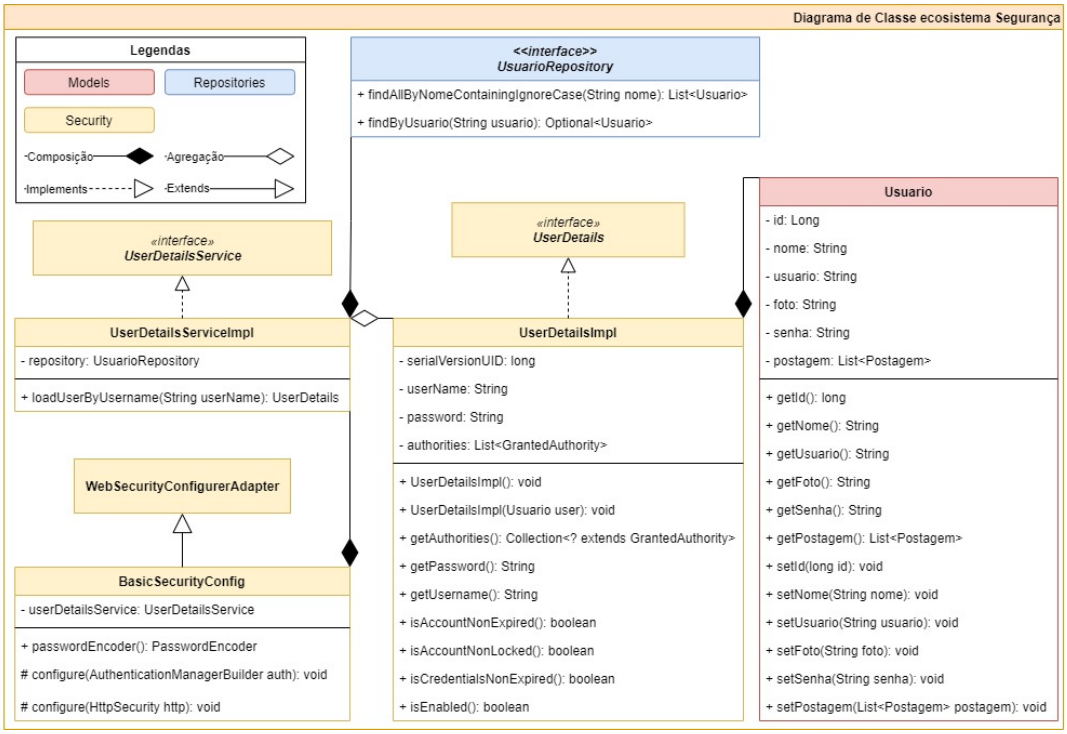
Projeto 02 - Blog Pessoal - Spring Security 02

O que veremos por aqui:

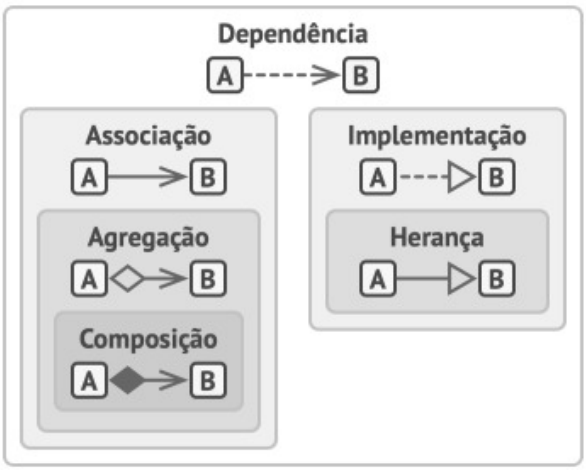
1. O Ecossistema da Segurança
2. A Classe UserDetailsImpl
3. A Classe UserDetailsServiceImpl
4. A Classe BasicSecurityConfig

Antes de finalizarmos o Ecossistema do Usuário, vamos compreender e implementar o Ecossistema da Segurança, porquê utilizaremos alguns Métodos deste ecossistema na implementação das **Classes UsuarioService e UsuarioController**.

