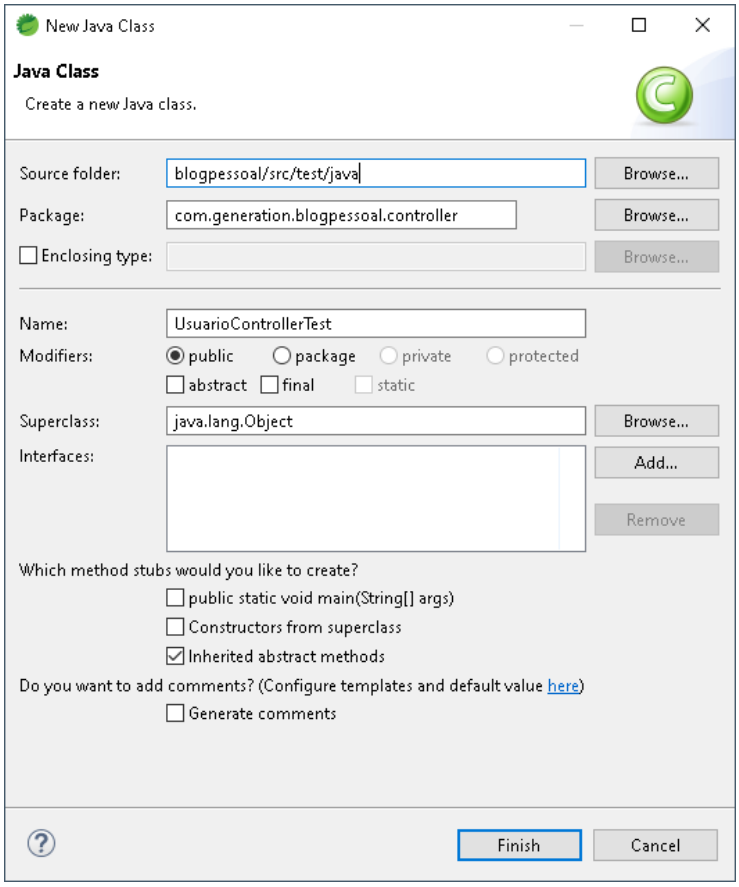
Teste de Software - JUnit 5 - Teste da Classe Controller

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 **com.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:



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
TestRestTemplate()	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.
HttpEntity()	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).
ResponseEntity()	Extensão de HttpEntity que adiciona um código de status (http Status)
TestRestTemplate .exchange(URI, HttpMethod, RequestType, ResponseType)	O método exchange executa uma requisição de qualquer método HTTP e retorna uma instância da Classe ResponseEntity. Ele pode ser usado para criar requisições com os verbos http **GET, POST, PUT e DELETE**. Usando o método exchange(), podemos realizar todas as operações do CRUD (criar, consultar, atualizar e excluir). Todas as requisições do método exchange() retornarão como resposta um Objeto da Classe ResponseEntity.
TestRestTemplate .withBasicAuth(username, password)	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**. Utilizaremos o usuário em memória (root), que foi criado na Classe BasicSecurityConfig, para execuatr os nossos testes nos endpoints protegidos.

Vamos analisar o código da Classe UsuarioControllerTest:

```
26
27@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
28@TestInstance(TestInstance.Lifecycle.PER_CLASS)
29@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
30public class UsuarioControllerTest {
31
32    @Autowired
33    private TestRestTemplate testRestTemplate;
34
35    @Autowired
36    private UsuarioService usuarioService;
37
38    @Autowired
39    private UsuarioRepository usuarioRepository;
40
41    @BeforeAll
42    void start(){
43
44        usuarioRepository.deleteAll();
45    }
46
```

Na **linha 27** 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 28** a anotação **@TestInstance** indica que o Ciclo de vida da Classe de Teste será por Classe.

Na **linha 29** 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 32 e 33** foi foi injetado (**@Autowired**), um objeto da Classe **TestRestTemplate** para enviar as requisições para a nossa aplicação.

Nas **linhas 35 e 36** foi injetado (**@Autowired**), um objeto da **Classe UsuarioService** para persistir os objetos no Banco de dados de testes com a senha criptografada.

Nas **linhas 38 e 39** foi foi injetado (**@Autowired**), um objeto da Interface UsuarioRepository para limpar o Banco de dados de testes.

