

Teste de Software - Spring Boot Testing (JUnit 5 + H2)

O teste de software é uma forma de avaliar a qualidade da aplicação e reduzir os riscos de falhas no código ao ser colocado em operação (Produção). Testar não se resume apenas em executar testes e verificar os resultados. **Executar** testes é apenas uma das atividades. Planejamento, análise, modelagem e implementação dos testes, relatórios de progresso, resultado e avaliação da qualidade, também são partes de um **processo de testes**.

Testar software não é somente **verificar** se os requisitos foram atendidos, atribui-se ao teste de software também a **validação**, ou seja, verificar se o sistema atenderá às necessidades do usuário e de outras partes interessadas em seu(s) ambiente(s) operacional(is).

1. A Pirâmide de Testes

A **Pirâmide de Testes** é uma representação gráfica que nos diz para agrupar testes de software em diferentes tipos. A pirâmide ilustra de forma implícita a quantidade de testes que devem ser realizados em tipo, os custos e o tempo de duração.

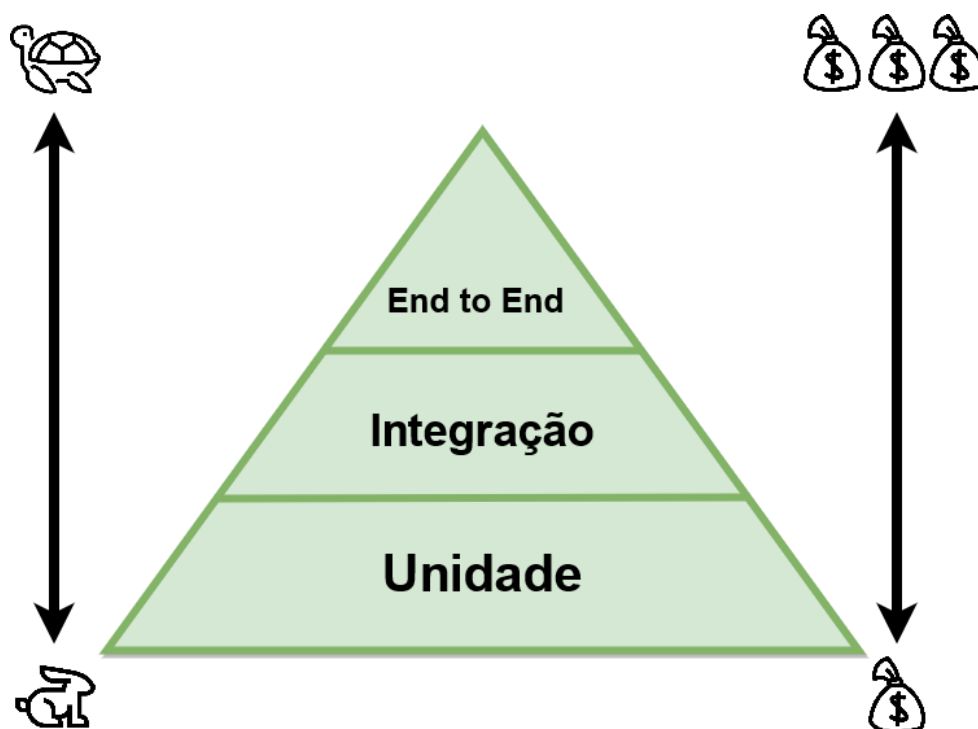


Figura 01: Pirâmide de Testes

Observe que os testes na base são mais rápidos e baratos do que os testes no topo da pirâmide.

Existem três tipos de teste:

- Teste de Unidade
- Teste de Integração
- Teste End to End (E2E)

1.1. Teste de unidade

Uma unidade pode ser uma função, uma classe, um pacote ou um subsistema. Portanto, o termo teste de unidade refere-se à prática de testar pequenas unidades do seu código, para garantir que funcionem conforme o esperado.

O Teste de Unidade é o teste mais comum, porque além de ser muito rápido é o teste mais barato porque pode ser criado pela própria pessoa desenvolvedora durante o processo de codificação.

1.2. Teste de integração

Teste de Integração é a fase do teste de software em que os módulos são combinados e testados em conjunto, para checar como os módulos se comportam quando interagem entre si.

O Teste de Integração é um pouco mais lento e um pouco mais caro do que o Teste de Unidade porque aumenta a complexidade.

1.3. Teste End to End

O Teste de ponta a ponta é uma metodologia de teste de software para testar um fluxo de aplicativo do início ao fim. O objetivo deste teste é simular um cenário real do usuário e validar o sistema em teste e seus componentes para integração e integridade dos dados.

O Teste End to End é mais lento (depende de pessoas para testarem o software como um todo em produção ou versão beta), o que o torna muito mais caro do que os Testes de Unidade e Integração, o que explica serem realizados em menor quantidade.

1.4. O que deve ser testado?

A prioridade sempre será escrever testes para as partes mais complexas ou críticas de seu código, ou seja, aquilo que é essencial para que o código traga o resultado esperado.

Exemplo: Em um e-commerce a finalização da compra é um ponto crítico da aplicação.

2. Spring Boot Testing

O Spring Boot Testing é parte integrante do Spring Boot e oferece suporte a testes de unidade e testes de integração, utilizando alguns Frameworks de Teste Java.

Ao criar um projeto com o Spring Boot, automaticamente as dependências de testes já são inseridas no projeto como veremos adiante.



[Documentação Oficial](#)

2.1. Spring Testing Annotations

Spring Boot Testing	Descrição
<code>@SpringBootTest</code>	<p>A anotação @SpringBootTest cria e inicializa o nosso ambiente de testes.</p> <p>A opção webEnvironment = WebEnvironment.RANDOM_PORT garante que durante os testes o Spring não utilize a porta da aplicação (em ambiente local nossa porta padrão é a 8080), caso ela esteja em execução. Através da opção, o Spring procura uma porta livre para executar os testes.</p>

O Spring Boot Testing trabalha de forma integrada com os principais Frameworks de Teste do Mercado tais como: **JUnit**, **MockMVC** (Parte integrante do Spring Boot Testing), entre outros. Para escrever os nossos testes utilizaremos o **JUnit 5**.

3. O framework JUnit

O JUnit é um Framework de testes de código aberto para a linguagem Java, que é usado para escrever e executar testes automatizados e repetitivos, para que possamos ter certeza que nosso código funciona conforme o esperado.

O JUnit fornece:

- Asserções para testar os resultados esperados.
- Recursos de teste para compartilhar dados de teste comuns.
- Conjuntos de testes para organizar e executar testes facilmente.
- Executa testes gráficos e via linha de comando.

O JUnit é usado para testar:

- Um objeto inteiro
- Parte de um objeto, como um método ou alguns métodos de interação
- Interação entre vários objetos

JUnit  [Documentação: JUnit 5](#)

3.1. Anotações do JUnit

JUnit 5	Descrição
<code>@Test</code>	A anotação <code>@Test</code> indica que o método deve ser executado como um teste.
<code>@BeforeEach</code>	A anotação <code>@BeforeEach</code> indica que o método deve ser executado antes de cada método da classe, para criar pré-condições necessárias para cada teste (criar variáveis, por exemplo).
<code>@BeforeAll</code>	A anotação <code>@BeforeAll</code> indica que o método deve ser executado uma única vez antes de todos os métodos da classe, para criar algumas pré-condições necessárias para todos os testes (criar objetos, por exemplo).
<code>@AfterEach</code>	A anotação <code>@AfterEach</code> indica que o método deve ser executado depois de cada teste para redefinir algumas condições após rodar

	<p>cada teste (redefinir variáveis, por exemplo).</p>
<i>@AfterAll</i>	<p>A anotação @AfterAll indica que o método deve ser executado uma única vez depois de todos os testes da classe, para redefinir algumas condições após rodar todos os testes (redefinir objetos, por exemplo).</p>
<i>@Disabled</i>	<p>A anotação @Disabled desabilita temporariamente a execução de um teste específico. Cada método que é anotado com @Disabled não será executado.</p>
<i>@DisplayName</i>	<p>Personaliza o nome do teste permitindo inserir um Emoji (tecla Windows + .) e texto.</p>
<i>@Order(1)</i>	<p>A anotação @Order informa a ordem em que o teste será executado, caso todos os testes sejam rodados de uma vez só. Para utilizar esta anotação, acrescente a anotação @TestMethodOrder(MethodOrderer.OrderAnnotation.class) antes do nome da Classe de testes.</p>
<i>@TestInstance</i>	<p>A anotação @TestInstance permite modificar o ciclo de vida da classe de testes.</p> <p>A instância de um teste possui dois tipos de ciclo de vida:</p> <ol style="list-style-type: none"> 1) O LifeCycle.PER_METHOD: ciclo de vida padrão, onde para cada método de teste é criada uma nova instância da classe de teste. Quando utilizamos as anotações @BeforeEach e @AfterEach é necessário utilizar esta anotação. 2) O LifeCycle.PER_CLASS: uma única instância da classe de teste é criada e reutilizada entre todos os métodos de teste da classe. Quando utilizamos as anotações @BeforeAll e @AfterAll é necessário utilizar esta anotação.

3.2. Asserções - JUnit

Asserções (Assertions) são métodos utilitários para testar afirmações em testes (1 é igual a 1, por exemplo).

Assertion	Descrição
<code>assertEquals(expected value, actual value)</code>	Afirma que dois valores são iguais.
<code>assertTrue(boolean condition)</code>	Afirma que uma condição é verdadeira.
<code>assertFalse(boolean condition)</code>	Afirma que uma condição é falsa.
<code>assertNotNull()</code>	Afirma que um objeto não é nulo.
<code>assertNull(Object object)</code>	Afirma que um objeto é nulo.
<code>assertSame(Object expected, Object actual)</code>	Afirma que dois objetos referem-se ao mesmo objeto.
<code>assertNotSame(Object expected, Object actual)</code>	Afirma que dois objetos não se referem ao mesmo objeto.
<code>assertArrayEquals(expectedArray, resultArray)</code>	Afirma que array esperado e o array resultante são iguais.



DICA: Ao escrever testes, sempre verifique se a importação dos pacotes do JUnit na Classe de testes estão corretos. O JUnit 5 tem como pacote base `org.junit.jupiter.api`.

4. Banco de Dados H2

O H2 é um Banco de dados relacional escrito em Java. Ele pode ser integrado em aplicativos Java ou executado no modo cliente-servidor. Como o H2 funciona em memória todo o seu armazenamento é **volátil**, ou seja, toda vez que a aplicação for reiniciada, ele será reconstruído e os dados serão removidos. Seu intuito é ser um banco de dados para testes, de configuração rápida e fácil, visando favorecer a produtividade.

[Tutorial Banco de dados H2](#)

5. Quais testes faremos?

Vamos criar testes nas Camadas Repository e Controller/Service do Recurso Usuário do Blog Pessoal.

Para executarmos os testes, faremos algumas configurações na Source Folder de testes **src/test**, algumas configurações no arquivo **pom.xml** e algumas alterações na **Classe Usuario** e na **Interface UsuarioRepository**.

Antes de prosseguir, assegure que o seu projeto Blog Pessoal não esteja em execução no STS.

Teste de Software - Spring Boot Testing (JUnit 5 + H2)

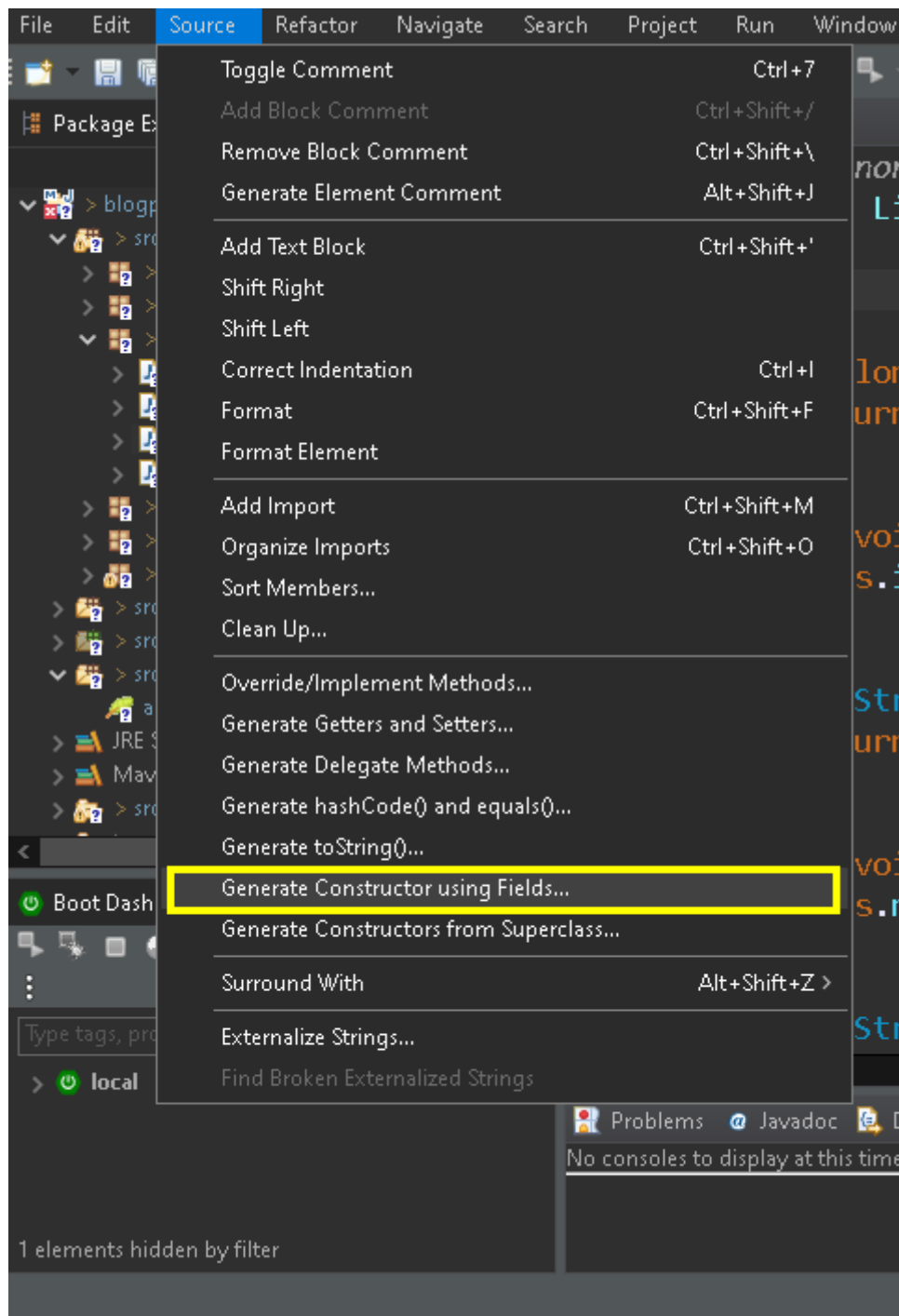
Passo 01 - Criar os Métodos Construtores na Classe Usuario (Camada Model)

Na **Classe Usuario**, na camada Model, vamos criar 2 métodos construtores: o primeiro com todos os atributos (exceto o atributo postagens) e um segundo método construtor vazio, ou seja, sem atributos como mostra a figura abaixo. Através destes dois métodos iremos instanciar alguns objetos da Classe Usuario nas nossas classes de teste.