1. Ecossistema da Segurança



No Diagrama de Classes acima, é possível visualizar por completo o ecossistema da Segurança, que representa uma das principais implementações de segurança do sistema. Nesta implementação é possível realizar a autenticação do usuário a partir de um Banco de dados. Para isso é necessário enviar as informações necessárias para que a Spring Security valide a permissão do usuário no sistema. O ecossistema de segurança consiste de uma classe de configuração (**BasicSecurityConfig**), uma de Classe Serviço (**UserDetailsService**) e uma Classe Model (**UserDetailsImpl**). A cor de cada uma das Classes representa o pacote onde cada uma deve ser implementada, segundo informações da legenda. Para este Diagrama de Classes, repare que a Classe Model do Usuario e o Repositório do Usuário já foram criadas anteriormente. Para melhor compreensão das relações entre as Classes veja uma breve definição sobre as relações entre os Objetos:



Relações entre objetos e classes: da mais fraca a mais forte.

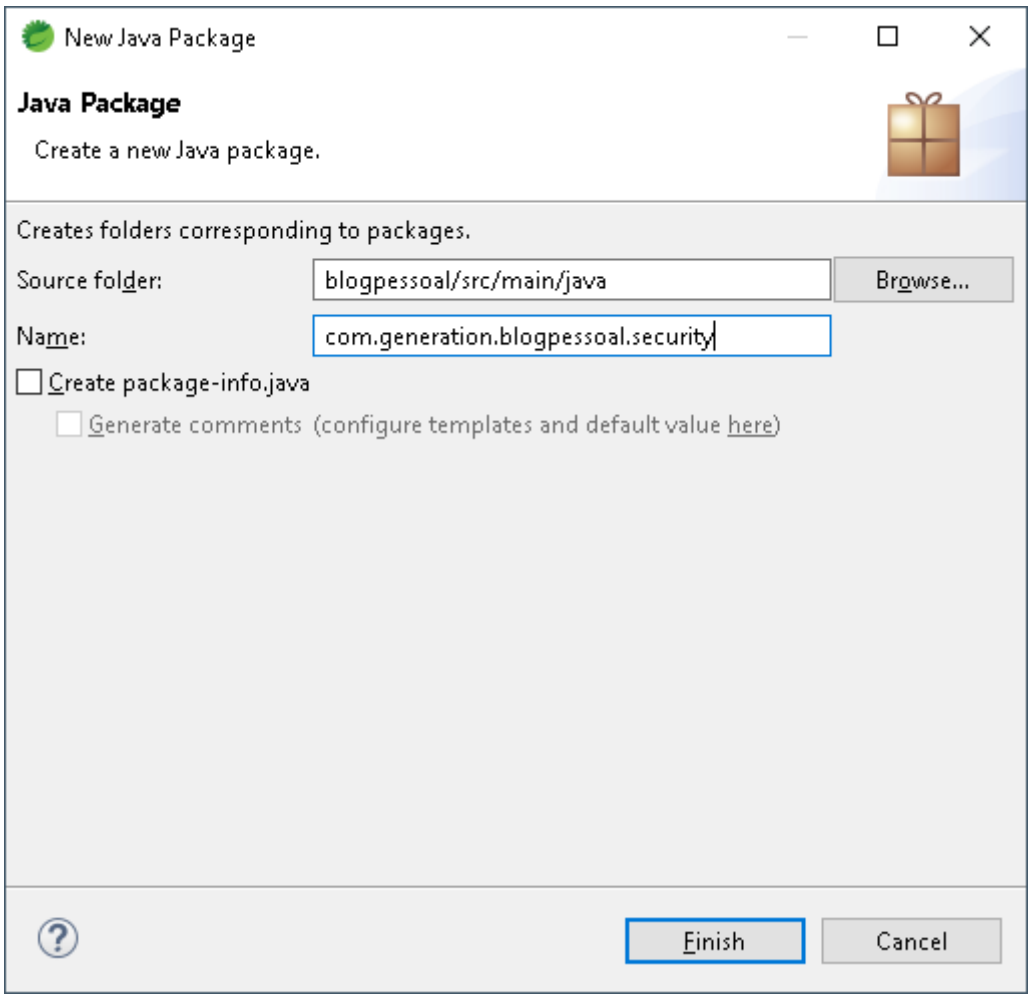
Relação	Descrição
Dependência	Classe A pode ser afetada por mudanças na classe B;
Associação	Objeto A sabe sobre objeto B. Classe A depende de B;
Agregação	Objeto A sabe sobre objeto B, e consiste de B. Classe A depende de B;
Composição	Objeto A sabe sobre objeto B, consiste de B, e gerencia o ciclo de vida de B. Classe A depende de B;
Implementação	Classe A define métodos declarados na interface B. objetos de A podem ser tratados como B. Classe A depende de B;
Herança	Classe A herda a interface e implementação da classe B mas pode estende-la. Objetos de A podem ser tratados como B. Classe A depende de B.