Entre as **linhas 42 e 45**, o método **start()**, anotado com a anotação **@BeforeAll**, apaga todos os dados da tabela.

Documentação: @SpringBootTest
Documentação: @TestInstance
Documentação: Lifecycle
Documentação: @TestMethodOrder
Documentação: *Classe TestRestTemplate*
Documentação: *deleteAll()*

Método 01 - Cadastrar Usuário

```
46
47      @Test
48      @Order(1)
49      @DisplayName("Cadastrar Um Usuário")
50      public void deveCriarUmUsuario() {
51
52          HttpEntity<Usuario> requisicao = new HttpEntity<Usuario>(new Usuario(0L,
53              "Paulo Antunes", "https://i.imgur.com/FETvs20.jpg", "paulo_antunes@email.com.br", "13465278"));
54
55          ResponseEntity<Usuario> resposta = testRestTemplate
56              .exchange("/usuarios/cadastrar", HttpMethod.POST, requisicao, Usuario.class);
57
58          assertEquals(HttpStatus.CREATED, resposta.getStatusCode());
59          assertEquals(requisicao.getBody().getNome(), resposta.getBody().getNome());
60          assertEquals(requisicao.getBody().getFoto(), resposta.getBody().getFoto());
61          assertEquals(requisicao.getBody().getUsuario(), resposta.getBody().getUsuario());
62      }
63
```

Na **linha 47**, o Método **deveCriarUmUsuario()** foi antotado com a anotação **@Test** que indica que este método executará um teste.

Na **linha 48**, a anotação **@Order(1)** indica que o método será o primeiro a ser executado.

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

Na **linha 52**, 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 55**, 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 58**, 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 59 e 61**, 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.

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

```
63
64    @Test
65    @Order(2)
66    @DisplayName("Não deve permitir duplicação do Usuário")
67    public void naoDeveDuplicarUsuario() {
68
69        usuarioService.cadastrarUsuario(new Usuario(0L,
70            "Maria da Silva", "https://i.imgur.com/NtyGneo.jpg", "maria_silva@email.com.br", "13465278"));
71
72        HttpEntity<Usuario> requisicao = new HttpEntity<Usuario>(new Usuario(0L,
73            "Maria da Silva", "https://i.imgur.com/NtyGneo.jpg", "maria_silva@email.com.br", "13465278"));
74
75        ResponseEntity<Usuario> resposta = testRestTemplate
76            .exchange("/usuarios/cadastrar", HttpMethod.POST, requisicao, Usuario.class);
77
78        assertEquals(HttpStatus.BAD_REQUEST, resposta.getStatusCode());
79    }
80
```

Na **linha 69**, através do método **cadastrarUsuario()** da **Classe UsuarioService**, foi persistido um Objeto da Classe Usuario no Banco de dados (Maria da Silva).

Na **linha 72**, 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 75**, 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 78**, 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

```
80
81    @Test
82    @Order(3)
83    @DisplayName("Alterar um Usuário")
84    public void deveAtualizarUmUsuario() {
85
86        Optional<Usuario> usuarioCreate = usuarioService.cadastrarUsuario(new Usuario(0L,
87            "Juliana Andrews", "https://i.imgur.com/yDRVeK7.jpg", "juliana_andrews@email.com.br", "juliana123"));
88
89        Usuario usuarioUpdate = new Usuario(usuarioCreate.get().getId(),
90            "Juliana Andrews Ramos", "https://i.imgur.com/T12NIp9.jpg", "juliana_ramos@email.com.br", "juliana123");
91
92        HttpEntity<Usuario> requisicao = new HttpEntity<Usuario>(usuarioUpdate);
93
94        ResponseEntity<Usuario> resposta = testRestTemplate
95            .withBasicAuth("root", "root")
96            .exchange("/usuarios/atualizar", HttpMethod.PUT, requisicao, Usuario.class);
97
98        assertEquals(HttpStatus.OK, resposta.getStatusCode());
99        assertEquals(usuarioUpdate.getNome(), resposta.getBody().getNome());
100       assertEquals(usuarioUpdate.getFoto(), resposta.getBody().getFoto());
101       assertEquals(usuarioUpdate.getUsuario(), resposta.getBody().getUsuario());
102   }
```

Na **linha 86**, 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 89**, foi criado um Objeto do tipo Usuario, chamado **usuarioUpdate**, que será utilizado para atualizar os dados persistidos no Objeto **usuarioCreate** (linha 78).