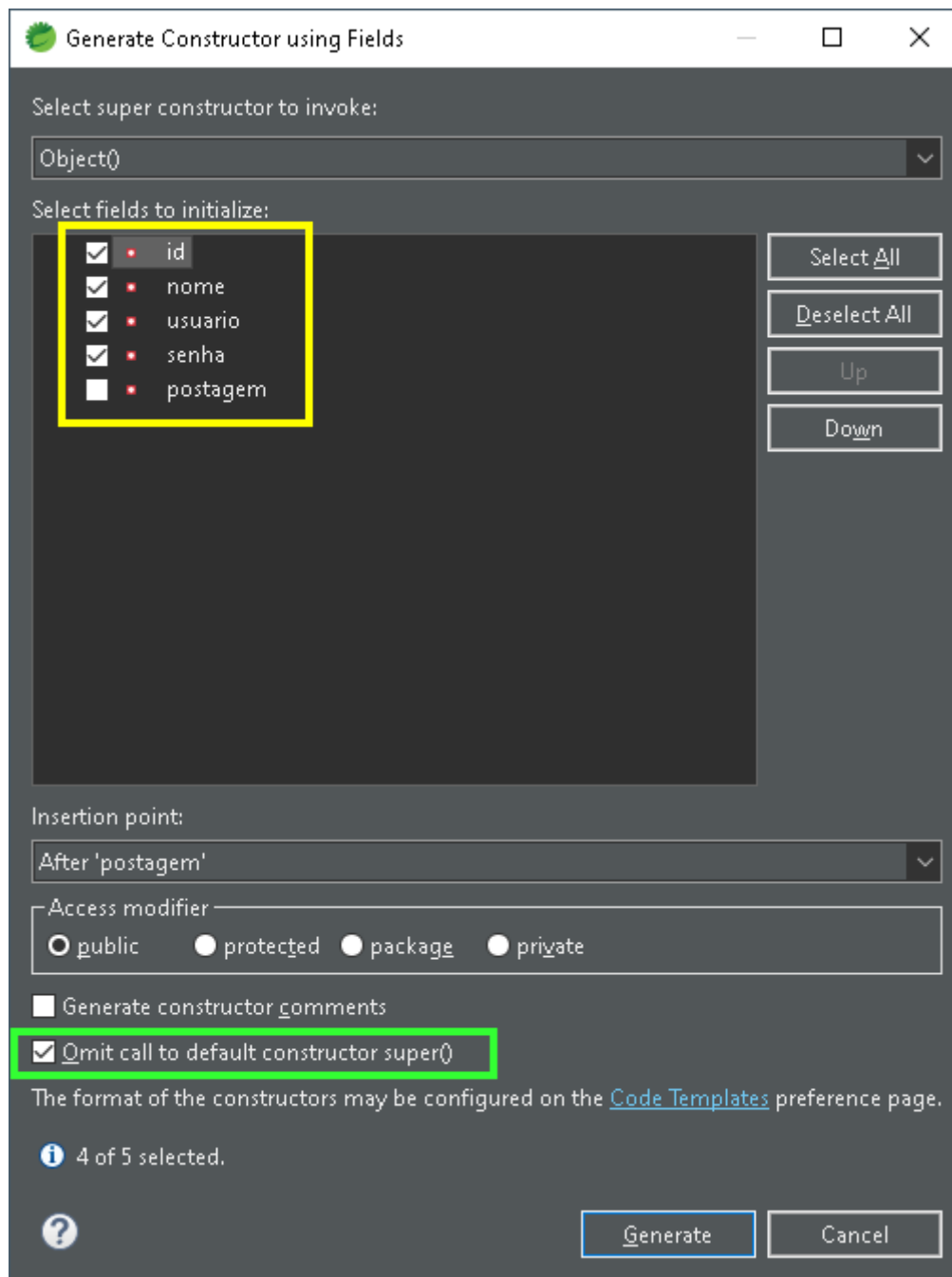
```
37
38●  @OneToMany(mappedBy = "usuario", cascade = CascadeType.REMOVE)
39  @JsonIgnoreProperties("usuario")
40  private List<Postagem> postagem;
41
42  // Primeiro Método Construtor - Com os atributos
43
44●  public Usuario(long id, String nome, String usuario, String senha) {
45      this.id = id;
46      this.nome = nome;
47      this.usuario = usuario;
48      this.senha = senha;
49  }
50
51
52  // Segundo Método Construtor - Sem os atributos
53
54  public Usuario() { }
55
```

O trecho de código destacado em amarelo é o Construtor com parâmetros. O trecho de código destacado em verde é o Construtor vazio (sem parâmetros).

1. Para criar o Primeiro Construtor, posicione o cursor após o último atributo da Classe (em nosso exemplo Postagem) e clique no menu **Source** → **Generate Constructor using fields**.



2. Na janela **Generate Constructor using fields**, selecione todos os atributos, exceto postagem e marque a opção **Omit call to default constructor super()** como mostra a figura abaixo:



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

4. O Construtor será gerado com todas as anotações nos parâmetros, como mostra a figura abaixo:

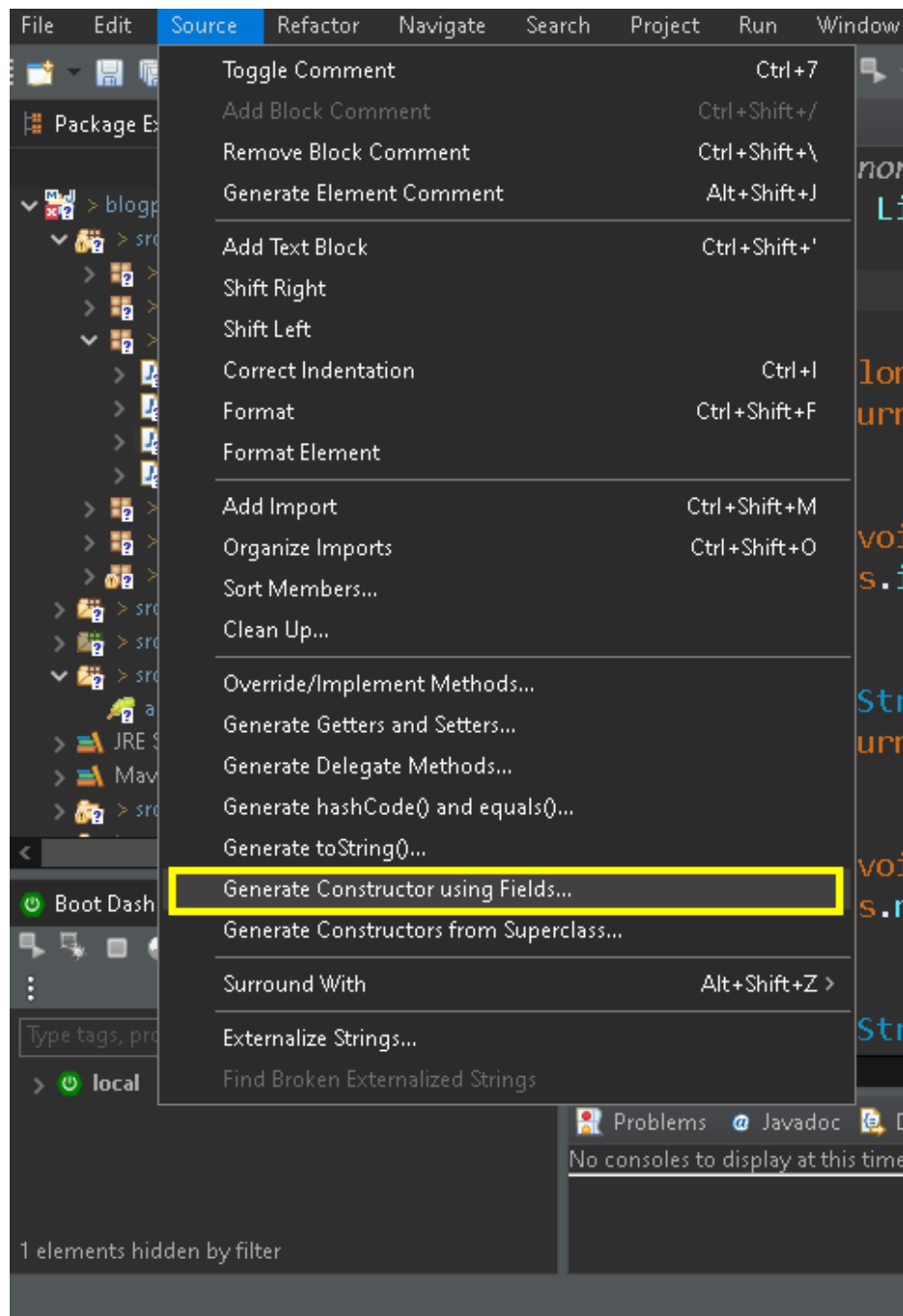
```
42
43 public Usuario(long id,
44     @NotNull(message = "O atributo Nome é Obrigatório!") String nome,
45     @NotNull(message = "O atributo Usuário é Obrigatório!")
46     @Email(message = "O atributo Usuário deve ser um email!") String usuario,
47     @NotBlank(message = "O atributo Senha é Obrigatória!")
48     @Size(min = 8, message = "A Senha deve ter no mínimo 8 caracteres") String senha) {
49     this.id = id;
50     this.nome = nome;
51     this.usuario = usuario;
52     this.senha = senha;
53 }
54
```

5. Apague as anotações dos parâmetros do Método Construtor e deixe o código igual a imagem abaixo:

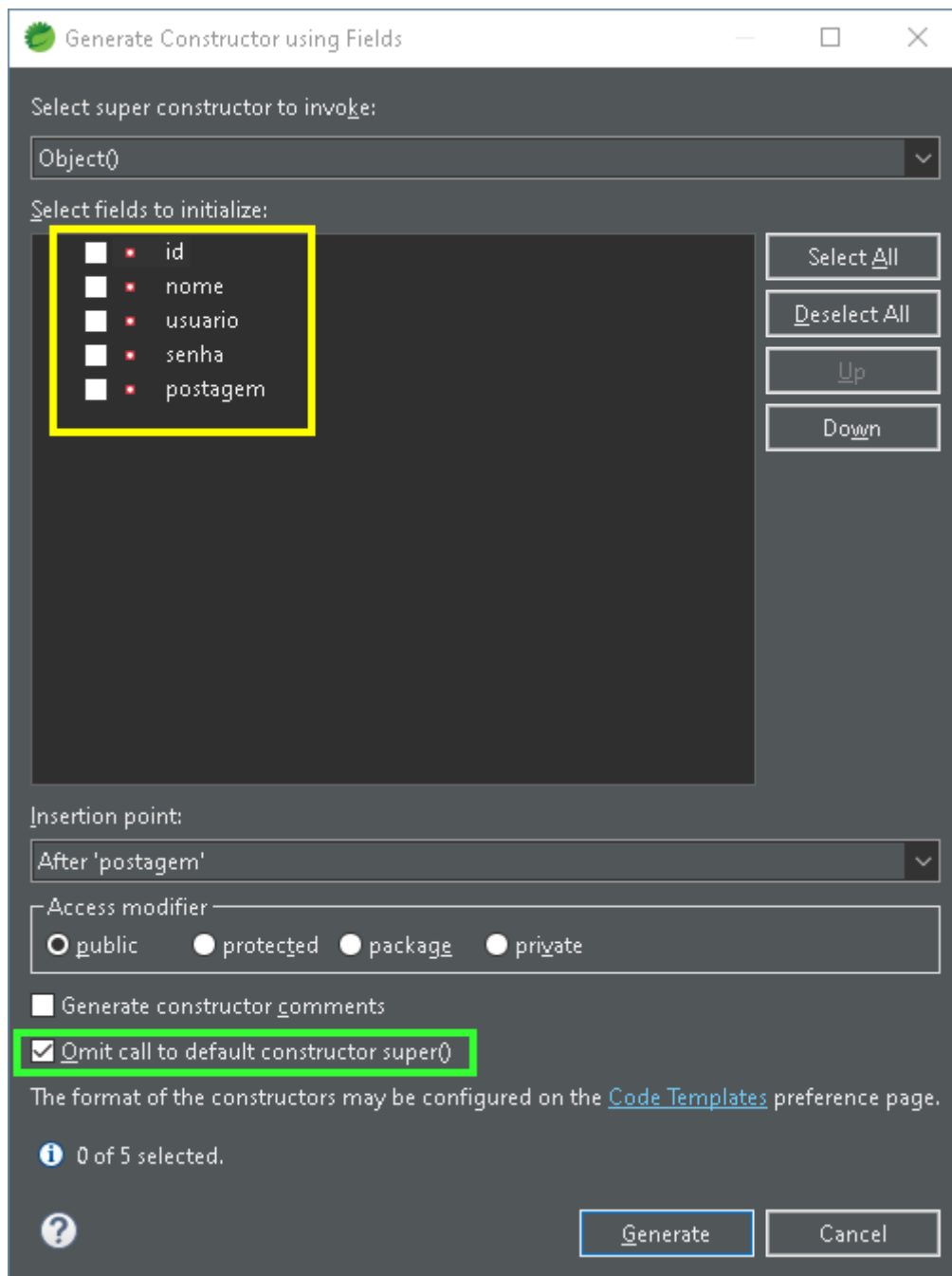
```
41
42 public Usuario(long id, String nome, String usuario, String senha) {
43     this.id = id;
44     this.nome = nome;
45     this.usuario = usuario;
46     this.senha = senha;
47 }
48
```

Agora vamos criar o segundo Método Construtor:

1. Posicione o cursor após o Método Construtor com parâmetros e clique no menu **Source** → **Generate Constructor using fields**.