Após compreender o Diagrama de Classes acima e aprender um pouco sobre UML, segue a aplicação do código em cada uma de suas Classes.

👣 Passo 01 - Criar o Pacote Security

Primeiro vamos criar a camada de segurança, ou seja, o Pacote Security, onde as Classes serão implementadas:

1. No lado esquerdo superior, na Guia **Package explorer**, clique com o botão direito do mouse sobre a Package **com.generation.blogpessoal**, na Source Folder **src/main/java** e clique na opção **New → Package**.
2. Na janela **New Java Package**, no item **Name**, acrescente no final do nome da Package **.security**, como mostra a figura abaixo:



3. Clique no botão **Finish** para concluir.

👣 Passo 02 - Criar a Classe UserDetailsImpl na Camada Security

Esta classe implementa a Interface **UserDetails**, que tem como principal funcionalidade fornecer as informações básicas do usuário para a Spring Security. Ao criar esta Classe, será permitido que uma Classe de Serviço (Service), preencha os atributos através do Método Construtor e retorne apenas o necessário para que a Spring Security possa validar o usuário que está tentando acessar o sistema.

1. Clique com o botão direito do mouse sobre o **Pacote Security** (**com.generation.blogpessoal.security**), na Source Folder Principal (**src/main/java**), e clique na opção **New → Class**
2. Na janela **New Java Class**, no item **Name**, digite o nome da Classe (**UserDetailsImpl**), e na sequência clique no botão **Finish** para concluir.

	**ATENÇÃO:** ***Por se tratar de uma Implementação de uma Interface (UserDetails), todos os Métodos da Interface devem ser implementados e o nome da Classe deve obrigatoriamente conter o sufixo Impl (Implements), indicando que a Classe está Implementando a Interface.***

Vejamos abaixo a implementação da Classe:

```java
package com.generation.blogpessoal.security;

import java.util.Collection;
import java.util.List;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import com.generation.blogpessoal.model.Usuario;

public class UserDetailsImpl implements UserDetails{

private static final long serialVersionUID =1L;

    private String userName;
    private String password;

    private List<GrantedAuthority> authorities;

    public UserDetailsImpl (Usuario user){
        this.userName = user.getUsuario();
        this.password = user.getSenha();
    }

    public UserDetailsImpl (){}

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {

        return authorities;
    }

    @Override
    public String getPassword() {

        return password;
    }

    @Override
    public String getUsername() {

        return userName;
    }

    @Override
    public boolean isAccountNonExpired() {

        return true;
    }

    @Override
    public boolean isAccountNonLocked() {

        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
```

```
        return true;
    }



    @Override
    public boolean isEnabled() {

        return true;
    }

}
```

Para concluir, não esqueça de Salvar o código (**File → Save All**).

