Override  
    public boolean isAccountNonLocked() { return false; }  
  
    @Override  
    public boolean isCredentialsNonExpired() { return false; }  
  
    @Override  
    public boolean isEnabled() { return false; }  
}
```

**isEnabled():** Indica se o usuário está habilitado ou desabilitado.

**isAccountNonExpired(), isAccountNonLocked(), isCredentialsNonExpired():** Métodos que indicam se a conta, bloqueio e credenciais do usuário estão expirados ou não.

```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
  
    @Override  
    public Collection<?> getRoles() {  
        return null;  
    }  
  
    @Override  
    public String getPassword() { return null; }  
  
    @Override  
    public String getUsername() { return null; }  
  
    @Override  
    public boolean isAccountNonExpired() { return false; }  
  
    @Override  
    public boolean isAccountNonLocked() { return false; }  
  
    @Override  
    public boolean isCredentialsNonExpired() { return false; }  
  
    @Override  
    public boolean isEnabled() { return false; }  
}
```

**isEnabled():** Indica se o usuário está habilitado ou desabilitado.

**isAccountNonExpired(), isAccountNonLocked(), isCredentialsNonExpired():** Métodos que indicam se a conta, bloqueio e credenciais do usuário estão expirados ou não.

Como neste exemplo  
não vamos gerenciar  
esses detalhes,  
precisamos trocar  
todos os retornos  
para true.

```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return null;  
    }  
  
    @Override  
    public String getPassword() { return null; }  
  
    @Override  
    public String getUsername() { return null; }  
  
    @Override  
    public boolean isAccountNonExpired() { return true; }  
  
    @Override  
    public boolean isAccountNonLocked() { return true; }  
  
    @Override  
    public boolean isCredentialsNonExpired() { return true; }  
  
    @Override  
    public boolean isEnabled() { return true; }  
}
```

**isEnabled():** Indica se o usuário está habilitado ou desabilitado.

**isAccountNonExpired(), isAccountNonLocked(), isCredentialsNonExpired():** Métodos que indicam se a conta, bloqueio e credenciais do usuário estão expirados ou não.

Como neste exemplo não vamos gerenciar esses detalhes, precisamos trocar todos os retornos para true.

```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;   
    private String senha;  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return null;  
    }  
  
    @Override  
    public String getPassword() { return null; }   
  
    @Override  
    public String getUsername() { return null; }   
  
    @Override  
    public boolean isAccountNonExpired() { return true; }  
  
    @Override  
    public boolean isAccountNonLocked() { return true; }  
  
    @Override  
    public boolean isCredentialsNonExpired() { return true; }  
  
    @Override  
    public boolean isEnabled() { return true; }  
}
```

```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login; ←  
    private String senha;  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return null;  
    }  
  
    @Override  
    public String getPassword() { return senha; }  
  
    @Override  
    public String getUsername() { return login; } ←  
  
    @Override  
    public boolean isAccountNonExpired() { return true; }  
  
    @Override  
    public boolean isAccountNonLocked() { return true; }  
  
    @Override  
    public boolean isCredentialsNonExpired() { return true; }  
  
    @Override  
    public boolean isEnabled() { return true; }  
}
```

As permissões  
("papéis", privilégios)  
que o usuário vai ter:



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return null;  
    }  
}
```

3. **getAuthorities()**: Retorna uma coleção de objetos 'GrantedAuthority', que representam os papéis (roles) do usuário.

```
@Override  
public boolean isAccountNonExpired() { return true; }  
  
@Override  
public boolean isAccountNonLocked() { return true; }  
  
@Override  
public boolean isCredentialsNonExpired() { return true; }  
  
@Override  
public boolean isEnabled() { return true; }  
}
```



No contexto do Spring Security, a interface `'GrantedAuthority'` representa a autoridade (ou papel) concedida a um usuário. Uma instância de `'GrantedAuthority'` representa um único privilégio ou permissão associado a um usuário.

A classe que implementa `'GrantedAuthority'` geralmente é bastante simples, contendo apenas um método:

java

 Copy code

```
public interface GrantedAuthority {  
    String getAuthority();  
}
```

O método `'getAuthority()'` retorna uma String que representa o papel ou autoridade atribuída ao usuário. Por exemplo, essa String pode ser "ROLE\_USER", "ROLE\_ADMIN", ou qualquer outro valor que represente os privilégios específicos concedidos a um usuário.

A utilização de `'GrantedAuthority'` é comum em conjunto com a interface `'UserDetails'` no Spring Security. O método `'getAuthorities()'` da interface `'UserDetails'` retorna uma coleção de objetos `'GrantedAuthority'`, representando os papéis do usuário.



por que o método `getAuthorities` retorna uma coleção de `GrantedAuthority`, e não apenas uma permissão? seria porque um único usuário pode ter várias permissões (privilégios, papéis) diferentes?

Exatamente, você captou a ideia correta! O método `'getAuthorities()'` retorna uma coleção de `'GrantedAuthority'` em vez de apenas uma permissão para lidar com a flexibilidade de representar vários privilégios, papéis ou permissões associados a um usuário.