2. Na janela **Generate Constructor using fields**, desmarque todos os atributos e marque a opção **Omit call to default constructor super()** como mostra a figura abaixo:



3. Clique no botão **Generate** para concluir.
4. O construtor vazio ficará igual a imagem abaixo:

```
49  
50     public Usuario() { }  
51
```



Passo 02 - Atualizar a Interface UsuarioRepository (Camada Repository)

Na Interface UsuarioRepository, na camada Repository, vamos criar o método **findAllByNomeContainingIgnoreCase(String nome)** para efetuar alguns testes na Camada Repository. Vamos alterar o código da Interface UsuarioRepository de:

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

    public Optional<Usuario> findByUsuario(String usuario);
}
```

Para:

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

    public Optional<Usuario> findByUsuario(String usuario);

    public List <Usuario> findAllByNomeContainingIgnoreCase(String nome);
}
```

 [Código fonte: UsuarioRepository.java](#)

Passo 03 - Configurações iniciais

1. Configurar a Dependência Spring Testing

Vamos Configurar a Dependência Spring Testting para aceitar apenas a versão mais nova do JUnit 5. No arquivo, **pom.xml**, vamos alterar as linhas atuais:

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

Para:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>org.junit.vintage</groupId>
      <artifactId>junit-vintage-engine</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

*Essa alteração irá ignorar as versões anteriores ao **JUnit 5** (vintage).

2. Adicionar a Dependência do Banco de Dados H2

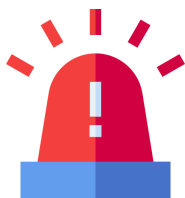
Para utilizar o Banco de Dados H2 no seu projeto será necessário inserir a dependência no seu arquivo **pom.xml**. No arquivo, **pom.xml**, vamos adicionar as linhas abaixo:

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

*Sugerimos adicionar esta dependência logo abaixo da dependência do MySQL.



[Código fonte: pom.xml](#)

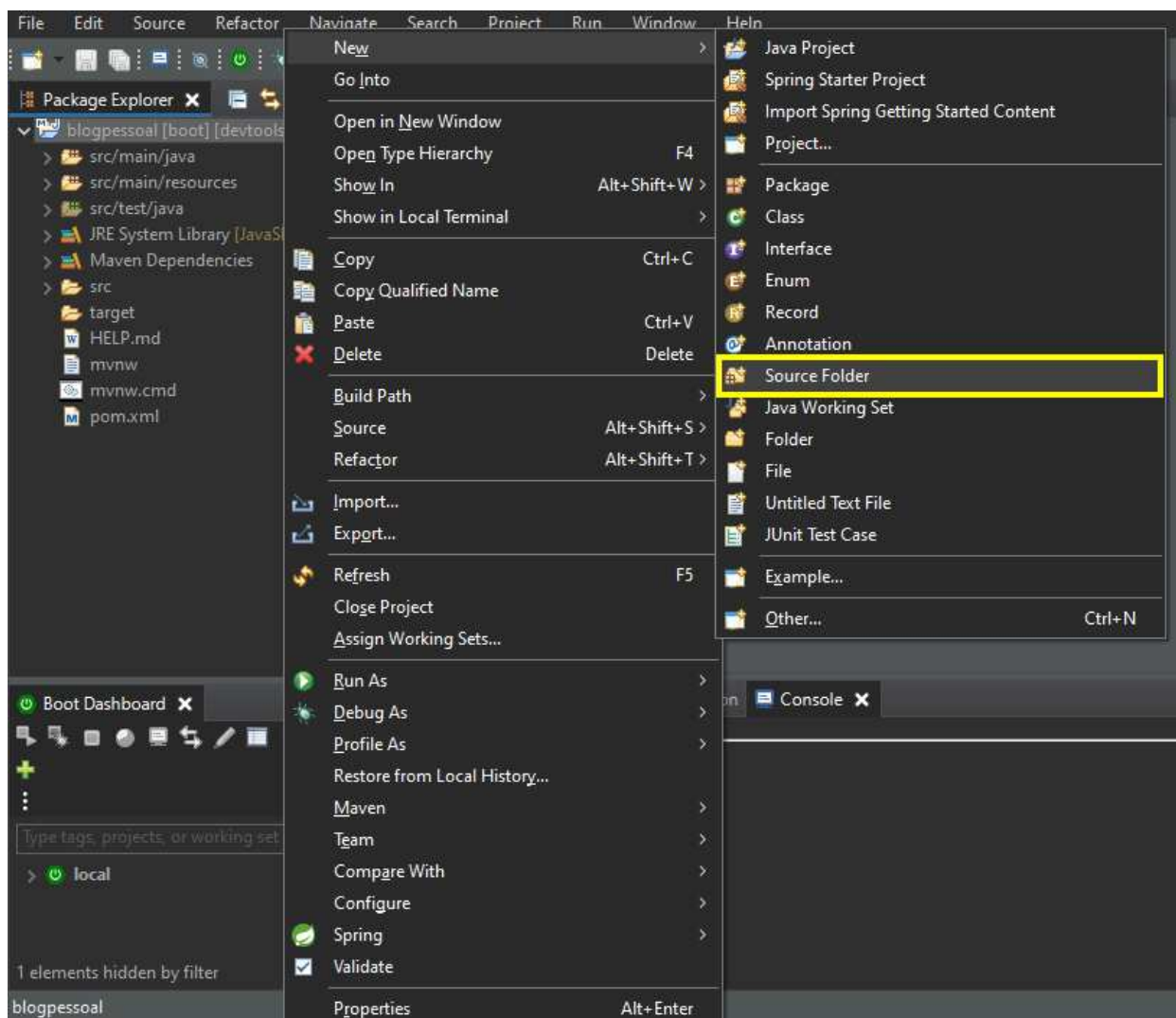


ALERTA DE BSM: *Mantenha a atenção aos detalhes nos próximos passos. Até o Passo 03, todas ações foram realizadas dentro da Source Folder Principal (src/main/java e src/main/resources). A partir do Passo 04, todas as ações serão efetuadas dentro da Source Folder de Testes (src/test/java e src/test/resources).*

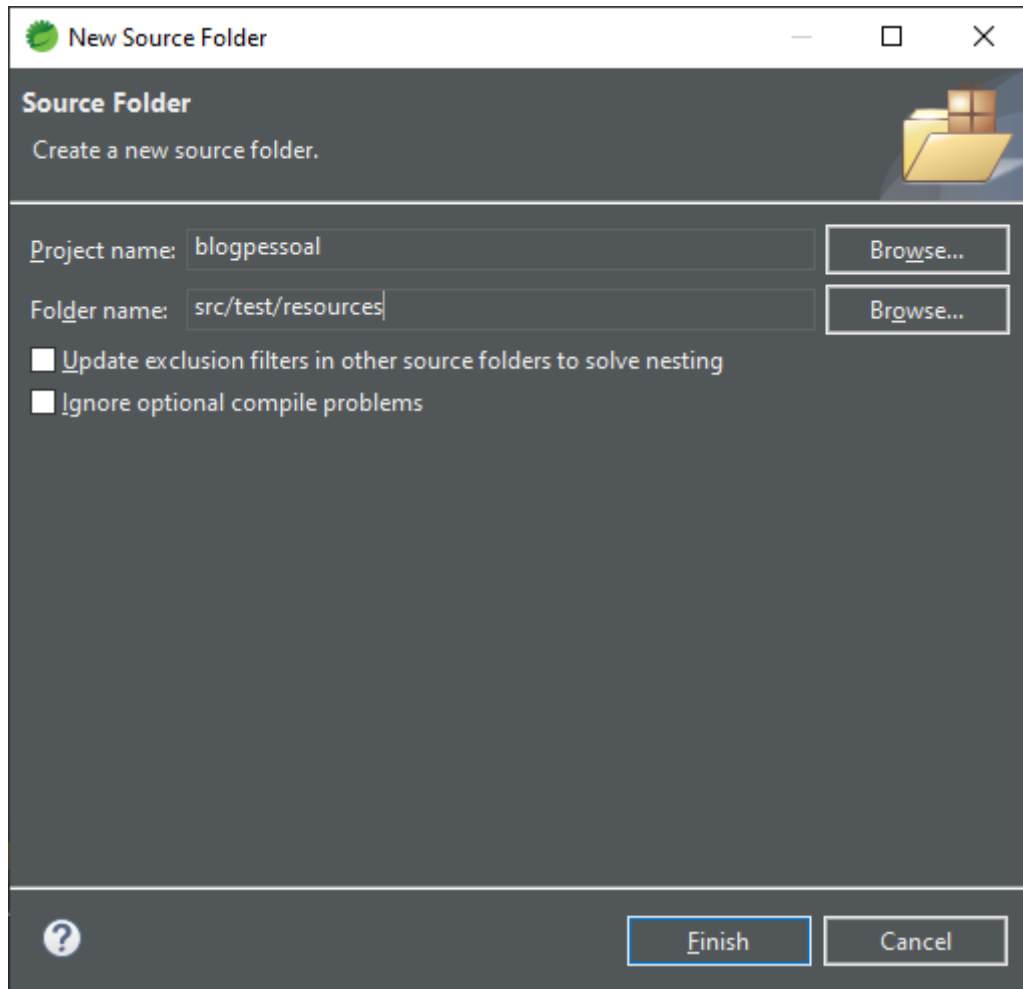
Passo 04 - Configurar o Banco de dados H2

Agora vamos configurar um Banco de dados para executar os testes para não usar o Banco de dados principal da aplicação. Como não temos em nosso projeto a **Source Folder resources**, dentro da **Source Folder src/test**, vamos criá-la e na sequência inserir o arquivo `application.properties` para configurarmos o Banco de dados de testes (H2). Vamos utilizar nos testes o Banco de dados H2 porque não precisaremos persistir os dados dos testes após a sua conclusão.