Na **linha 92**, 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 94**, 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 **linha95**, 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 98**, 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 99 a 101**, 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!.***

[Documentação: *Classe TestRestTemplate - Método .withBasicAuth()*]

Método 04 - Listar todos os Usuários

```
103
104     @Test
105     @Order(4)
106     @DisplayName("Listar todos os Usuários")
107     public void deveMostrarTodosUsuarios() {
108
109         usuarioService.cadastrarUsuario(new Usuario(0L,
110             "Sabrina Sanches", "https://i.imgur.com/EcJG8kB.jpg", "sabrina_sanches@email.com.br", "sabrina123"));
111
112         usuarioService.cadastrarUsuario(new Usuario(0L,
113             "Ricardo Marques", "https://i.imgur.com/Sk5SjWE.jpg", "ricardo_marques@email.com.br", "ricardo123"));
114
115         ResponseEntity<String> resposta = testRestTemplate
116             .withBasicAuth("root", "root")
117             .exchange("/usuarios/all", HttpMethod.GET, null, String.class);
118
119         assertEquals(HttpStatus.OK, resposta.getStatusCode());
120     }
121
```

Na **linhas 109 e 112**, foram persistidos dois Objetos da Classe Usuario no Banco de dados, através do método cadastrarUsuario() da Classe UsuarioService.

Na **linha 115**, 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 116**, 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.

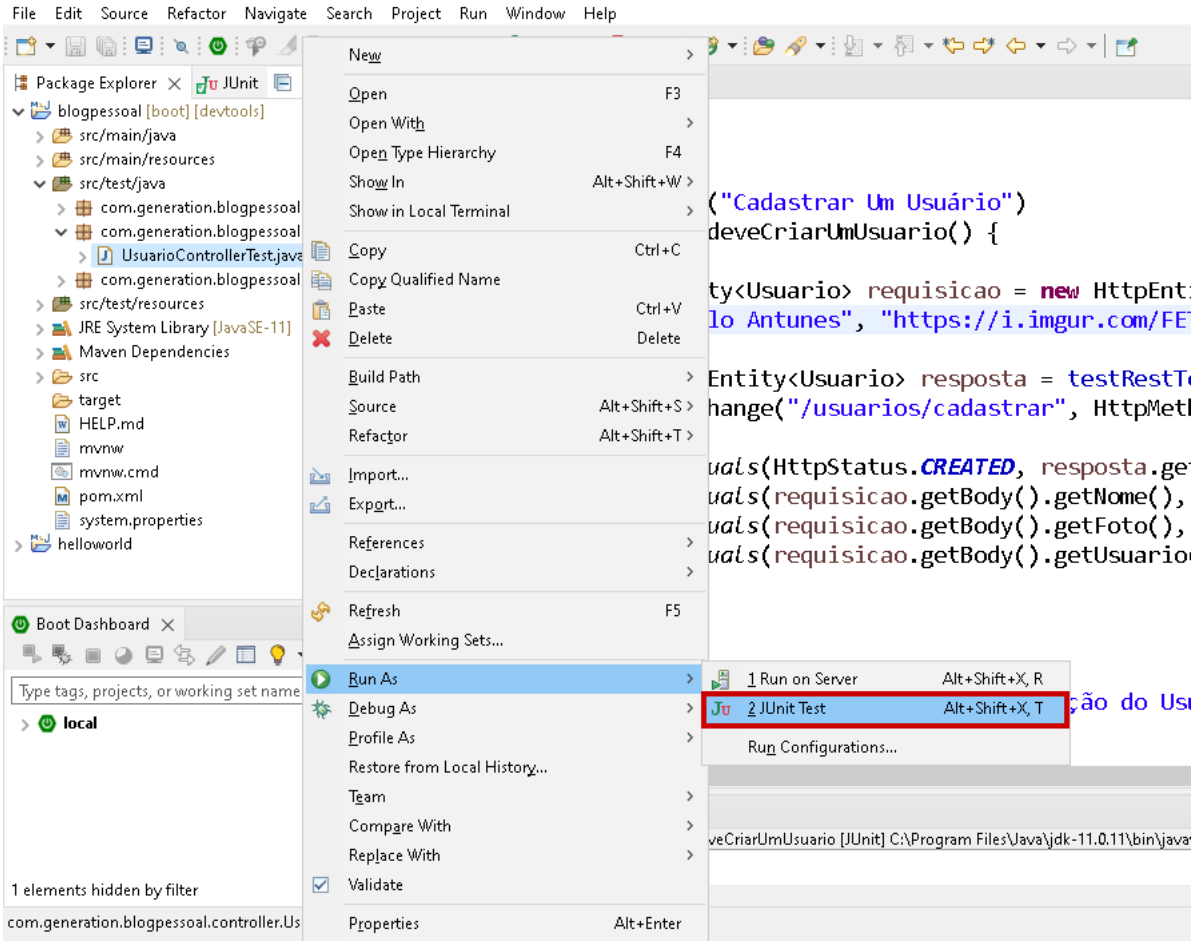
Na **linha 111**, através do método de asserção **AssertEquals()**, checaremos se a resposta da requisição (Response), é a resposta esperada (**OK → 200**). Para obter o status da resposta vamos utilizar o método **getStatusCode()** da **Classe ResponseEntity**.

