Spring Testing

O Spring Testing é parte integrante do Ecossistema Spring e oferece suporte a testes de unidade e testes de integração, utilizando o Framework **JUnit 5**. Ao criar um projeto com o Spring Boot, automaticamente as dependências de testes já são inseridas no projeto como veremos adiante.

O que é 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 que é teste de integração?

Teste de integração é a fase do teste de software em que os módulos são combinados e testados em grupo.

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.

O framework JUnit

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 annotations

JUnit 5	Descrição	JUnit 4
@SpringBootTest	A anotação @SpringBootTest cria e inicializa o nosso ambiente de testes. A opção **webEnvironment=WebEnvironment.RANDOM_PORT** garante que durante os testes o Spring não utilize a mesma porta da aplicação (em ambiente local nossa porta padrão é a 8080).	*@SpringBootTest*
@Test	A anotação @Test indica que o método deve ser executado como um teste.	*@Test*
@BeforeEach	A anotação @BeforeEach indica que o método deve ser executado antes de cada teste da classe, para criar algumas pré-condições necessárias para cada teste (criar variáveis, por exemplo).	*@Before*
@BeforeAll	A anotação @BeforeAll indica que o método deve ser executado uma única vez antes de todos os testes da classe, para criar algumas pré-condições necessárias para todos os testes (criar objetos, por exemplo).	*@BeforeClass*
@AfterEach	A anotação @AfterEach indica que o método deve ser executado depois de cada teste para redefinir algumas condições após rodar cada teste (redefinir variáveis, por exemplo).	*@After*
@AfterAll	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).	*@AfterClass*

@Disabled	A anotação @Disabled desabilita temporariamente a execução de um teste específico. Cada método que é anotado com @Disabled não será executado.	*@Ignore*
@DisplayName	Personaliza o nome do teste permitindo inserir um Emoji (tecla Windows + .) e texto.	*@DisplayName*
@Order(1)	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.	*@Order(1)*

JUnit Assertions

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

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

Quais testes faremos?

Criaremos testes nas 3 Camadas principais da entidade Usuário do Blog Pessoal:

- Model (Entity);
- Repository;
- Controller.

Para executarmos os testes, faremos algumas configurações no módulo de testes do Spring em: **src/test** e alguns ajustes no arquivo **pom.xml**.
Antes de prosseguir, assegure que o projeto não esteja em execução no Spring.

#01 Configurações iniciais

Dependências

No arquivo, **pom.xml**, vamos alterar as linhas:

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

Para:

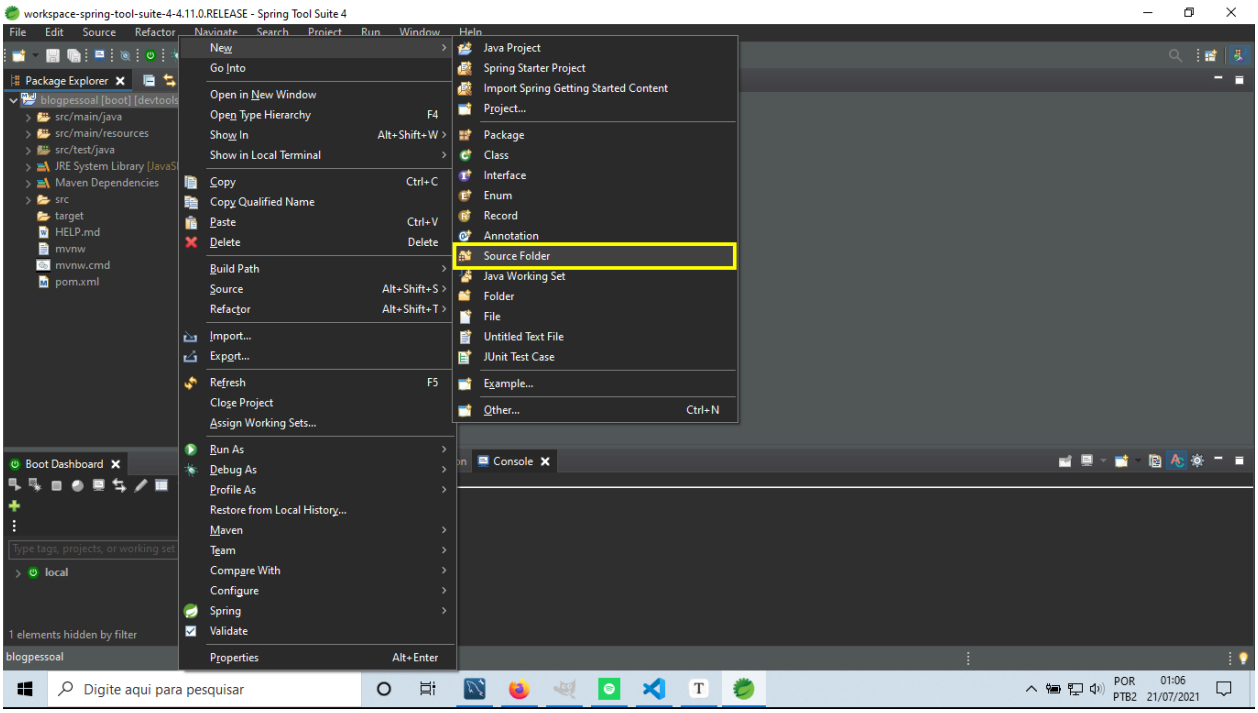
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>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