1. No lado esquerdo superior, na Guia **Package Explorer**, clique sobre a pasta do projeto com o botão direito do mouse e clique na opção **New → Source folder**

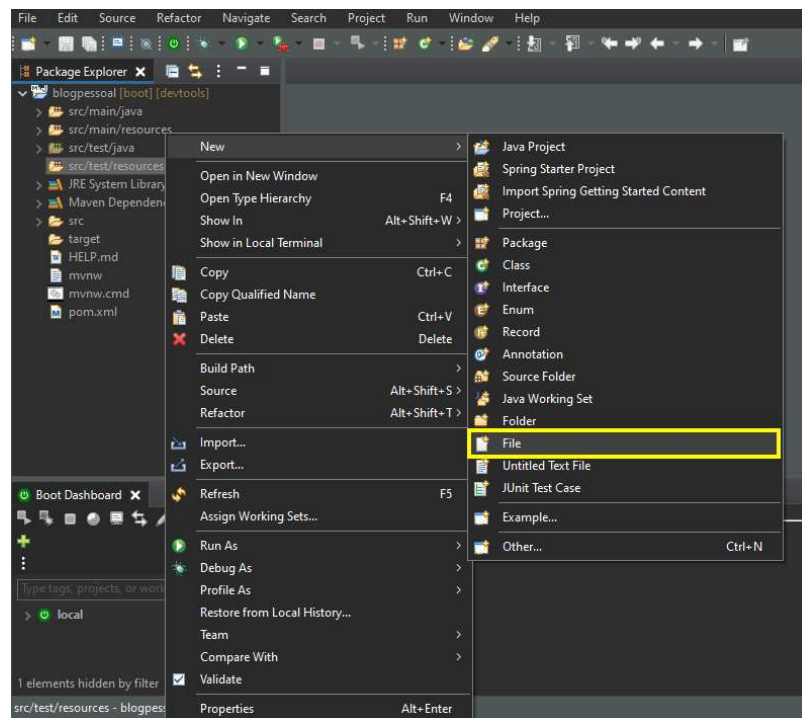


2. Em **Source Folder**, no item **Folder name**, informe o caminho como mostra a figura abaixo (**src/test/resources**), e clique em **Finish**:

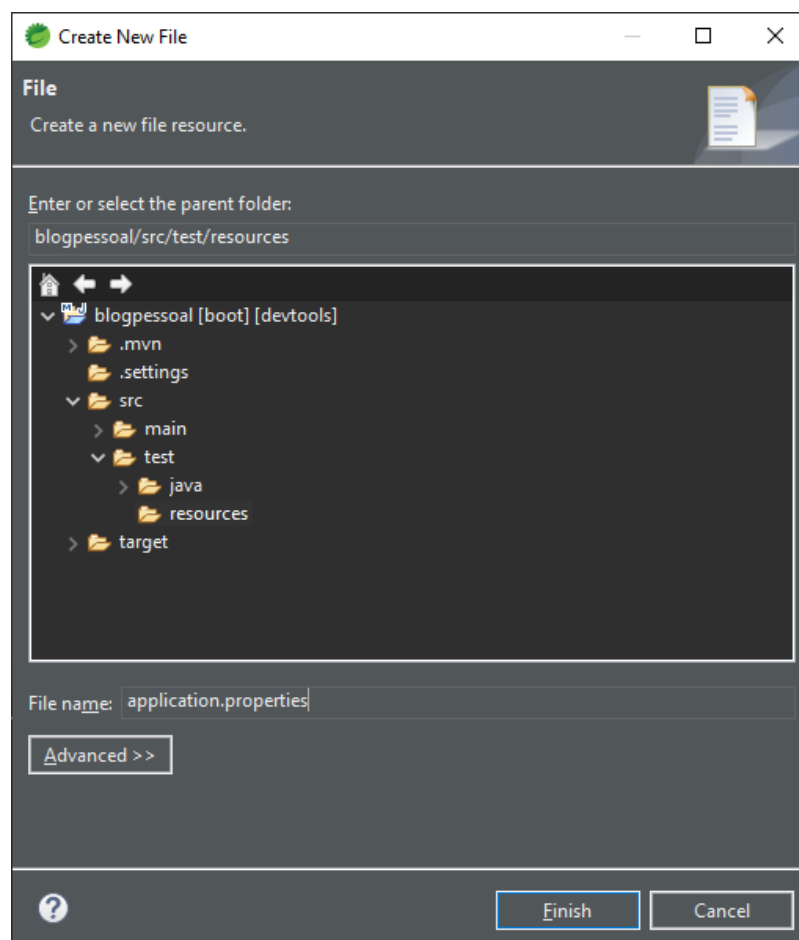


3. Na nova Source Folder (**src/test/resources**) , crie o arquivo **application.properties**, para configurarmos a conexão com o Banco de Dados de testes

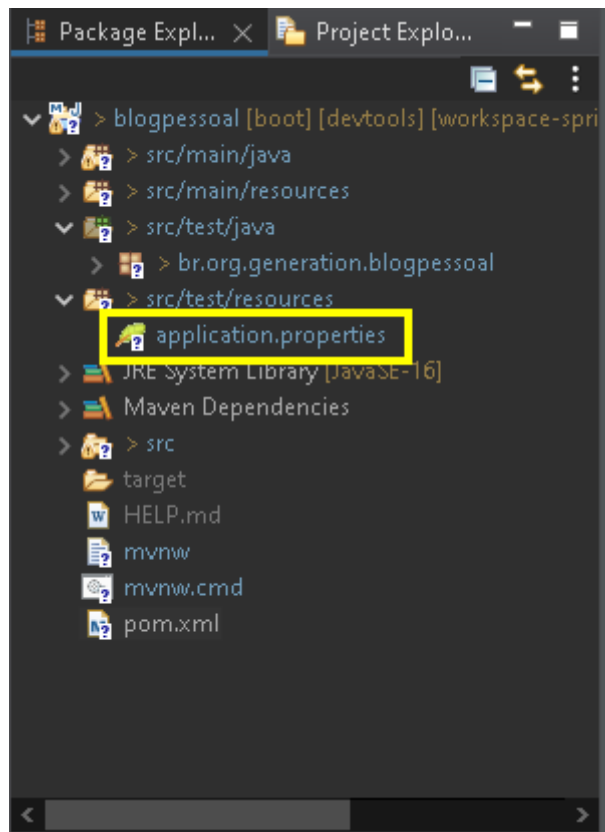
4. No lado esquerdo superior, na Guia **Package explorer**, na Source Folder **src/test/resources**, clique com o botão direito do mouse e clique na opção **New** → **File**.



5. Em File name, digite o nome do arquivo (**application.properties**) e clique em **Finish**.



6. Veja o arquivo criado na **Package Explorer**



7. Insira no arquivo **application.properties** criado em **src/test/resources** o código abaixo, para configurar o Banco de dados H2:

```
spring.datasource.url=jdbc:h2:mem:db_blogpessoal_test
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=sa
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

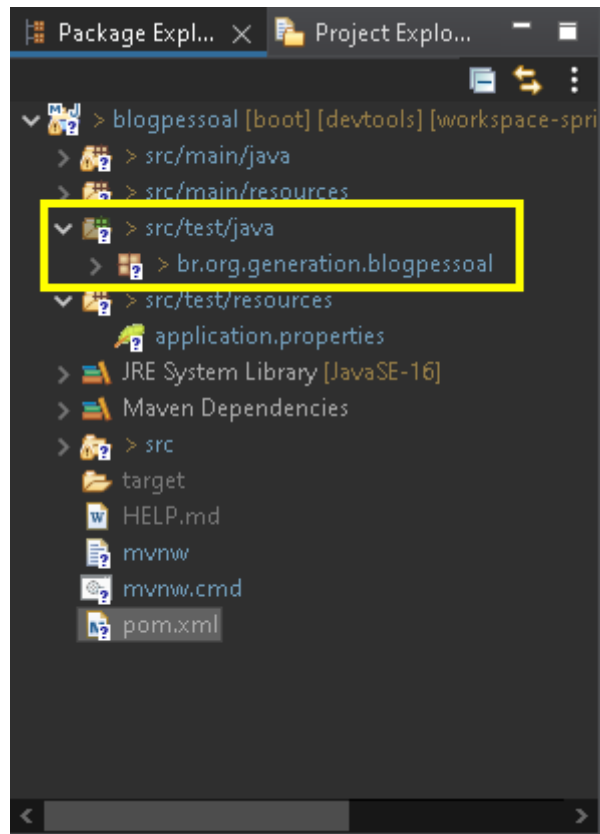
Linha	Descrição
spring.datasource.url	Define o nome do Banco de dados de teste (db_blogpessoal_test)
spring.datasource.driverClassName	Define o Driver do Banco de dados (H2)
spring.datasource.username	Define o usuário do H2 (sa)
spring.datasource.password	Define a senha do usuário do H2 (sa)
spring.jpa.database-platform	Configura o tipo do Banco de dados (H2).



[Código fonte: application.properties \(src/test/resources\)](#)

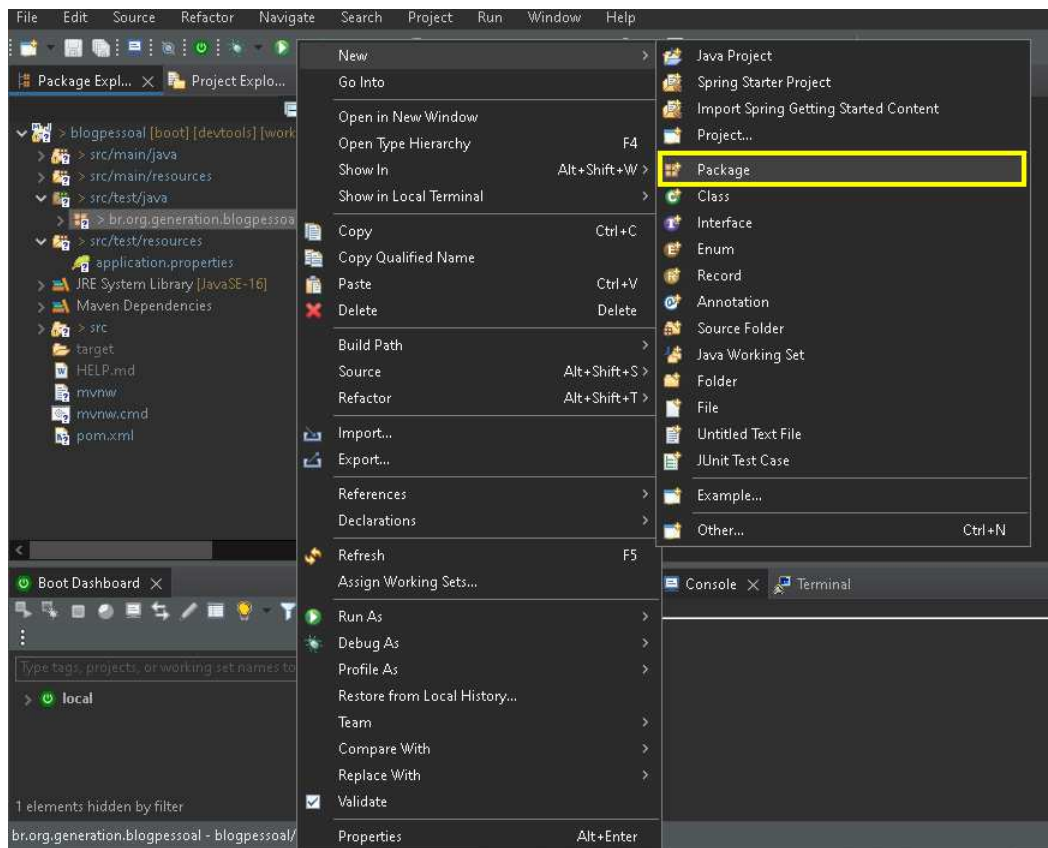
Passo 05 - Criar a estrutura de pacotes na Source Folder de Testes

Na Source Folder de Testes (**src/test/java**), observe que existe a mesma estrutura de pacotes da Source Folder Principal (**src/main/java**) quando criamos o projeto Spring Boot.

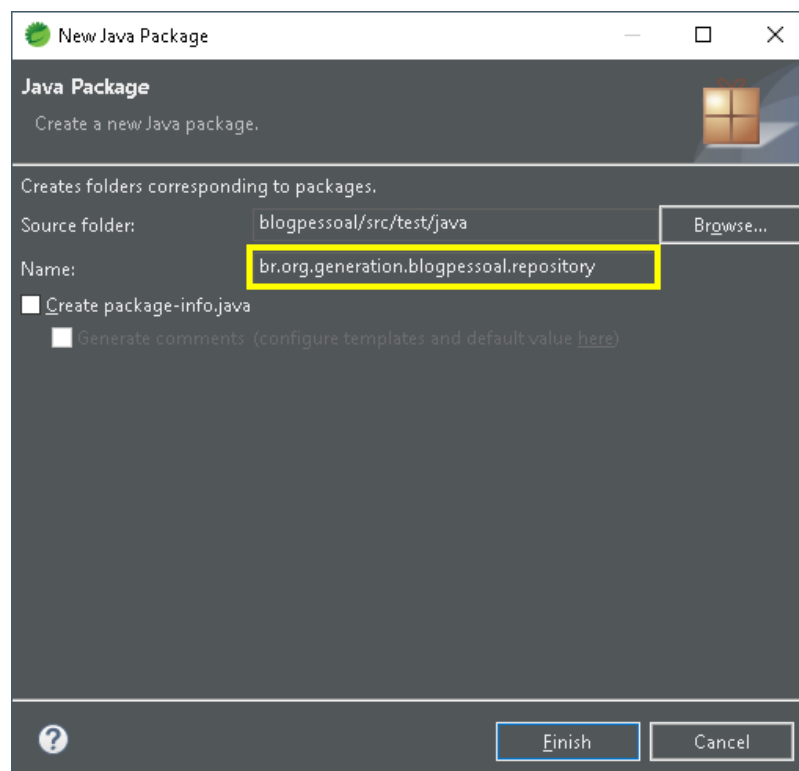


Antes de começarmos a escrever os nossos testes, vamos criar na Source Folder de Testes (**src/test/java**) as packages **Repository** e **Controller**, começando pela Repository:

1. No lado esquerdo superior, na Guia **Package explorer**, clique com o botão direito do mouse sobre a Package **br.org.generation.blogpessoal**, na Source Folder **src/test/java** e clique na opção **New → Package**.



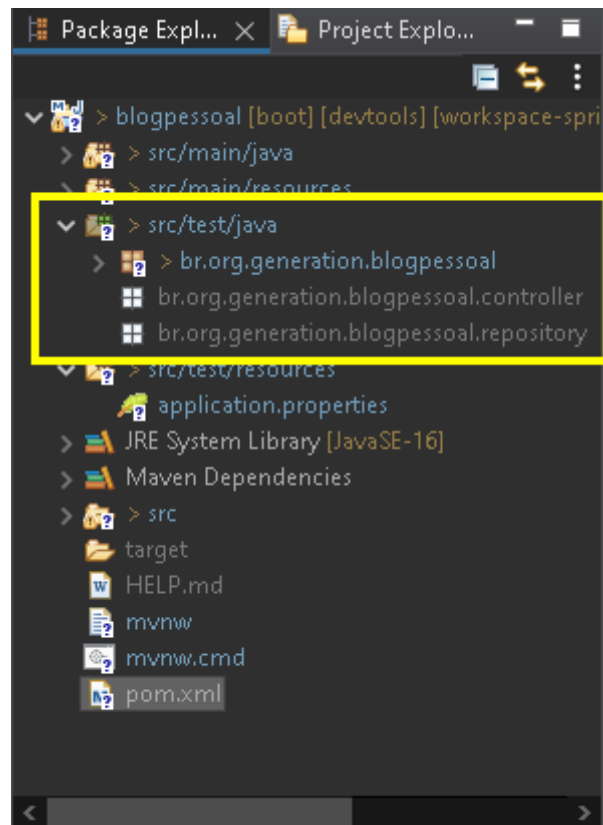
2. Na janela **New Java Package**, no item **Name**, acrescente no final do nome da Package **.repository**, como mostra a figura abaixo:



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

4. **Repita os passos 1-3** para criar a **Package .controller**.

Quando você terminar de criar as novas Packages, a sua estrutura de pacotes ficará semelhante a figura abaixo:

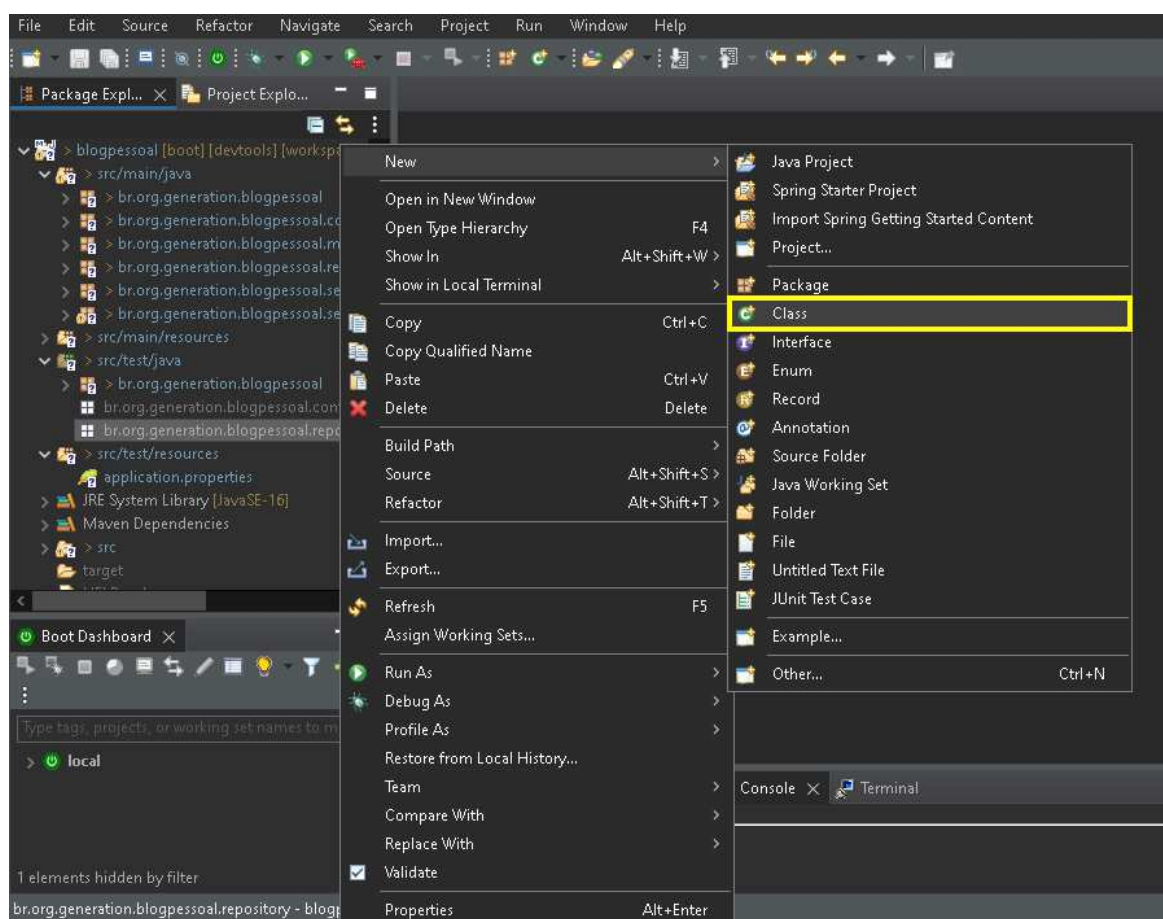


Teste de Software - Spring Boot Testing (JUnit 5 + H2)

Passo 06 - Criar os Testes da Camada Repository

A Classe **UsuarioRepositoryTest** será utilizada para testar a Classe Repository do Usuario. Crie a classe **UsuarioRepositoryTest** na package **repository**, na Source Folder de Testes (**src/test/java**)

1. No lado esquerdo superior, na Guia **Package Explorer**, clique com o botão direito do mouse sobre a Package **br.org.generation.blogpessoal.repository**, na Source Folder **src/test/java** e clique na opção **New → Class**.



2. Na janela **New Java Class**, no item **Name**, informe o nome da classe que será o mesmo nome da Classe Principal (**UsuarioRepository**) + a palavra **Test**, para indicar que se trata de uma Classe de Testes, ou seja, **UsuarioRepositoryTest**, como mostra a figura abaixo:

The image shows the 'New Java Class' dialog box. The 'Name' field is highlighted with a yellow box and contains the text 'UsuarioRepositoryTest'. The 'Package' field contains 'br.org.generation.blogpessoal.repository'. The 'Superclass' field contains 'java.lang.Object'. The 'Which method stubs would you like to create?' section has 'Inherited abstract methods' checked. The 'Finish' button is highlighted in blue.

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

Importante: O Teste da Classe UsuarioRepository, na camada Repository, utiliza o Banco de Dados, entretanto ele não criptografa a senha ao gravar um novo usuário no Banco de dados. O teste da Camada Repository não utiliza a Classe de Serviço UsuarioService, ele utiliza o método save(), da Classe JpaRepository de forma direta.

1. UsuarioRepositoryTest