A razão para isso é que em sistemas mais complexos, um usuário pode ter diferentes papéis ou privilégios em contextos distintos. Por exemplo, um usuário pode ser um "Usuário Padrão" em um contexto e um "Administrador" em outro. Ou ainda, em um sistema de blog, um usuário pode ter permissões de "Leitura" e "Escrita".

Ao retornar uma coleção de `'GrantedAuthority'`, o Spring Security permite a modelagem de cenários onde um usuário pode ter múltiplos papéis ou permissões. Cada `'GrantedAuthority'` na coleção representa um papel específico ou permissão associada ao usuário.

A flexibilidade dessa abordagem é útil para sistemas que exigem uma granularidade fina de controle de acesso, onde diferentes partes do sistema podem exigir diferentes níveis de autorização. Dessa forma, a mesma entidade de usuário pode ter diferentes papéis ou permissões em contextos variados.

```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return null;  
    }  
  
    @Override  
    public String getPassword() { return senha; }  
  
    @Override  
    public String getUsername() { return login; }  
  
    @Override  
    public boolean isAccountNonExpired() { return true; }  
  
    @Override  
    public boolean isAccountNonLocked() { return true; }  
  
    @Override  
    public boolean isCredentialsNonExpired() { return true; }  
  
    @Override  
    public boolean isEnabled() { return true; }  
}
```

Neste exemplo,  
não implementaremos  
permissões diversas,  
apenas a permissão de  
“usuário”.

```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return List.of(new SimpleGrantedAuthority("ROLE_USER"));  
    }  
  
    @Override  
    public String getPassword() { return senha; }  
  
    @Override  
    public String getUsername() { return login; }  
  
    @Override  
    public boolean isAccountNonExpired() { return true; }  
  
    @Override  
    public boolean isAccountNonLocked() { return true; }  
  
    @Override  
    public boolean isCredentialsNonExpired() { return true; }  
  
    @Override  
    public boolean isEnabled() { return true; }  
}
```

Retorna uma Lista,  
com apenas um objeto  
dentro dela,  
a permissão “ROLE\_USER”

# Versão final da classe Usuario:

```
public class Usuario implements UserDetails {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String login;  
    private String senha;  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return List.of(new SimpleGrantedAuthority("ROLE_USER"));  
    }  
  
    @Override  
    public String getPassword() { return senha; }  
  
    @Override  
    public String getUsername() { return login; }  
  
    @Override  
    public boolean isAccountNonExpired() { return true; }  
  
    @Override  
    public boolean isAccountNonLocked() { return true; }  
  
    @Override  
    public boolean isCredentialsNonExpired() { return true; }  
  
    @Override  
    public boolean isEnabled() { return true; }  
}
```

# ACOMPANHAMENTO DOS PASSOS

- Adicionado o Spring Security no projeto.
- Criada a classe/entidade que vai representar um usuário de nossa API
- Criada a classe que representa o “serviço” de autenticação de usuários
- Criada a classe que define configurações de segurança.
  - util | security | SecurityConfigurations
  - criado método para gerar um objeto (@Bean) responsável pela (de)codificação BCrypt
- Criado usuário na tabela de usuários
  - senha encriptada por BCrypt (hash unidirecional)
- Criada classe controller para permitir o login: AutenticationController (/login)
- Configurar o projeto indicando que a senha está codificada com BCrypt
- **Classe Usuario implementar a interface UserDetails**
  - implementar métodos da interface





INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

# Como configuramos stateless, aquela senha some...



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

Project .idea .mvn DATA pw3api.mv.db src

Run Api20251cApplication

↻ □ 📸 ↻ :

```
2025-05-10T15:11:09.564-03:00 INFO 22762 --- [api20251c] [main] o.f.core.internal.command.DbMigrate : Current version of schema "PUBLIC": 5
2025-05-10T15:11:09.575-03:00 INFO 22762 --- [api20251c] [main] o.f.core.internal.command.DbMigrate : Migrating schema "PUBLIC" to version "6"
2025-05-10T15:11:09.594-03:00 INFO 22762 --- [api20251c] [main] o.f.core.internal.command.DbMigrate : Successfully applied 1 migration to sch
2025-05-10T15:11:09.677-03:00 INFO 22762 --- [api20251c] [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitIn
2025-05-10T15:11:09.710-03:00 INFO 22762 --- [api20251c] [main] org.hibernate.Version : HHH000412: Hibernate ORM core version 6
2025-05-10T15:11:09.735-03:00 INFO 22762 --- [api20251c] [main] o.h.c.internal.RegionFactoryInitiator : HHH000026: Second-level cache disabled
2025-05-10T15:11:09.928-03:00 INFO 22762 --- [api20251c] [main] o.s.o.j.p.SpringPersistenceUnitInfo : No LoadTimeWeaver setup: ignoring JPA c
2025-05-10T15:11:09.977-03:00 INFO 22762 --- [api20251c] [main] org.hibernate.orm.connections.pooling : HHH10001005: Database info:
Database JDBC URL [Connecting through datasource 'HikariDataSource (HikariPool-1)']
Database driver: undefined/unknown
Database version: 2.2.224
Autocommit mode: undefined/unknown
Isolation level: undefined/unknown
Minimum pool size: undefined/unknown
Maximum pool size: undefined/unknown
2025-05-10T15:11:10.377-03:00 INFO 22762 --- [api20251c] [main] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000489: No JTA platform available (s
2025-05-10T15:11:10.378-03:00 INFO 22762 --- [api20251c] [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory fo
2025-05-10T15:11:10.531-03:00 INFO 22762 --- [api20251c] [main] r$InitializeUserDetailsManagerConfigurer : Global AuthenticationManager configured
2025-05-10T15:11:10.578-03:00 WARN 22762 --- [api20251c] [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by d
2025-05-10T15:11:10.674-03:00 WARN 22762 --- [api20251c] [main] s.w.s.m.m.a.RequestMappingHandlerMapping : Multiple @RequestMapping annotations fo
2025-05-10T15:11:10.923-03:00 INFO 22762 --- [api20251c] [main] o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'.
2025-05-10T15:11:10.975-03:00 INFO 22762 --- [api20251c] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with
2025-05-10T15:11:10.983-03:00 INFO 22762 --- [api20251c] [main] b.e.i.p.api20251c.Api20251cApplication : Started Api20251cApplication in 3.055 s
```