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

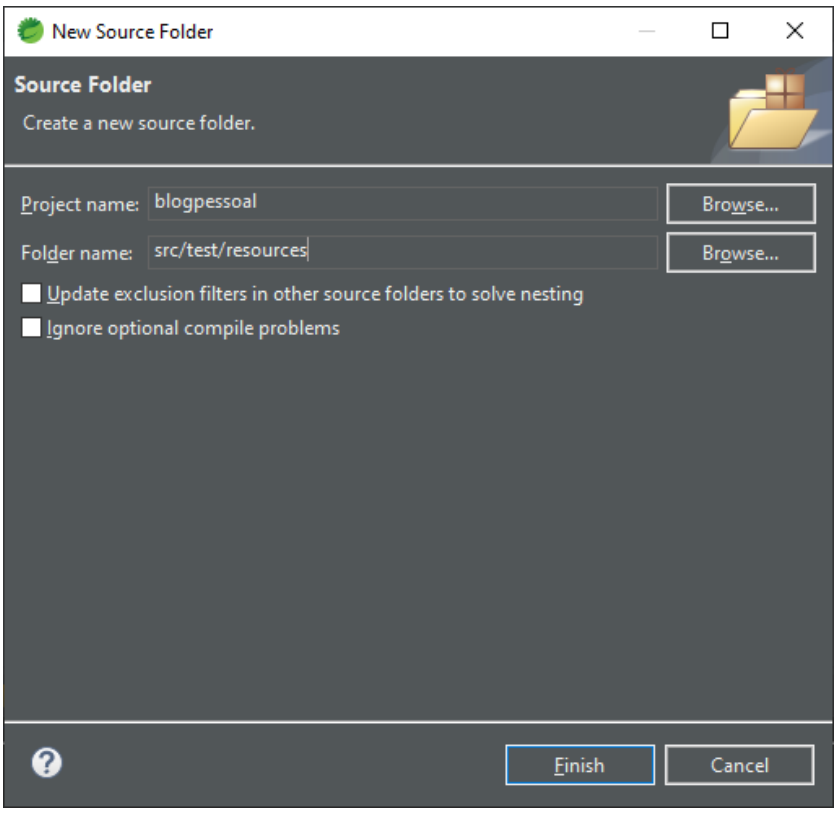
Banco de Dados para testes

Agora vamos configurar um Banco de dados de testes para não usar o Banco de dados principal.

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**

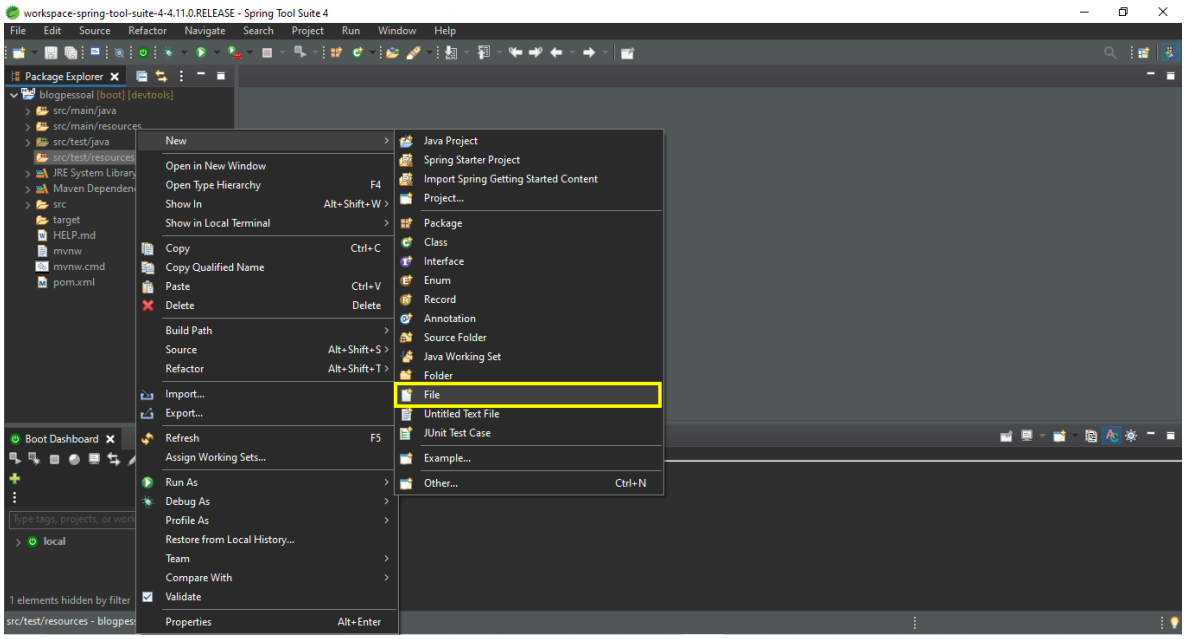


2. Em **Source Folder**, no item **Folder name**, informe o caminho como mostra a figura abaixo (**src/test/resources**), e clique em **Finish**:

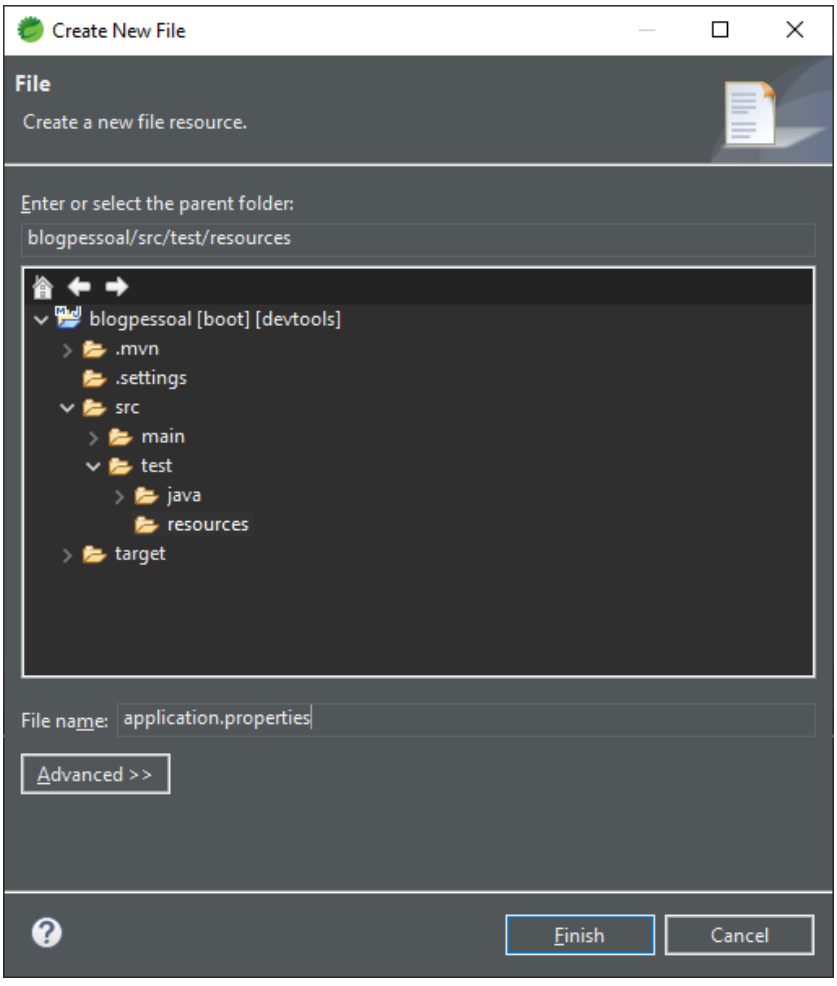


3. Na nova Source Folder (**src/test/resources**) , crie o arquivo **application.properties**, para configurarmos a conexão com o Banco de Dados de testes

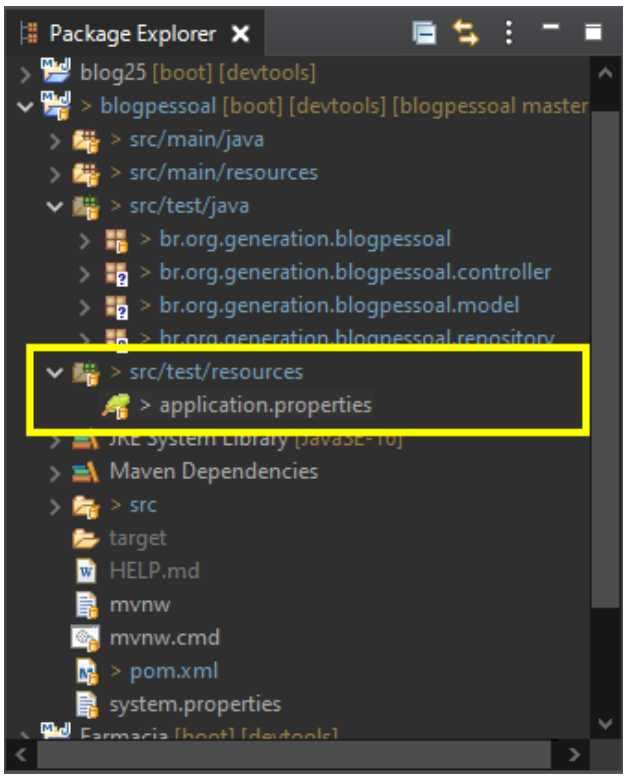
4. No lado esquerdo superior, na Guia **Package explorer**, na Package **src/test/resources**, clique com o botão direito do mouse e clique na opção **New->File**.