```
19
20 @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
21 @TestInstance(TestInstance.Lifecycle.PER_CLASS)
22 public class UsuarioRepositoryTest {
23
24     @Autowired
25     private UsuarioRepository usuarioRepository;
26
27     @BeforeAll
28     void start(){
29
30         usuarioRepository.save(new Usuario(0L, "João da Silva", "joao@email.com.br", "13465278"));
31         usuarioRepository.save(new Usuario(0L, "Manuela da Silva", "manuela@email.com.br", "13465278"));
32         usuarioRepository.save(new Usuario(0L, "Adriana da Silva", "adriana@email.com.br", "13465278"));
33         usuarioRepository.save(new Usuario(0L, "Paulo Antunes", "paulo@email.com.br", "13465278"));
34
35     }
36
37 }
38
39
```

Na **linha 20** a anotação **@SpringBootTest** indica que a Classe `UsuarioRepositoryTest` é uma Classe Spring Boot Testing. A Opção **environment** indica que caso a porta principal (8080 para uso local) esteja ocupada, o Spring irá atribuir uma outra porta automaticamente.

Na **linha 21** a anotação **@TestInstance** indica que o Ciclo de vida da Classe de Teste será por Classe.

Nas **linhas 24 e 25** foi injetado (**@Autowired**), um objeto da Interface `UsuarioRepository` para persistir os objetos no Banco de dados de testes.

Entre as **linhas 27 e 38**, o método **start()**, anotado com a anotação **@BeforeAll**, inicializa 4 objetos do tipo `Usuario` e insere no Banco de dados de testes através do método **.save()** uma única vez (**Lifecycle.PER_CLASS**).



[Documentação: @SpringBootTest](#)



[Documentação: @TestInstance](#)



[Documentação: Lifecycle](#)



[Documentação: @BeforeAll](#)

Método 01 - Retornar um Usuário

```
39
40  @Test
41  @DisplayName("Retorna 1 usuario")
42  public void deveRetornarUmUsuario() {
43
44      Optional<Usuario> usuario = usuarioRepository.findByUsuario("joao@email.com.br");
45      assertTrue(usuario.get().getUsuario().equals("joao@email.com.br"));
46  }
47
```

Na **linha 40**, o Método **deveRetornarUmUsuario()** foi anotado com a anotação **@Test** que indica que este método executará um teste.

Na **linha 41**, a anotação **@DisplayName** configura uma mensagem que será exibida ao invés do nome do método

Na **linha 44**, o objeto **usuario** recebe o resultado do método **findByUsuario()**.

Na **linha 45**, através do método de asserção **assertTrue()**, verifica se o usuário cujo e-mail é "joao@email.com.br" foi encontrado. Se o e-mail for encontrado o resultado do teste será **Aprovado!**. Caso não encontre, o resultado do teste será **Falhou!**.



[Documentação: @Test](#)



[Documentação: @DisplayName](#)



[Documentação: assertTrue](#)

Método 02 - Retornar três Usuários