ALERTA DE BSM: Mantenha a Atenção aos Detalhes, é necessário ter um construtor adaptado para receber atributos que serão utilizados para logar no sistema. Os atributos em questão são usuário e senha, mas poderia ser também e-mail e senha. Tudo irá depender de como a model de Usuario esta elaborada.

ATENÇÃO: *Observe que o Método getAuthorities(), que retorna a lista com os direitos de acesso do usuário, sempre retornará uma List vazia, porquê este atributo não pode ser Nulo. Com o objetivo de simplificar a implementação, todo o Usuário autenticado terá todos os direitos de acesso sobre a aplicação. *

Documentação: *UserDetails*

Documentação: *GrantedAuthority*

🦶 Passo 03 - Criar a Implementação da Classe UserDetailsServiceImpl

Esta Classe é uma implementação da Interface **UserDetailsService**, responsável por fornecer a estrutura de um método utilizado pela Spring Security para validar a existência de um usuário no Banco de dados e retornar um **UserDetails**. Com os dados fornecidos ao sistema, será possível autenticar um usuario baseando-se na sua existência no banco. Vale lembrar que para isso é necessário que ao salvar o usuário, sua senha esteja criptografada (Veremos na implementação da Classe UsuarioService), utilizando uma biblioteca de criptografia válida para o sistema de segurança.

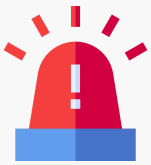
1. Clique com o botão direito do mouse sobre o **Pacote Security** (**com.generation.blogpessoal.security**), na Source Folder Principal (**src/main/java**), e clique na opção **New → Class**
2. Na janela **New Java Class**, no item **Name**, digite o nome da Classe (**UserDetailsServiceImpl**), e na sequência clique no botão **Finish** para concluir.

ATENÇÃO: ***Por se tratar de uma Implementação de uma Interface (UserDetailsService), todos os Métodos da Interface devem ser implementados e o nome da Classe deve obrigatoriamente conter o sufixo Impl (Implements), indicando que a Classe está Implementando a Interface.***

Vejamos abaixo a implementação da Classe:

```
package com.generation.blogpessoal.security;

import java.util.Optional;

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;
import com.generation.blogpessoal.model.Usuario;
import com.generation.blogpessoal.repository.UsuarioRepository;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UsuarioRepository repository;

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

        Optional<Usuario> usuario = repository.findByUsuario(userName);
        usuario.orElseThrow(() -> new UsernameNotFoundException(userName +
        " não encontrado."));

        return usuario.map(UserDetailsImpl::new).get();
    }
}
```

Observe que na linha **return usuario.map(UserDetailsImpl::new).get();**, a Classe Retorna um objeto do tipo UserDetailsImpl criado com os dados recuperados do Banco de dados. O **operador ::** faz parte de uma expressão que referencia um Método, complementando uma função Lambda. Neste exemplo, o operador faz referência ao Método Construtor da Classe UserDetailsImpl. Esta instrução poderia é equivalente ao seguinte comando: **return new UserDetailsImpl(optional.get());**.



ALERTA DE BSM: Mantenha a Atenção aos Detalhes, ao criar este serviço não esquecer de colocar a anotação @Service. Desta forma o Spring entenderá que a classe é um serviço e fará o carregamento com prioridade para esta classe.

 [Documentação: *UserDetailsService*]

🦶 Passo 04 - Criar a Classe BasicSecurityConfig

Esta Classe é utilizada para sobrescrever a configuração já predefinida pela **Spring Security** para a segurança. A Classe possui a anotação **@EnableWebSecurity**, que habilita e sobrescreve os Métodos que irão redefinir as regras de Segurança de acordo com as Regras da sua aplicação.