[Código fonte: UsuarioControllerTest.java]

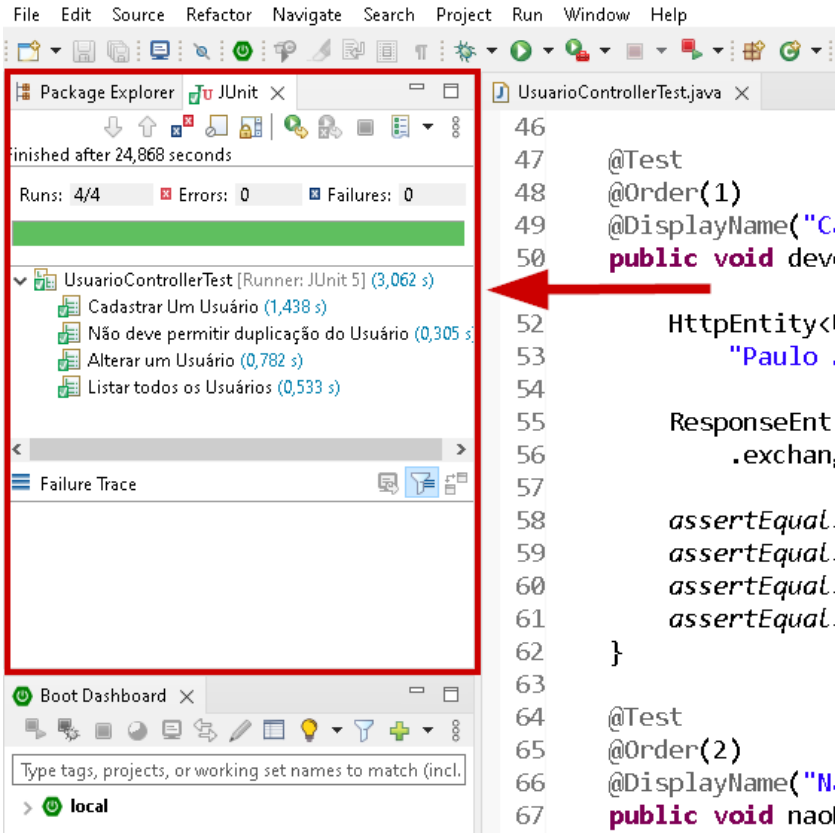
Passo 07 - Executando os Testes no STS

7.1. Executar todos os testes

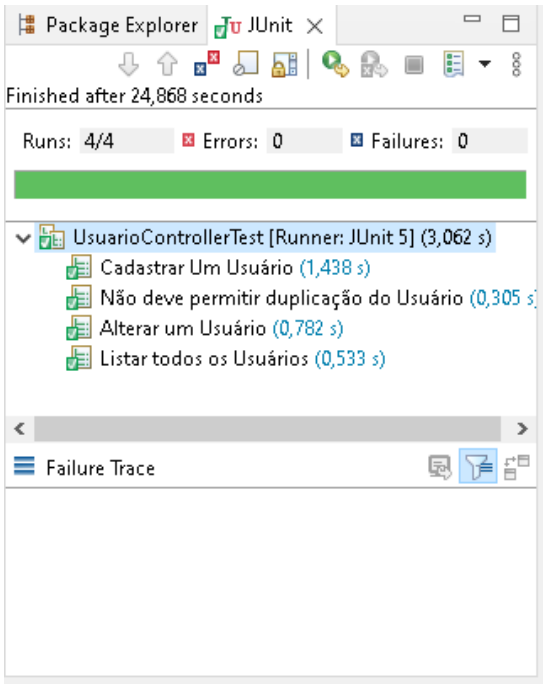
1. No lado esquerdo superior, na Guia **Project**, na Package **src/test/java**, clique com o botão direito do mouse sobre a Classe de teste que você deseja executar e clique na opção **Run As → JUnit Test**.