```
47
48  @Test
49  @DisplayName("Retorna 3 usuarios")
50  public void deveRetornarTresUsuarios() {
51
52      List<Usuario> listaDeUsuarios = usuarioRepository.findAllByNameContainingIgnoreCase("Silva");
53      assertEquals(3, listaDeUsuarios.size());
54      assertTrue(listaDeUsuarios.get(0).getNome().equals("João da Silva"));
55      assertTrue(listaDeUsuarios.get(1).getNome().equals("Manuela da Silva"));
56      assertTrue(listaDeUsuarios.get(2).getNome().equals("Adriana da Silva"));
57  }
58
59
```

Na **linha 48**, o Método **deveRetornarTresUsuarios()** foi anotado com a anotação **@Test** que indica que este método executará um teste.

Na **linha 49**, a anotação **@DisplayName** configura uma mensagem que será exibida ao invés do nome do método

Na **linha 52**, o objeto **listaDeUsuarios** recebe o resultado do método **findAllByNameContainingIgnoreCase()**.

Na **linha 53**, através do método de asserção **assertEquals()**, verifica se o tamanho da List é igual a 3 (quantidade de usuários cadastrados no método start()) cujo sobrenome é **“Silva”**). O método **size()**, (java.util.List), retorna o tamanho da List. Se o tamanho da List for igual 3, o 1º teste será **Aprovado!**.

Nas **linhas 54 a 56**, através do método de asserção **AssertTrue()**, verifica em cada posição da Collection List **listaDeUsuarios** se os usuários, que foram inseridos no Banco de dados através no método start(), foram gravados na mesma sequência.

- O Teste da **linha 54** checará se o primeiro usuário inserido (João da Silva) está na posição 0 da List **listaDeUsuarios** (1ª posição da List),
- O Teste da **linha 55** checará se o segundo usuário (Manuela da Silva) está na posição 1 da List **listaDeUsuarios** (2ª posição da List).
- O Teste da **linha 56** checará se o terceiro usuário (Adriana da Silva) está na posição 2 da List **listaDeUsuarios** (3ª posição da List).

A posição na List é obtida através do método **get(int index)** (java.util.List), passando como parâmetro a posição desejada. O nome do usuário é obtido através do método **getNome()** da Classe Usuario. Se os três usuários foram gravados na mesma sequência do método start(), os três testes serão **Aprovados!**.

Documentação: assertEquals



Documentação: Collection List



Documentação: Collection List - Método Size()



Documentação: Collection List - Método get(int index)



Atenção!: cuidado para não confundir o método get(int index) do pacote java.util.List com o método get() do pacote java.util.Optional.



ATENÇÃO: *Para que o método `deveRetornarTresUsuarios()` seja aprovado, os 4 testes (linhas 53 a 56) devem ser aprovados, caso contrário o JUnit indicará que o teste Falhou!.*



Código fonte: [UsuarioRepositoryTest.java](#)



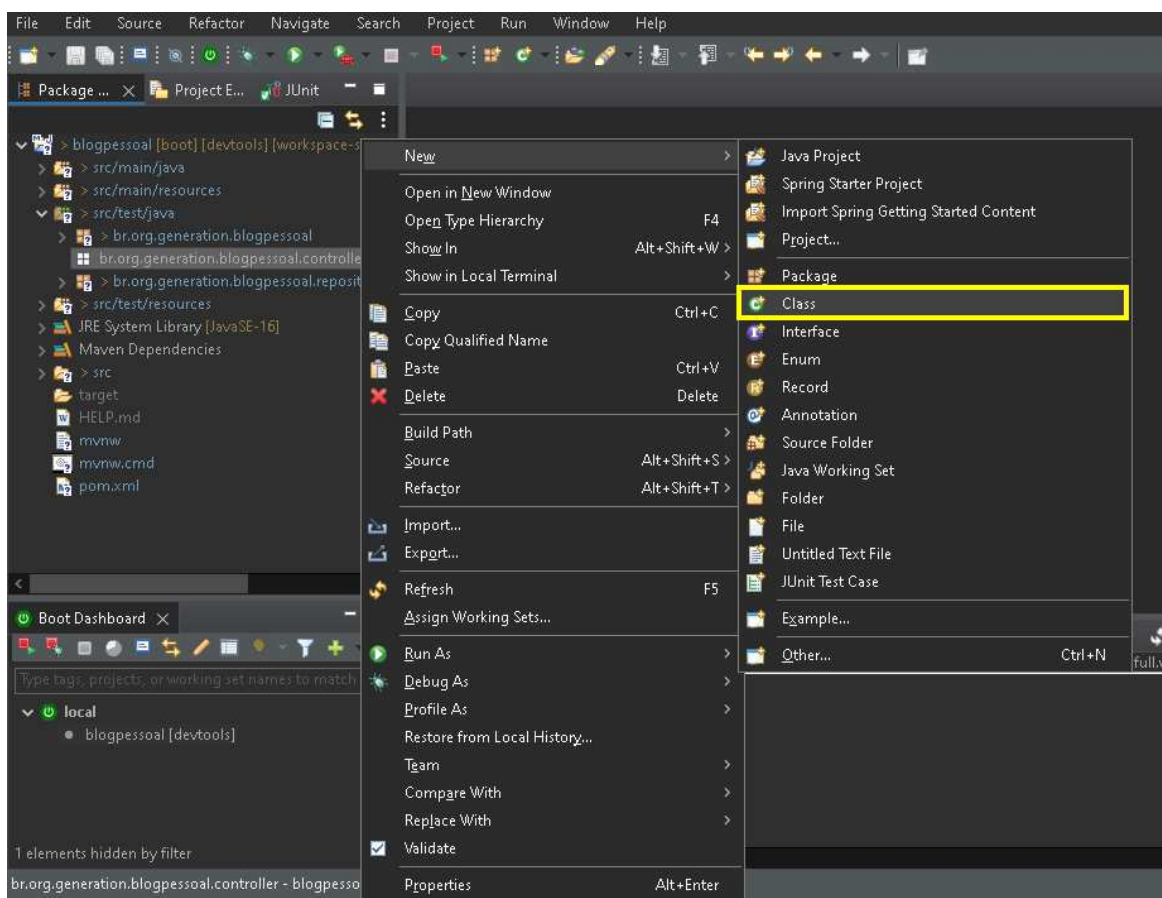
DICA: Faça algumas alterações nos dados dos objetos e/ou escreva outros testes para praticar. A melhor forma de aprender e compreender como funcionam os testes é praticando!

Passo 06 - Testes na Camada Controller

UsuarioControllerTest

A Classe UsuarioControllerTest será utilizada para testar a Classe Controller do Usuario. Crie a classe **UsuarioControllerTest** na package **controller**, na Source Folder de Testes (**src/test/java**)

1. No lado esquerdo superior, na Guia **Package Explorer**, clique com o botão direito do mouse sobre a Package **br.org.generation.blogpessoal.controller**, na Source Folder **src/test/java** e clique na opção **New → Class**.



2. Na janela **New Java Class**, no item **Name**, informe o nome da classe que será o mesmo nome da Classe Principal (**UsuarioController**) + a palavra **Test**, para indicar que se trata de uma Classe de Testes, ou seja, **UsuarioControllerTest**, como mostra a figura abaixo:

The image shows the 'New Java Class' dialog box. The 'Name' field is highlighted with a yellow box and contains the text 'UsuarioControllerTest'. The 'Source folder' is 'blogpessoal/src/test/java', the 'Package' is 'br.org.generation.blogpessoal.controller', and the 'Superclass' is 'java.lang.Object'. The 'Which method stubs would you like to create?' section has 'Inherited abstract methods' checked. The 'Finish' button is highlighted with a blue border.

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

O teste da Camada Controller é um pouco diferente dos testes da Camada Repository porquê faremos Requisições (**http Request**) e na sequencia o teste analisará se as Respostas das Requisições (**http Response**) foram as esperadas.

Para simular as Requisições e Respostas, utilizaremos algumas classes e métodos do Spring Framework:

Classes / Métodos	Descrição
<i>TestRestTemplate()</i>	É um cliente para escrever testes criando um modelo de comunicação com as APIs HTTP. Ele fornece os mesmos métodos, cabeçalhos e outras construções do protocolo HTTP.
<i>HttpEntity()</i>	Representa uma solicitação HTTP ou uma entidade de resposta, composta pelo status da resposta (2XX, 4XX ou 5XX), o corpo (Body) e os cabeçalhos (Headers).
<i>ResponseEntity()</i>	Extensão de <i>HttpEntity</i> que adiciona um código de status (<i>http Status</i>)
<i>TestRestTemplate</i> <i>.exchange(URI,</i> <i>HttpMethod,</i> <i>RequestType,</i> <i>ResponseType)</i>	<p>O método <i>exchange</i> executa uma requisição de qualquer método HTTP e retorna uma instância da Classe <i>ResponseEntity</i>. Ele pode ser usado para criar requisições com os verbos http GET, POST, PUT e DELETE.</p> <p>Usando o método <i>exchange()</i>, podemos realizar todas as operações do CRUD (criar, consultar, atualizar e excluir). Todas as requisições do método <i>exchange()</i> retornarão como resposta um Objeto da Classe <i>ResponseEntity</i>.</p>
<i>TestRestTemplate</i> <i>.withBasicAuth(username,</i> <i>password)</i>	<p>O método withBasicAuth permite efetuar login na aplicação para testar os endpoints protegidos pela Spring Security - Padrão Http Basic. Nos endpoints liberados não é necessário efetuar o login. Para checar os métodos liberados verifique a Classe BasicSecurityConfig.</p> <p>Utilizaremos o usuário em memória (root), que foi criado na Classe <i>BasicSecurityConfig</i>, para executar os nossos testes nos endpoints protegidos.</p>

Vamos analisar o código da Classe UsuarioControllerTest:

```
25
26 @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
27 @TestInstance(TestInstance.Lifecycle.PER_CLASS)
28 @TestMethodOrder(MethodOrderer.OrderAnnotation.class)
29 public class UsuarioControllerTest {
30
31     @Autowired
32     private TestRestTemplate testRestTemplate;
33
34     @Autowired
35     private UsuarioService usuarioService;
36
```

Na **linha 26** a anotação **@SpringBootTest** indica que a Classe UsuarioControllerTest é uma Classe Spring Boot Testing. A Opção **environment** indica que caso a porta principal (8080 para uso local) esteja ocupada, o Spring irá atribuir uma outra porta automaticamente.

Na **linha 27** a anotação **@TestInstance** indica que o Ciclo de vida da Classe de Teste será por Classe.

Na **linha 28** a anotação **@TestMethodOrder** indica em qual ordem os testes serão executados. A opção **MethodOrderer.OrderAnnotation.class** indica que os testes serão executados na ordem indicada pela anotação **@Order** inserida em cada teste. **Exemplo:** **@Order(1)** → indica que este será o primeiro teste que será executado

Nas **linhas 31 e 32** foi injetado (**@Autowired**), um objeto da Classe **TestRestTemplate** para enviar as requisições para a nossa aplicação.

Nas **linhas 34 e 35** foi injetado (**@Autowired**), um objeto da **Classe UsuarioService** para persistir os objetos no Banco de dados de testes com a senha criptografada.

Referências:



[Documentação: @SpringBootTest](#)



[Documentação: @TestInstance](#)



[Documentação: Lifecycle](#)



[Documentação: @TestMethodOrder](#)



[Documentação: Classe TestRestTemplate](#)

Método 01 - Cadastrar Usuário

```
37
38  @Test
39  @Order(1)
40  @DisplayName("Cadastrar Um Usuário")
41  public void deveCriarUmUsuario() {
42
43      HttpEntity<Usuario> requisicao = new HttpEntity<Usuario>(new Usuario(0L,
44          "Paulo Antunes", "paulo_antunes@email.com.br", "13465278"));
45
46      ResponseEntity<Usuario> resposta = testRestTemplate
47          .exchange("/usuarios/cadastrar", HttpMethod.POST, requisicao, Usuario.class);
48
49      assertEquals(HttpStatus.CREATED, resposta.getStatusCode());
50      assertEquals(requisicao.getBody().getName(), resposta.getBody().getName());
51      assertEquals(requisicao.getBody().getUsuario(), resposta.getBody().getUsuario());
52  }
53
```

Na **linha 38**, o Método **deveCriarUmUsuario()** foi anotado com a anotação **@Test** que indica que este método executará um teste.

Na **linha 39**, a anotação **@Order(1)** indica que o método será o primeiro a ser executado.

Na **linha 40**, a anotação **@DisplayName** configura uma mensagem que será exibida ao invés do nome do método.

Na **linha 43**, foi criado um objeto da Classe **HttpEntity** chamado **requisicao**, recebendo um objeto da Classe **Usuario**. Nesta etapa, o processo é equivalente ao que o Postman faz em uma requisição do tipo **POST**: Transforma os atributos num objeto da Classe **Usuario**, que será enviado no corpo da requisição (Request Body).

Na **linha 46**, a Requisição HTTP será enviada através do método **exchange()** da Classe **TestRestTemplate** e a Resposta da Requisição (Response) será recebida pelo objeto **resposta** do tipo **ResponseEntity**. Para enviar a requisição, o será necessário passar 4 parâmetros:

- **A URI**: Endereço do endpoint (/usuarios/cadastrar);
- **O Método HTTP**: Neste exemplo o método **POST**;
- **O Objeto HttpEntity**: Neste exemplo o objeto **requisicao**, que contém o objeto da Classe **Usuario**;
- **O conteúdo esperado no Corpo da Resposta (Response Body)**: Neste exemplo será do tipo **Usuario (Usuario.class)**.











Na **linha 49**, através do método de asserção **AssertEquals()**, checaremos se a resposta da requisição (Response), é a resposta esperada (**CREATED → 201**). Para obter o status da resposta vamos utilizar o método **getStatusCode()** da **Classe ResponseEntity**.

Nas **linhas 50 e 51**, através do método de asserção **AssertEquals()**, checaremos se o nome e o usuário(e-mail) enviados na requisição foram persistidos no Banco de Dados. Através do método **getBody()** faremos o acesso aos objetos requisição e resposta, e através dos métodos **getNome()** e **getUsuario()** faremos o acesso aos atributos que serão comparados.



ATENÇÃO: Para que o método *deveCriarUmUsuario()* seja aprovado, os 3 testes (linhas 49 a 51) devem ser aprovados, caso contrário o JUnit indicará que o teste Falhou!.

Referências:

-  [Documentação: @Test](#)
-  [Documentação: @Order](#)
-  [Documentação: @DisplayName](#)
-  [Documentação: Classe HttpEntity](#)
-  [Documentação: Classe TestRestTemplate](#)
-  [Documentação: Classe TestRestTemplate - Método .exchange\(\)](#)
-  [Documentação: Classe ResponseEntity](#)
-  [Documentação: HttpMethod](#)
-  [Documentação: HttpStatus](#)
-  [Documentação: assertEquals](#)
-  [Documentação: Classe HttpEntity](#)
-  [Documentação: Classe HttpEntity - Método getBody\(\)](#)

Método 02 - Não deve permitir duplicação do Usuário

```
54
55  @Test
56  @Order(2)
57  @DisplayName("Não deve permitir duplicação do Usuário")
58  public void naoDeveDuplicarUsuario() {
59
60      usuarioService.cadastrarUsuario(new Usuario(0L,
61          "Maria da Silva", "maria_silva@email.com.br", "13465278"));
62
63      HttpEntity<Usuario> requisicao = new HttpEntity<Usuario>(new Usuario(0L,
64          "Maria da Silva", "maria_silva@email.com.br", "13465278"));
65
66      ResponseEntity<Usuario> resposta = testRestTemplate
67          .exchange("/usuarios/cadastrar", HttpMethod.POST, requisicao, Usuario.class);
68
69      assertEquals(HttpStatus.BAD_REQUEST, resposta.getStatusCode());
70  }
71
```

Na **linha 55**, o Método **naoDeveDuplicarUsuario()** foi anotado com a anotação **@Test** que indica que este método executará um teste.

Na **linha 56**, a anotação **@Order(2)** indica que o método será o segundo a ser executado

Na **linha 75**, a anotação **@DisplayName** configura uma mensagem que será exibida ao invés do nome do método

Na **linha 60**, através do método **cadastrarUsuario()** da **Classe UsuarioService**, foi persistido um Objeto da Classe **Usuario** no Banco de dados (Maria da Silva).

Na **linha 63**, foi criado um objeto **HttpEntity** chamado **requisicao**, recebendo um objeto da Classe **Usuario** **contendo os mesmos dados do objeto persistido na linha 60** (Maria da Silva).

Na **linha 66**, a Requisição HTTP será enviada através do método **exchange()** da Classe **TestRestTemplate** e a Resposta da Requisição (Response) será recebida pelo objeto **resposta** do tipo **ResponseEntity**. Para enviar a requisição, o será necessário passar 4 parâmetros:

- **A URI:** Endereço do endpoint (/usuarios/cadastrar);
- **O Método HTTP:** Neste exemplo o método **POST**;
- **O Objeto HttpEntity:** Neste exemplo o objeto **requisicao**, que contém o objeto da Classe **Usuario**;
- **O conteúdo esperado no Corpo da Resposta (Response Body):** Neste exemplo será do tipo **Usuario (Usuario.class)**.

Na **linha 69**, através do método de asserção **AssertEquals()**, checaremos se a resposta da requisição (Response), é a resposta esperada (**BAD_REQUEST → 400**). Para obter o status da resposta vamos utilizar o método **getStatusCode()** da **Classe ResponseEntity**.



Observe que neste método temos o objetivo de testar o Erro! (Usuário Duplicado) e não a persistência dos dados. Observe que enviamos o mesmo objeto 2 vezes e verificamos se o aplicativo rejeita a persistência do mesmo objeto pela segunda vez (BAD REQUEST).

Como o teste tem por objetivo checar se está duplicando usuários no Banco de dados, ao invés de checarmos se o objeto foi persistido (**CREATE → 201**), checaremos se ele não foi persistido (**BAD_REQUEST → 400**). Se retornar o Status 400, o teste será aprovado!

Método 03 - Alterar um Usuário