1. Clique com o botão direito do mouse sobre o **Pacote Security** (**com.generation.blogpessoal.security**), na Source Folder Principal (**src/main/java**), e clique na opção **New → Class**
2. Na janela **New Java Class**, no item **Name**, digite o nome da Classe (**BasicSecurityConfig**), e na sequência clique no botão **Finish** para concluir.

```
package com.generation.blogpessoal.security;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.authentication
.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web
.builders.HttpSecurity;
import org.springframework.security.config.annotation.web
.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web
.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@EnableWebSecurity
public class BasicSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
         auth.userDetailsService(userDetailsService);

         auth.inMemoryAuthentication()
            .withUser("root")
            .password(passwordEncoder().encode("root"))
            .authorities("ROLE_USER");
        }

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

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {
        http.authorizeRequests()
            .antMatchers("/usuarios/logar").permitAll()
```

```
                .antMatchers("/usuarios/cadastrar").permitAll()
                .antMatchers(HttpMethod.OPTIONS).permitAll()
                .anyRequest().authenticated()
                .and().httpBasic()
                .and().sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and().cors()
                .and().csrf().disable();
    }
}
```

Na implementação acima a anotação **@Bean** no método ***passwordEncoder()*** , indica ao Spring Security que a aplicação está baseada em algum modelo de criptografia, que será implementada no Método **PasswordEncoder**. Para esta aplicação estamos utilizando o modelo **BCryptPasswordEncoder()**. Vale ressaltar que esta criptografia está sendo esperada para analisar a senha que estamos guardando no Banco de dados.

Dica: No primeiro link abaixo você pode conhecer mais sobre o BCrypt e no segundo link testar a codificação e a decodificação de Strings através do BCrypt.


Artigo: *Uma breve introdução sobre BCrypt*


Site: *Codificador BCrypt*

Quando sobrescrevemos o método ***configure(AuthenticationManagerBuilder auth)***, estamos informando ao spring que a configuração de autenticação será redefinida em sua implementação. Neste método estamos fornecendo duas formas de autenticação:

.userDetailsService(userDetailsService): utiliza um serviço para pesquisar a credencial do usuário no Banco de dados.

.inMemoryAuthentication(): utiliza um usuário em memória, no caso definido no próprio método, como usuário **root** e senha **root**.

Para mais detalhes sobre esta definição, o mais correto é olhar sua fonte no link abaixo:


Documentação: *AuthenticationManagerBuilder*

Quando sobrescrevemos o método ***configure(HttpSecurity http)***, estamos informando ao Spring que a configuração por padrão definida será alterada. Nesta configuração é possivel customizar de diferentes maneiras a sua aplicação.

.authorizeRequests(): podemos definir quais endpoints poderão acessar o sistema sem precisar de autenticação. No caso acima foi definido que os endpoints **logar e cadastrar** serão autorizados para todos os usuários, enquanto para os demais endpoints será necessário autenticar.

.antMatchers("/usuarios/logar").permitAll() e .antMatchers("/usuarios/cadastrar").permitAll(): permite definir o caminho do endpoint que estará acessível sem autenticação.

ATENÇÃO: ***Os endpoints cadastrar e logar foram liberados porquê caso contrário ninguém conseguirá acessar a aplicação.*** **Além disso, é fundamental que os endereços dos endpoints: /usuarios/logar e /usuarios/cadastrar estejam definidos na implementação da Classe UsuarioController.**

.antMatchers(HttpMethod.OPTIONS).permitAll(): O parâmetro HttpMethod.OPTIONS permite que o cliente (frontend), possa descobrir quais são as opções permitidas em uma requisição, para um determinado recurso em um servidor. Nesta implementação, está sendo liberada todas as opções das requisições através do método **.permitAll()**.