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



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



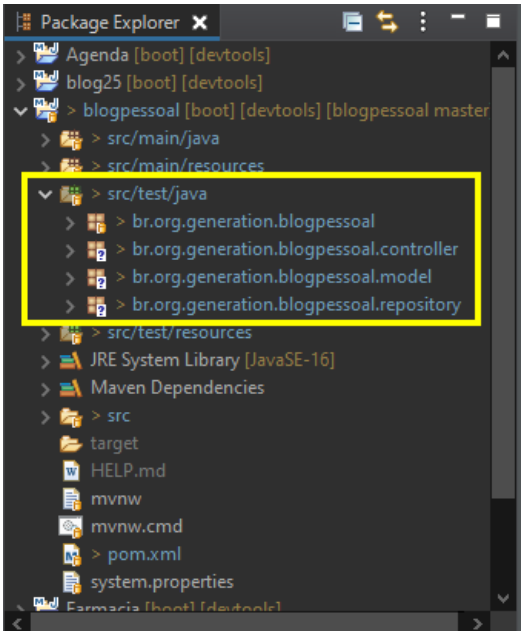
7. Insira no arquivo application.properties o código abaixo:

```
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.database=mysql
spring.datasource.url=jdbc:mysql://localhost/db_testeblogpessoal?
createDatabaseIfNotExist=true&serverTimezone=America/Sao_Paulo&useSSl=false
spring.datasource.username=root
spring.datasource.password=root

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect

spring.jackson.date-format=yyyy-MM-dd HH:mm:ss
spring.jackson.time-zone=Brazil/East
```

Observe que a linha: **spring.jpa.hibernate.ddl-auto=create-drop** foi alterada para ***create-drop***, ou seja, apagar e criar o banco de dados toda vez que executar o teste.
Observe que o nome do Banco de dados possui a palavra **teste** para indicar que será apenas para a execução dos testes.
Não esqueça de configurar a senha do usuário root.

#02 Criando os Testes no STS

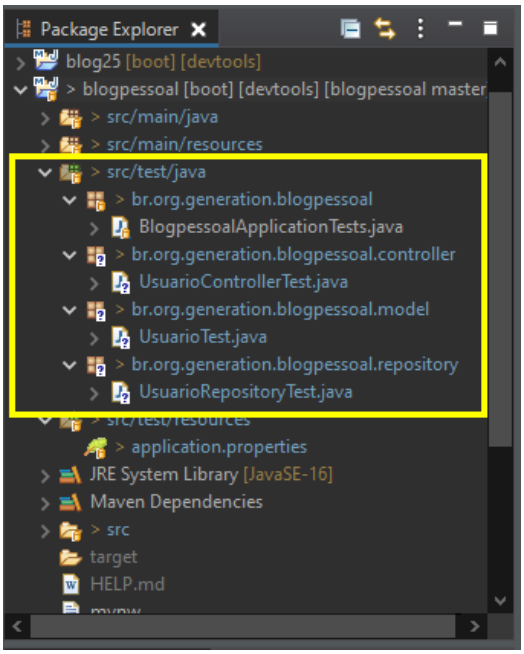
Na Source Folder de Testes (**src/test/java**) , observe que existe uma estrutura de pacotes identica a Source Folder Principal (**src/main/java**). Crie na Source Folder de Testes as packages Model, Repository e Controller como mostra a figura abaixo:



O Processo de criação dos arquivos é o mesmo do código principal. O nome das classes deverão ser iguais aos da Source Folder Principal (**src/main/java**) acrescentando a palavra Test no final do nome.