Testando a requisição de login...  
Se valida corretamente login e senha.



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

POST  localhost:8080/login

Params Authorization Headers (9) **Body**  Pre-request Script Tests Settings Cookies

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL **JSON**

```
1 {  
2   "login" : "asdrubal@gmail.com",  
3   "senha" : "123456"  
4 }  
5
```



```
public record dadosAutenticacao(  
  
    @NotBlank  
    String login,  
  
    @NotBlank  
    String senha) {  
}
```

**POST**

▼

localhost:8080/login

**Send**

▼

[Params](#) [Authorization](#) [Headers \(9\)](#) [Body](#) • [Pre-request Script](#) [Tests](#) [Settings](#) [Cookies](#) none  form-data  x-www-form-urlencoded  raw  binary  GraphQL **JSON** ▾[Beautify](#)

```
1 {  
2   "login" : "asdrubal@gmail.com",  
3   "senha" : "123456"  
4 }  
5
```

[Body](#) [Cookies](#) [Headers \(10\)](#) [Test Results](#)

200 OK

325 ms 293 B



Save as example

..

[Pretty](#)[Raw](#)[Preview](#)[Visualize](#)[Text](#) ▾

1



**INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO**  
Câmpus São Carlos

# Testando com senha errada...



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos

POST



localhost:8080/login

Send



Params

Authorization

Headers (9)

Body

Pre-request Script

Tests

Settings

Cookies

 none form-data x-www-form-urlencoded raw binary GraphQL JSON

Beautify

```
1 {  
2   "login": "asdrubal@gmail.com",  
3   "senha": "12345678"  
4 }  
5
```

Body

Cookies

Headers (11)

Test Results



403 Forbidden 257 ms 461 B

Save as example



Pretty

Raw

Preview

Visualize

JSON



```
1 {  
2   "timestamp": "2023-11-09T15:40:30.810+00:00",  
3   "status": 403,  
4   "error": "Forbidden",  
5   "message": "Access Denied",  
6   "path": "/login"  
7 }
```



Vamos encerrar por aqui hoje.

Neste ponto, o projeto funciona,  
mas aceita todas as requisições  
(ainda não está realizando a autenticação).

Falta (muita) coisa ainda...

OBS:  
Estado atual do projeto  
disponível no github.

[https://github.com/carlaopereirasanca/prw3\\_api\\_2025\\_2](https://github.com/carlaopereirasanca/prw3_api_2025_2)

# Exercícios



# Exercícios

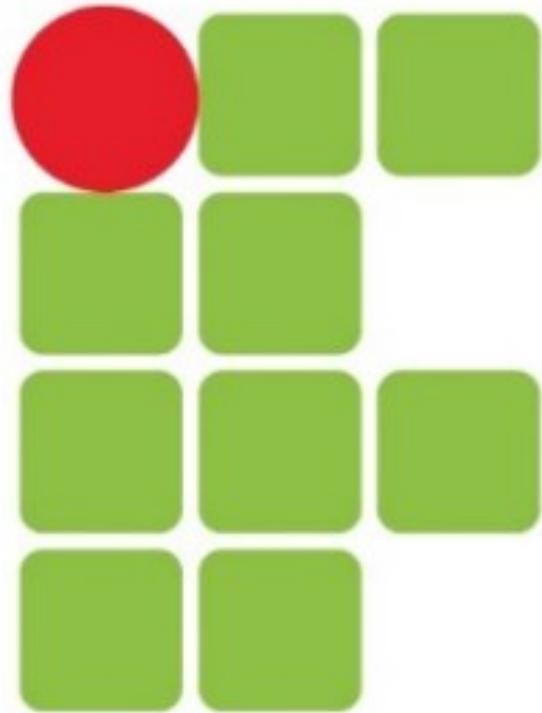
- Experimente com os códigos vistos na aula de hoje...



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos



INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos



**INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos**



**INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
SÃO PAULO  
Câmpus São Carlos**