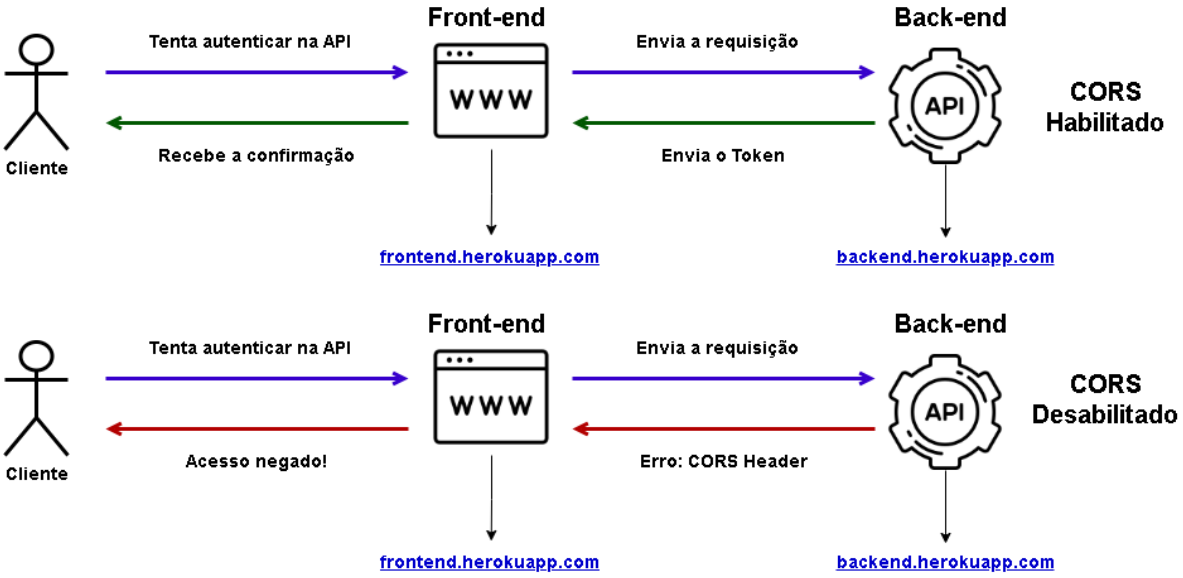
.anyRequest().authenticated(): informa ao sistema que todas as demais rotas que não estiverem especificadas no escopo do *authorizeRequests()*, deverão ser autenticadas.

.httpBasic(): informa ao sistema que o servidor irá receber requisições que devem ter o esquema HTTP Basic de autenticação. Desta forma habilitamos o servidor para que as requisições tenham um formato específico de autenticação.

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS): define que o nosso sistema não guardará sessões para o cliente. Quando o cliente faz uma solicitação HTTP, ele inclui todas as informações necessárias para o servidor atender à solicitação. O servidor nunca dependerá de informações de solicitações anteriores do cliente. Se alguma dessas informações for importante, o cliente a enviará novamente a informação como parte da solicitação atual.

.cors(): libera o acesso de outras origens, desta forma nossa API poderá ser acessada de outros domínios, ou seja, de outros servidores, além do servidor onde a aplicação está hospedada.

Exemplo:



.csrf().disable(): desabilita a proteção que vem ativa contra o ataque do tipo **CSRF (Cross-Site-Request-Forgery)**, que seria uma interceptação dos dados de autenticação por um Hacker. Esta configuração foi desabilitada porquê o Spring Security fica procurando por um *parâmetro oculto adicional* em **qualquer** requisição **POST / PUT / DELETE**, chamado **token CSRF**. Como ele não vai encontrar, apenas as requisições do tipo GET seriam aceitas.

Para mais detalhes sobre esta definição, o mais correto é olhar sua fonte no link abaixo:

[Documentação: *HttpSecurity*]

 Documentação: HTTP Status Code 401 - Unauthorized

 Documentação: CORS

 Artigo: Complete Guide to CSRF/XSRF (Cross-Site Request Forgery)

 Código fonte: Camada Security