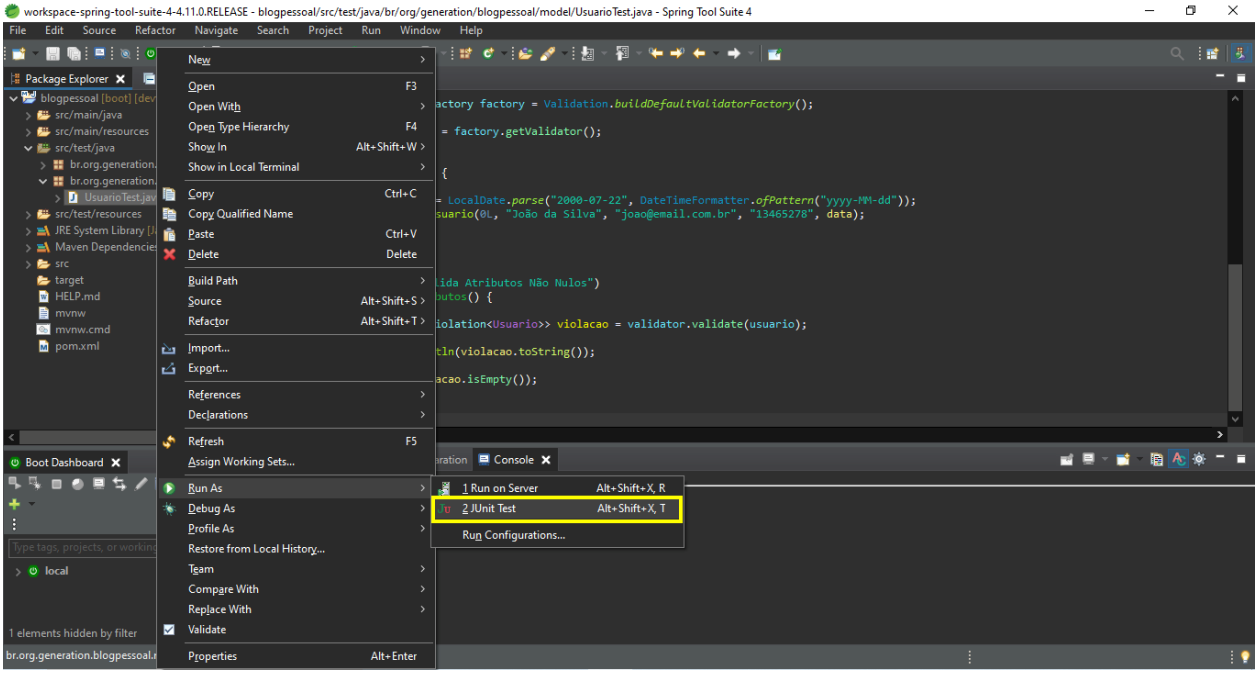
Exemplo: ContatoRepository -> ContatoRepositoryTest.

Quando você terminar de escrever todos os testes, a sua estrutura de pacotes ficará semelhante a figura abaixo:

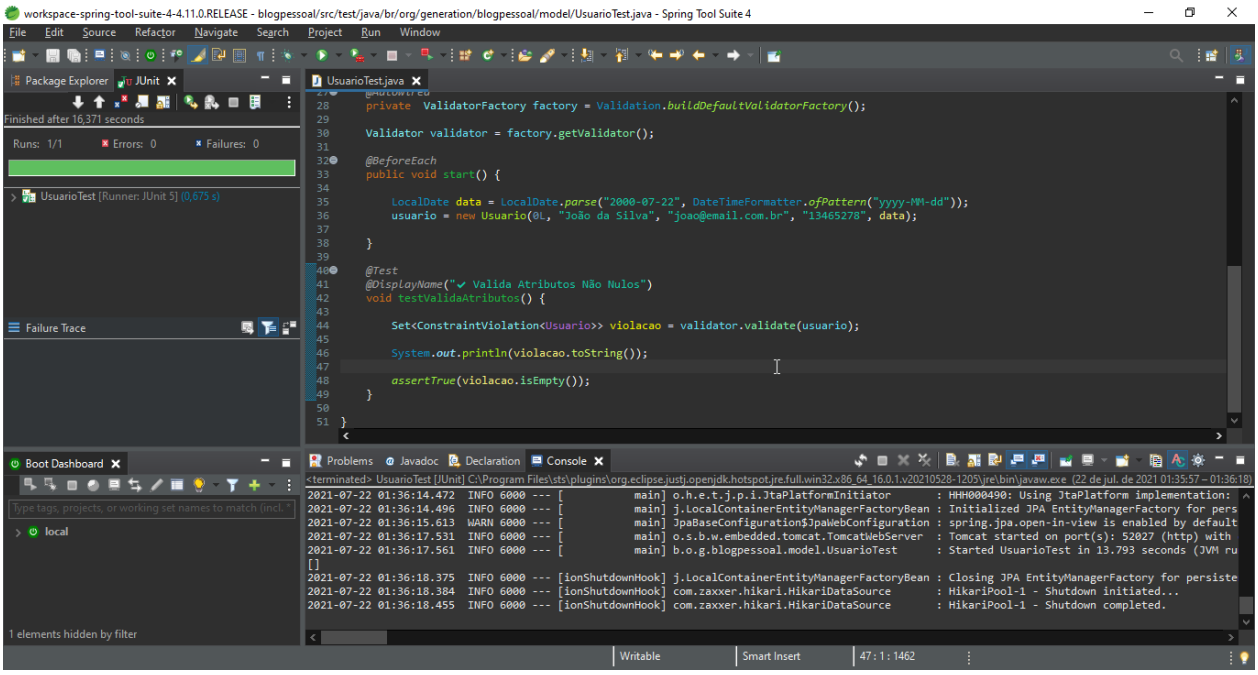


#03 Executando os Testes no STS

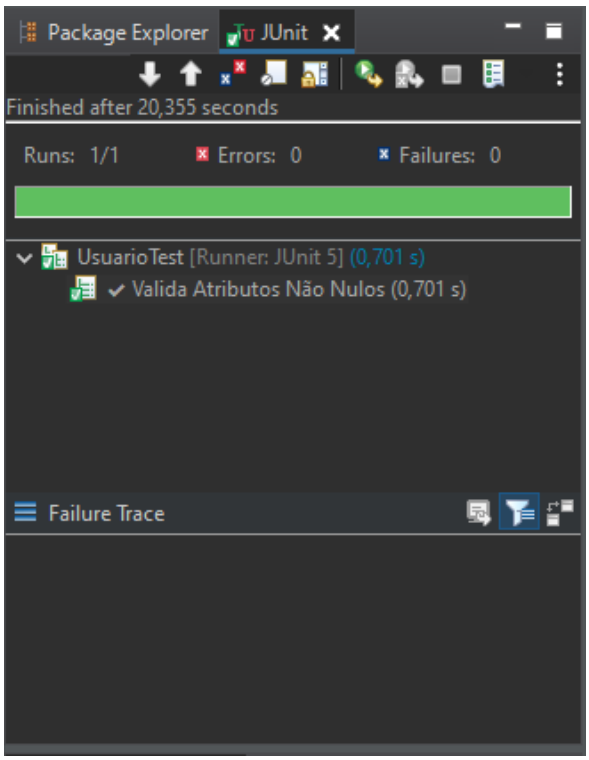
1. No lado esquerdo superior, na Guia **Project**, na Package **src/test/java**, clique com o botão direito do mouse sobre o teste que você deseja executar e clique na opção **Run As->JUnit Test**.



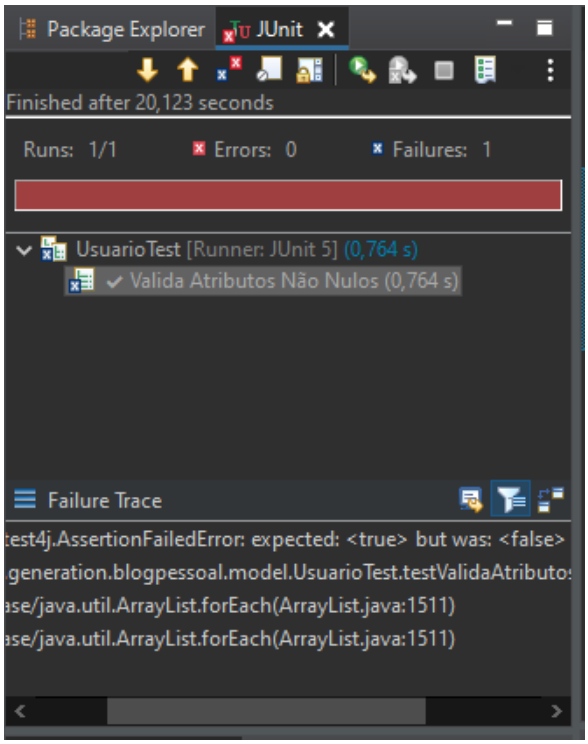
2. Para acompanhar os testes, ao lado da Guia **Project**, clique na Guia **JUnit**.



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.*

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***.

#04 Criando os Métodos Construtores na Classe Usuario (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. Através destes dois métodos iremos instanciar alguns objetos da Classe Usuario nas nossas classes de teste.

```
package br.org.generation.blogpessoal.model;

import java.time.LocalDate;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import com.fasterxml.jackson.annotation.JsonFormat;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@Entity
@Table(name = "tb_usuarios")
public class Usuario {

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

    @NotNull(message = "O atributo nome é obrigatório")
    @Size(min = 5, max = 100,
    message = "O atributo nome deve ter no mínimo 05 e no máximo 100 caracteres")
    private String nome;

    @NotNull(message = "O atributo usuário é obrigatório")
    @NotBlank(message = "O atributo usuário não pode ser vazio")
    @Email(message = "O atributo usuário deve ser um email")
    private String usuario;
```

```java
@NotNull(message = "O atributo senha é obrigatório")
@Size(min = 8, message = "O atributo senha deve ter no mínimo 8 caracteres")
private String senha;

@Column(name = "dt_nascimento")
@JsonFormat(pattern="yyyy-MM-dd")
private LocalDate dataNascimento;

@OneToMany (mappedBy = "usuario", cascade = CascadeType.REMOVE)
@JsonIgnoreProperties("usuario")
private List <Postagem> postagem;

// Primeiro método Construtor - Com os atributos

public Usuario(long id, String nome, String usuario, String senha,
LocalDate dataNascimento) {
    this.id = id;
    this.nome = nome;
    this.usuario = usuario;
    this.senha = senha;
    this.dataNascimento = dataNascimento;
}

// Segundo método Construtor - Sem os atributos
public Usuario() { }

public long getId() {
    return this.id;
}

public void setId(long id) {
    this.id = id;
}

public String getNome() {
    return this.nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public String getUsuario() {
    return this.usuario;
}

public void setUsuario(String usuario) {
    this.usuario = usuario;
}
```

```
    public String getSenha() {
        return this.senha;
    }

    public void setSenha(String senha) {
        this.senha = senha;
    }

    public LocalDate getDataNascimento() {
        return this.dataNascimento;
    }

    public void setDataNascimento(LocalDate dataNascimento) {
        this.dataNascimento = dataNascimento;
    }

    public List<Postagem> getPostagem() {
        return this.postagem;
    }

    public void setPostagem(List<Postagem> postagem) {
        this.postagem = postagem;
    }

}
```