```
72
73  @Test
74  @Order(3)
75  @DisplayName("Alterar um Usuário")
76  public void deveAtualizarUmUsuario() {
77
78      Optional<Usuario> usuarioCreate = usuarioService.cadastrarUsuario(new Usuario(0L,
79          "Juliana Andrews", "juliana_andrews@email.com.br", "juliana123"));
80
81      Usuario usuarioUpdate = new Usuario(usuarioCreate.get().getId(),
82          "Juliana Andrews Ramos", "juliana_amos@email.com.br", "juliana123");
83
84      HttpEntity<Usuario> requisicao = new HttpEntity<Usuario>(usuarioUpdate);
85
86      ResponseEntity<Usuario> resposta = testRestTemplate
87          .withBasicAuth("root", "root")
88          .exchange("/usuarios/atualizar", HttpMethod.PUT, requisicao, Usuario.class);
89
90      assertEquals(HttpStatus.OK, resposta.getStatusCode());
91      assertEquals(usuarioUpdate.getNome(), resposta.getBody().getNome());
92      assertEquals(usuarioUpdate.getUsuario(), resposta.getBody().getUsuario());
93  }
94
```

Na **linha 73**, o Método **deveAtualizarUmUsuario()** foi anotado com a anotação **@Test** que indica que este método executará um teste.

Na **linha 74**, a anotação **@Order(3)** indica que o método será o terceiro a ser executado

Na **linha 75**, a anotação **@DisplayName** configura uma mensagem que será exibida ao invés do nome do método

Na **linha 78**, foi criado um Objeto **Optional**, do tipo **Usuario**, chamado **usuarioCreate**, para armazenar o resultado da persistência de um Objeto da Classe **Usuario** no Banco de dados, através do método **cadastrarUsuario()** da Classe **UsuarioService**.

Na **linha 81**, foi criado um Objeto do tipo **Usuario**, chamado **usuarioUpdate**, que será utilizado para atualizar os dados persistidos no Objeto **usuarioCreate** (linha 78).

Na **linha 84**, foi criado um objeto **HttpEntity** chamado **requisicao**, recebendo o objeto da Classe **Usuario** chamado **usuarioUpdate**. Nesta etapa, o processo é equivalente ao que o Postman faz em uma requisição do tipo **PUT**: Transforma os atributos num objeto da Classe **Usuario**, que será enviado no corpo da requisição (Request Body).

Na **linha 86**, a Requisição HTTP será enviada através do método **exchange()** da Classe **TestRestTemplate** e a Resposta da Requisição (Response) será recebida pelo objeto **resposta** do tipo **ResponseEntity**. Para enviar a requisição, o será necessário passar 4 parâmetros:

- **A URI**: Endereço do endpoint (/usuarios/atualizar);
- **O Método HTTP**: Neste exemplo o método **PUT**;
- **O Objeto HttpEntity**: Neste exemplo o objeto **requisicao**, que contém o objeto da Classe **Usuario**;
- **O conteúdo esperado no Corpo da Resposta (Response Body)**: Neste exemplo será do tipo **Usuario** (**Usuario.class**).

Observe que na **linha 87**, como o Blog Pessoal está com o **Spring Security** habilitado com autenticação do tipo **Http Basic**, o Objeto **testRestTemplate** dos endpoints que exigem autenticação, deverá efetuar o login com um usuário e uma senha válida para realizar os testes. Para autenticar o usuário e a senha utilizaremos o método **withBasicAuth(user, password)** da Classe **TestRestTemplate**. Como criamos o usuário em memória (root), na **Classe BasicSecurityConfig**, vamos usá-lo para autenticar o nosso teste.

Na **linha 90**, através do método de asserção **AssertEquals()**, checaremos se a resposta da requisição (Response), é a resposta esperada (**OK → 200**). Para obter o status da resposta vamos utilizar o método **getStatusCode()** da **Classe ResponseEntity**.

Nas **linhas 91 e 92**, através do método de asserção **AssertEquals()**, checaremos se o nome e o usuário(e-mail) enviados na requisição **usuarioUpdate** foram persistidos no Banco de Dados. Através do método **getBody()** faremos o acesso aos objetos **usuarioUpdate** e **resposta**, e através dos métodos **getNome()** e **getUsuario()** faremos o acesso aos atributos que serão comparados.



ATENÇÃO: Para que o método `deveAtualizarUmUsuario()` seja aprovado, os 3 testes (linhas 90 a 92) devem ser aprovados, caso contrário o JUnit indicará que o teste *Falhou!*.

Referências:



[Documentação: Classe `TestRestTemplate` - Método `.withBasicAuth\(\)`](#)

Método 04 - Listar todos os Usuários