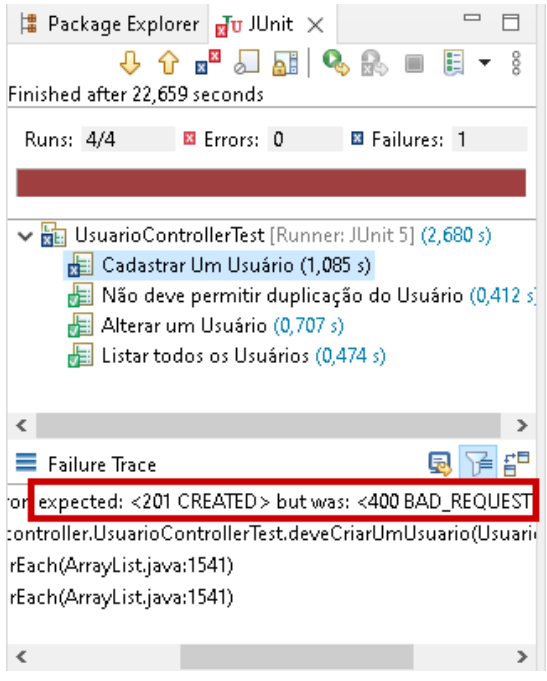
2. Para acompanhar os testes, ao lado da Guia **Project**, clique na Guia **JUnit** .



3. 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).