#05 Testes na Camada Model (Entity)

A Classe UsuarioTest será utilizada para testar a Classe Usuario na Camada Model. Crie a classe UsuarioTest na package **model**, dentro da Source Folder de Testes (**src/test/java**)

Importante: O Teste da Classe Usuario, na camada Model, não utiliza o Banco de Dados.

UsuarioTest

```
package br.org.generation.blogpessoal.model;

import static org.junit.jupiter.api.Assertions.assertTrue;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
```

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class UsuarioTest {

    public Usuario usuario;
    public Usuario usuarioNulo = new Usuario();

    @Autowired
    private  ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

    Validator validator = factory.getValidator();

    @BeforeEach
    public void start() {

        LocalDate data = LocalDate.parse("2000-07-22",
        DateTimeFormatter.ofPattern("yyyy-MM-dd"));
        usuario = new Usuario(0L, "João da Silva",
        "joao@email.com.br", "13465278", data);
    }

    @Test
    @DisplayName("✓ Valida Atributos Não Nulos")
    void testValidaAtributos() {

        Set<ConstraintViolation<Usuario>> violacao = validator.validate(usuario);
        System.out.println(violacao.toString());

        assertTrue(violacao.isEmpty());
    }

    @Test
    @DisplayName("✕ Não Valida Atributos Nulos")
    void  testNaoValidaAtributos() {

        Set<ConstraintViolation<Usuario>> violacao = validator.validate(usuarioNul
        System.out.println(violacao.toString());

        assertTrue(violacao.isEmpty());
    }
}
```

🕸 Para inserir os emojis na annotation **@DisplayName**, utilize as teclas de atalho **Windows + Ponto**

Observe que o método **start()** foi anotado com a anotação**@BeforeEach** para inicializar o objeto da Classe Usuario antes de iniciar o teste. Observe que antes de instanciar o objeto Usuario, foi instanciado um objeto do tipo **LocalDate** que contém a data de nascimento do usuário.
Para testar a Model foi injetado (**@Autowired**), um objeto da Classe **Validation** para capturar todas as mensagens de erro de validação (**Constraints**).
Estas mensagens são armazenadas na **Collection do tipo Set** chamada **violations**. Através do método Assertion **assertTrue()** verificamos se a Collection violations está vazia (**violations.isEmpty()**).
Se estiver vazia, nenhum erro de validação foi encontrado (**testValidaAtributos()**), caso contrário as mensagens de erro serão exibidas no Console e o teste não passará (**testNaoValidaAtributos()**).

#06 Testes na Camada Repository

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

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 não utiliza a Classe de Serviço UsuarioService, ele utiliza o método save(), da Classe JpaRepository de forma direta.

UsuarioRepositoryTest

```
package br.org.generation.blogpessoal.repository;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.List;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
```

```java
import br.org.generation.blogpessoal.model.Usuario;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class UsuarioRepositoryTest {

    @Autowired
    private UsuarioRepository usuarioRepository;

    @BeforeAll
    void start(){

        LocalDate data = LocalDate.parse("2000-07-22",
        DateTimeFormatter.ofPattern("yyyy-MM-dd"));

        Usuario usuario = new Usuario(0, "João da Silva",
        "joao@email.com.br", "13465278", data);
        if(!usuarioRepository.findByUsuario(usuario.getUsuario()).isPresent())
            usuarioRepository.save(usuario);

        usuario = new Usuario(0, "Manuel da Silva",
        "manuel@email.com.br", "13465278", data);
        if(!usuarioRepository.findByUsuario(usuario.getUsuario()).isPresent())
            usuarioRepository.save(usuario);

        usuario = new Usuario(0, "Frederico da Silva",
        "frederico@email.com.br", "13465278", data);
        if(!usuarioRepository.findByUsuario(usuario.getUsuario()).isPresent())
            usuarioRepository.save(usuario);

        usuario = new Usuario(0, "Paulo Antunes",
        "paulo@email.com.br", "13465278", data);
        if(!usuarioRepository.findByUsuario(usuario.getUsuario()).isPresent())
            usuarioRepository.save(usuario);
    }