```
95
96  @Test
97  @Order(4)
98  @DisplayName("Listar todos os Usuários")
99  public void deveMostrarTodosUsuarios() {
100
101      usuarioService.cadastrarUsuario(new Usuario(0L,
102          "Sabrina Sanches", "sabrina_sanches@email.com.br", "sabrina123"));
103
104      usuarioService.cadastrarUsuario(new Usuario(0L,
105          "Ricardo Marques", "ricardo_marques@email.com.br", "ricardo123"));
106
107      ResponseEntity<String> resposta = testRestTemplate
108          .withBasicAuth("root", "root")
109          .exchange("/usuarios/all", HttpMethod.GET, null, String.class);
110
111      assertEquals(HttpStatus.OK, resposta.getStatusCode());
112  }
113
```

Na **linha 96**, o Método `deveMostrarTodosUsuarios()` foi anotado com a anotação `@Test` que indica que este método executará um teste.

Na **linha 97**, a anotação `@Order(4)` indica que o método será o quarto a ser executado

Na **linha 98**, a anotação `@DisplayName` configura uma mensagem que será exibida ao invés do nome do método

Na **linhas 101 e 104**, foram persistidos dois Objetos da Classe `Usuario` no Banco de dados, através do método `cadastrarUsuario()` da Classe `UsuarioService`.

Na **linha 107**, a Requisição HTTP será enviada através do método `exchange()` da Classe `TestRestTemplate` e a Resposta da Requisição (`Response`) será recebida pelo objeto `resposta` do tipo `ResponseEntity`. Para enviar a requisição, o será necessário passar 4 parâmetros:

- **A URI:** Endereço do endpoint (/usuarios/all);
- **O Método HTTP:** Neste exemplo o método **GET**;
- **O Objeto HttpEntity:** O objeto será nulo (null). **Requisições do tipo GET não enviam Objeto no corpo da requisição;**
- **O conteúdo esperado no Corpo da Resposta (Response Body):** Neste exemplo como o objeto da requisição é nulo, a resposta esperada será do tipo **String (String.class)**.

Observe que na **linha 108**, como o Blog Pessoal está com o **Spring Security** habilitado com autenticação do tipo **Http Basic**, o Objeto **testRestTemplate** dos endpoints que exigem autenticação, deverá efetuar o login com um usuário e uma senha válida para realizar os testes. Para autenticar o usuário e a senha utilizaremos o método **withBasicAuth(user, password)** da Classe **TestRestTemplate**. Como criamos o usuário em memória (root), na **Classe BasicSecurityConfig**, vamos usá-lo para autenticar o nosso teste.



Observe que no Método GET não foi criada uma requisição. Requisição do tipo GET não envia um Objeto no Corpo da Requisição. Lembre-se: Ao criar uma requisição do tipo GET no Postman é enviado apenas a URL do endpoint. Esta regra também vale para o Método DELETE.

Na **linha 111**, através do método de asserção **AssertEquals()**, checaremos se a resposta da requisição (Response), é a resposta esperada (**OK → 200**). Para obter o status da resposta vamos utilizar o método **getStatusCode()** da **Classe ResponseEntity**.

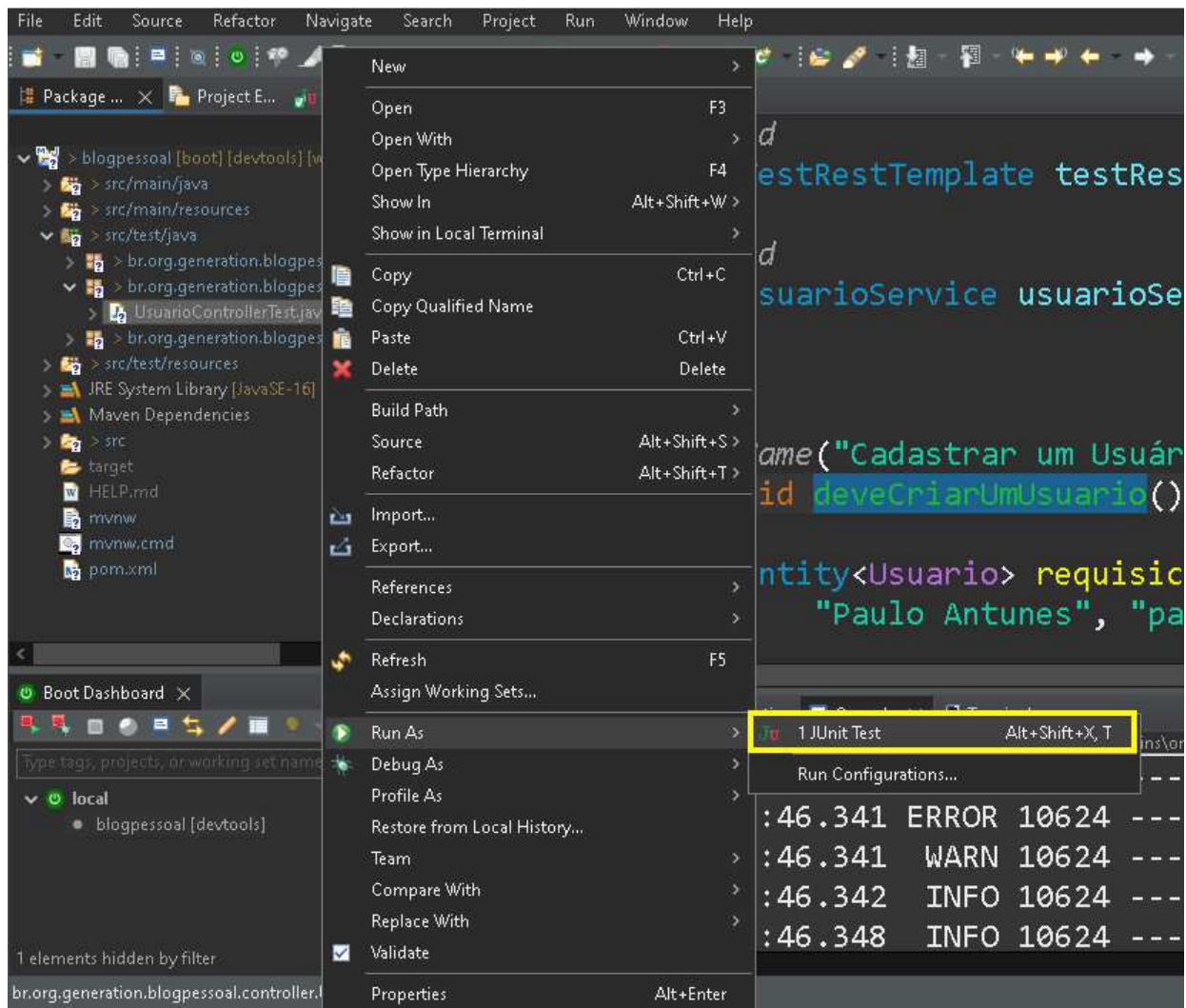


[Código fonte: UsuarioControllerTest.java](#)

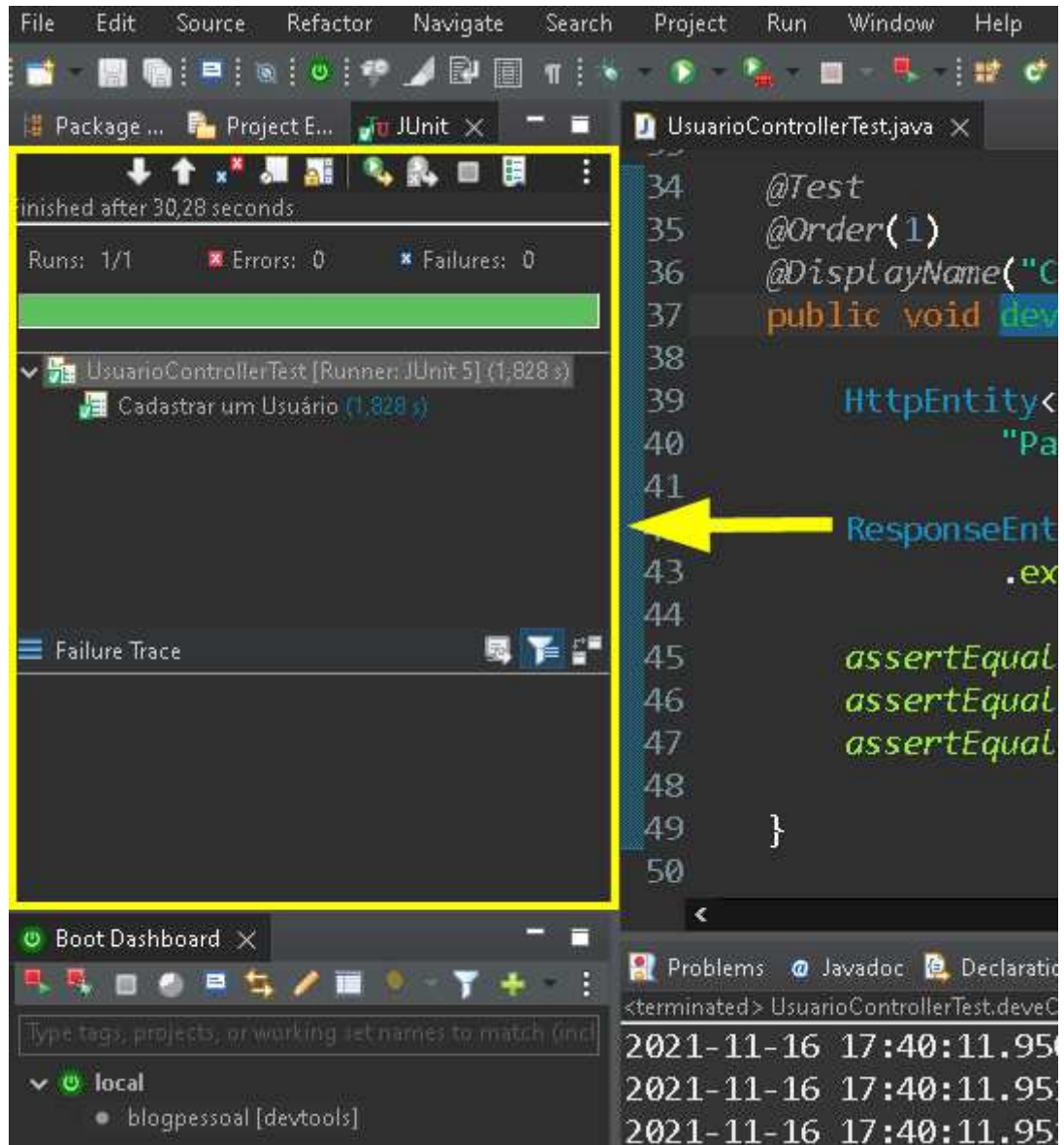
Passo 07 - Executando os Testes no STS

Executar todos os testes

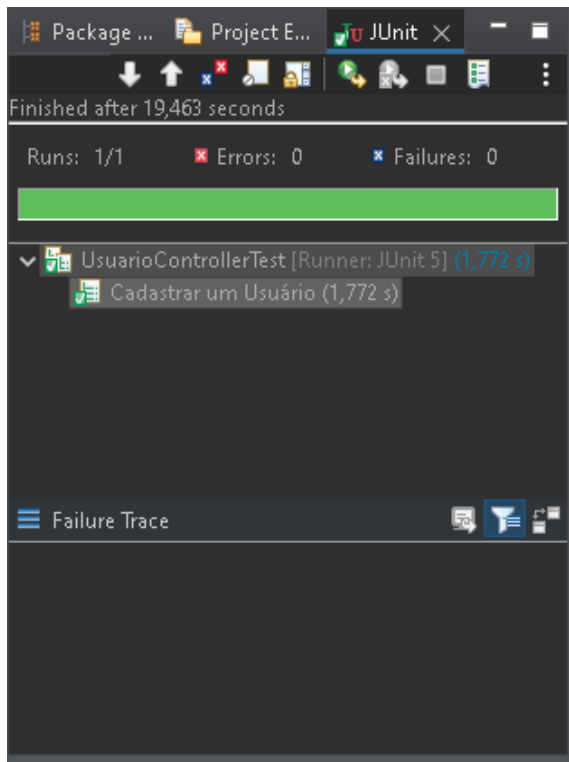
1. No lado esquerdo superior, na Guia **Project**, na Package **src/test/java**, clique com o botão direito do mouse sobre a Classe de teste que você deseja executar e clique na opção **Run As → JUnit Test**.



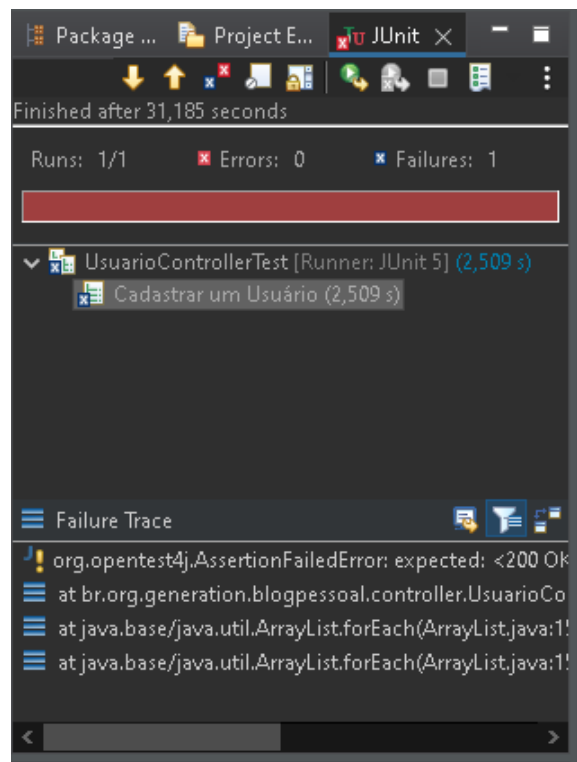
2. Para acompanhar os testes, ao lado da Guia **Project**, clique na Guia **JUnit** (Indicada na figura pela seta amarela).



- Se todos os testes passarem, a Guia do JUnit ficará com uma faixa verde (janela 01). Caso algum teste não passe, a Guia do JUnit ficará com uma faixa vermelha (janela 02). Neste caso, observe o item **Failure Trace** para identificar o (s) erro (s).



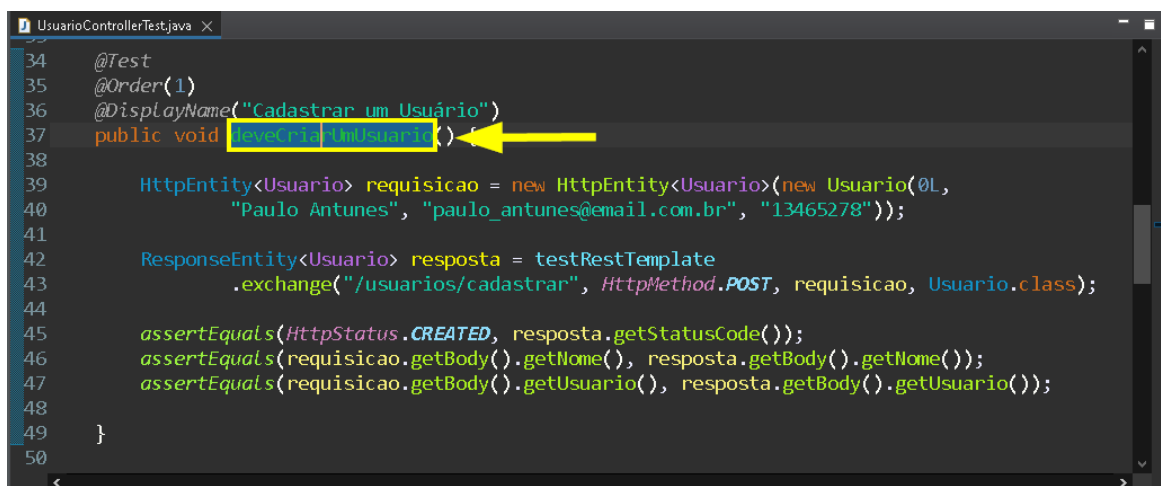
Janela 01: Testes aprovados.



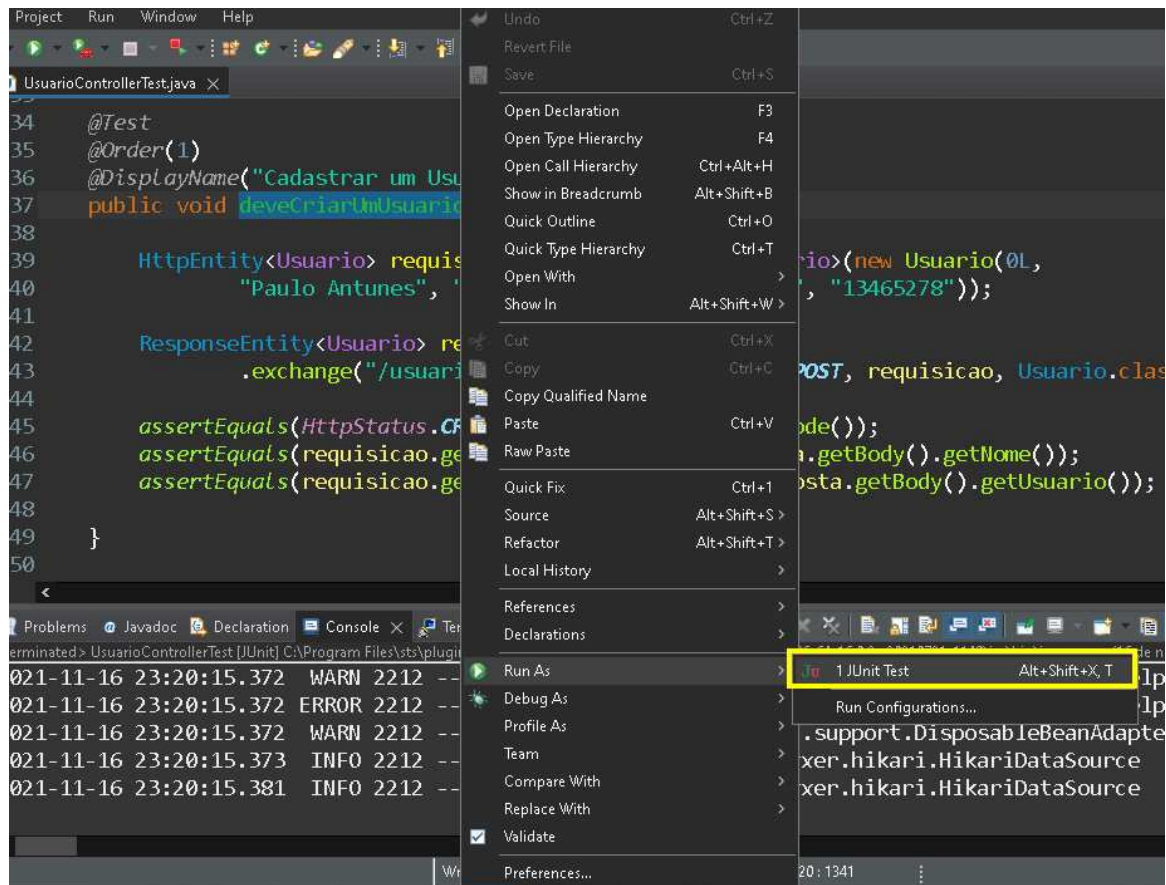
Janela 02: Testes reprovados.

Executar apenas um método específico

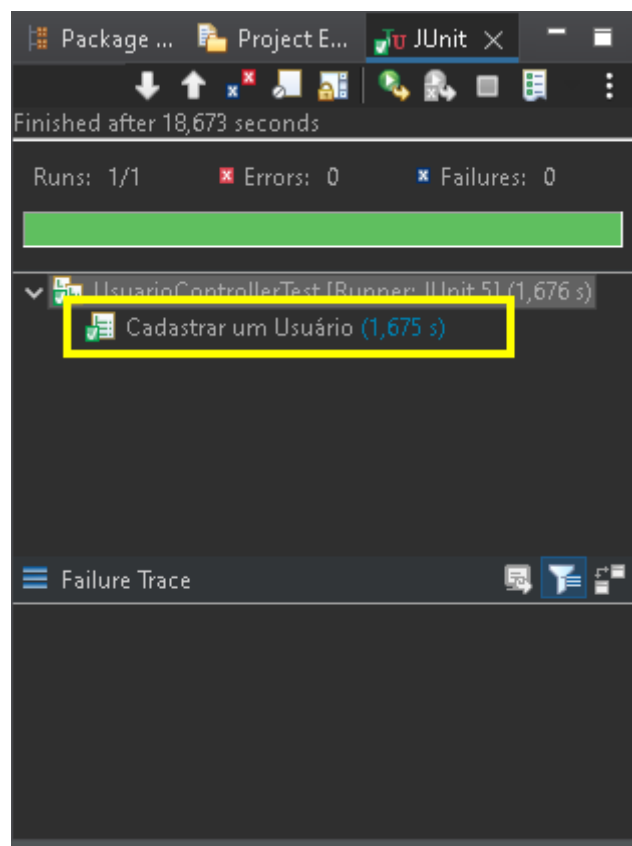
- Posicione o cursor do mouse sobre o nome do teste. Observe que o nome será selecionado, como mostra a figura abaixo:



2. Clique com o botão direito do mouse sobre o nome do Método que você deseja executar e clique na opção **Run As → JUnit Test**.



3. Observe que será executado apenas o Método que você selecionou.





DICA: Faça algumas alterações nos dados dos objetos e/ou escreva outros testes para praticar. A melhor forma de aprender e compreender como funcionam os testes é praticando!



[Código fonte: Projeto Finalizado](#)

✓ Boas práticas

1. **Faça testes pequenos.**
2. **Faça testes rápidos:** Os testes devem ser simples e objetivos porque serão executados o tempo todo.
3. **Faça testes determinísticos:** O teste deve garantir o resultado.
4. **Faça testes independentes:** Um teste não pode depender do resultado de outro teste.
5. **Utilize nomes auto descritivos:** A ideia é que você entenda o que o teste faz sem precisar abri-lo.
6. **Insira poucas asserções em cada teste:** O objetivo é que um teste seja responsável por apenas uma verificação.
7. **Sempre avalie os resultados dos seus testes.**