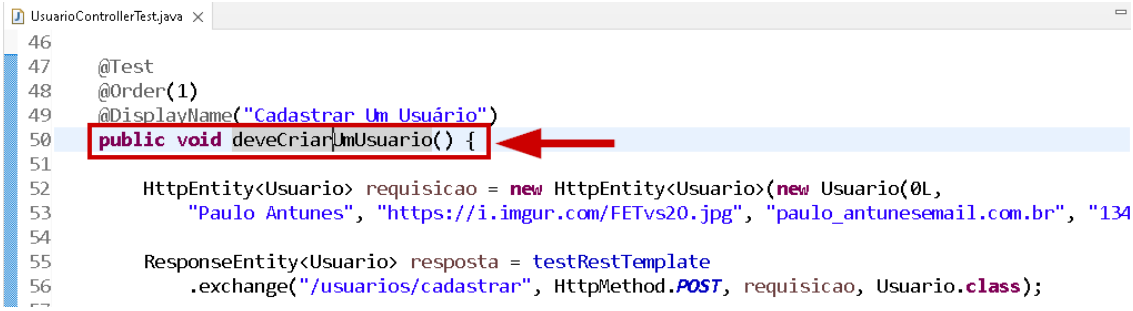
Janela 01: *Testes aprovados.*



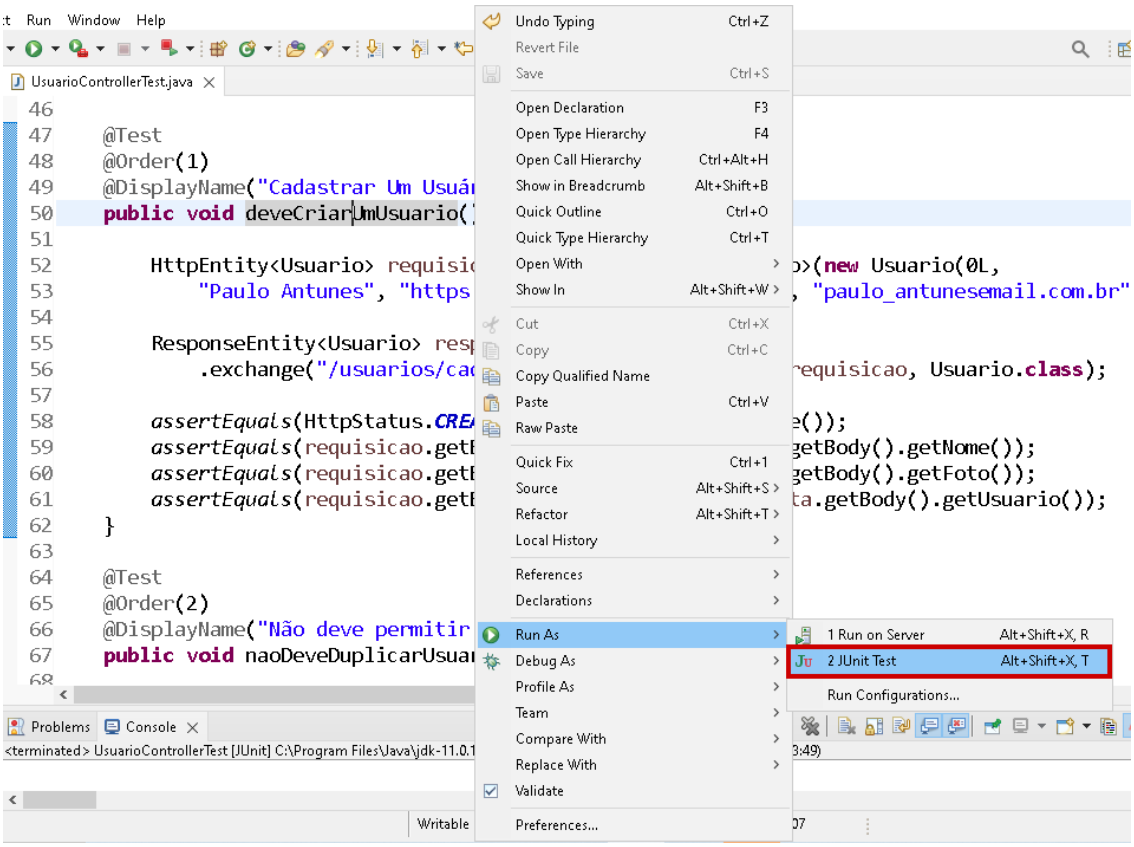
Janela 02: *Testes reprovados.*

7.2. Executar apenas um método espcífico

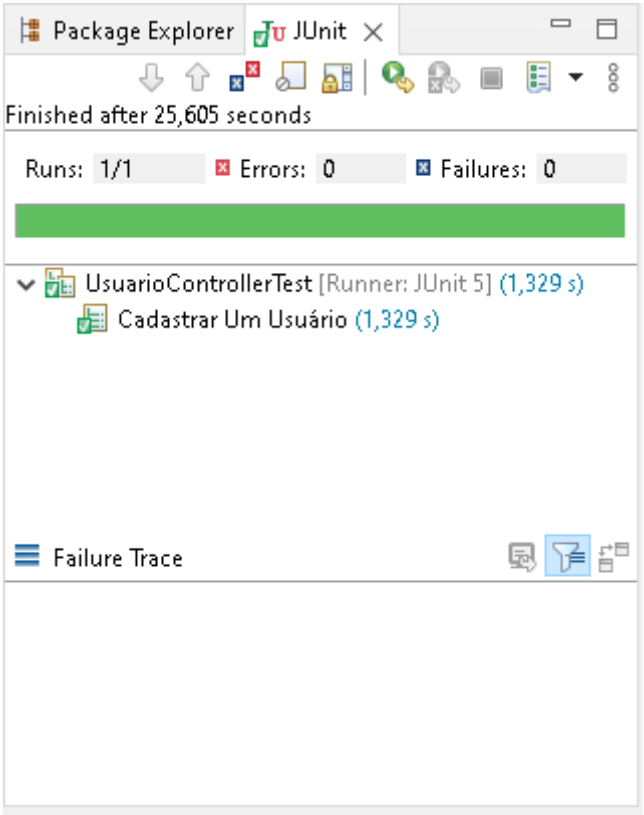
1. Posicione o cursor do mouse sobre o nome do teste. Observe que o nome será selecionado, como mostra a figura abaixo:



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.



DESAFIO: *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 porquê 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.**