    @Test
    @DisplayName("🖹 Retorna o nome")
    public void findByNomeRetornaNome(){

        Usuario usuario = usuarioRepository.findByNome("João da Silva");
        assertTrue(usuario.getNome().equals("João da Silva"));
    }

    @Test
    @DisplayName("🖹 Retorna 3 usuarios")
    public void findAllByNomeContainingIgnoreCaseRetornaTresUsuarios() {

        List<Usuario> listaDeUsuarios = usuarioRepository
        .findAllByNomeContainingIgnoreCase("Silva");
```

```
        assertEquals(3, listaDeUsuarios.size());
    }

    @AfterAll
    public void end() {
        System.out.println("Teste Finalizado!");
    }
}
```

Annotations adicionais presentes no código

Annotation	Descrição
@TestInstance	A anotação @TestInstance permite modificar o ciclo de vida da classe de testes. A instância de um teste possui dois tipos de ciclo de vida: 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** não 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.

O método **start()**, anotado com a anotação **@BeforeAll**, inicializa 4 objetos do tipo Usuario e executa em todos os objetos o método **findByUsuario()** para verificar se já existe o usuário antes de criar.

No método **findByNomeRetornaNome()**, verifica se existe algum usuário cujo nome seja “João da Silva”, através da assertion **AssertTrue()**. Se existir, passa no teste.

No método **findAllByNomeContainingIgnoreCaseRetornaTresUsuarios()**, verifica se o numero de usuários que contenham no nome a String “Silva” é igual a 3, através da assertion **AssertEquals()**. O método **size(),** que pertence a **Collection List**, retorna o tamanho da List. Se o tamanho da List for igual 3, o teste será aprovado.

#07 Testes na Camada Controller

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

UsuarioControllerTest

```
package br.org.generation.blogpessoal.controller;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.MethodOrderer;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;
import org.junit.jupiter.api.TestMethodOrder;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;

import br.org.generation.blogpessoal.model.Usuario;
import br.org.generation.blogpessoal.repository.UsuarioRepository;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class UsuarioControllerTest {

    @Autowired
    private TestRestTemplate testRestTemplate;

    private Usuario usuario;
    private Usuario usuarioUpdate;
    private Usuario usuarioAdmin;

    @Autowired
    private UsuarioRepository usuarioRepository;
```

```java
@BeforeAll
public void start(){

    LocalDate dataAdmin = LocalDate.parse("1990-07-22",
    DateTimeFormatter.ofPattern("yyyy-MM-dd"));
    usuarioAdmin = new Usuario(0L, "Administrador",
    "admin@email.com.br", "admin123", dataAdmin);

    if(!usuarioRepository.findByUsuario(usuarioAdmin.getUsuario()).isPresent()

        HttpEntity<Usuario> request = new HttpEntity<Usuario>(usuarioAdmin);
        testRestTemplate
        .exchange("/usuarios/cadastrar", HttpMethod.POST, request, Usuario.cla

    }

    LocalDate dataPost = LocalDate.parse("2000-07-22",
    DateTimeFormatter.ofPattern("yyyy-MM-dd"));
    usuario = new Usuario(0L, "Paulo Antunes",
    "paulo@email.com.br", "13465278", dataPost);

    LocalDate dataPut = LocalDate.parse("2000-07-22",
    DateTimeFormatter.ofPattern("yyyy-MM-dd"));
    usuarioUpdate = new Usuario(2L, "Paulo Antunes de Souza",
     "paulo_souza@email.com.br", "souza123", dataPut);
}

@Test
@Order(1)
@DisplayName("✓ Cadastrar Usuário!")
public void deveRealizarPostUsuario() {

    HttpEntity<Usuario> request = new HttpEntity<Usuario>(usuario);

    ResponseEntity<Usuario> resposta = testRestTemplate
    .exchange("/usuarios/cadastrar", HttpMethod.POST, request, Usuario.class);

    assertEquals(HttpStatus.CREATED, resposta.getStatusCode());

}

@Test
@Order(2)
@DisplayName("👍 Listar todos os Usuários!")
public void deveMostrarTodosUsuarios() {

    ResponseEntity<String> resposta = testRestTemplate
    .withBasicAuth("admin@email.com.br", "admin123")
    .exchange("/usuarios/all", HttpMethod.GET, null, String.class);
```

```
        assertEquals(HttpStatus.OK, resposta.getStatusCode());
    }

    @Test
    @Order(3)
    @DisplayName("😳 Alterar Usuário!")
    public void deveRealizarPutUsuario() {

        HttpEntity<Usuario> request = new HttpEntity<Usuario>(usuarioUpdate);

        ResponseEntity<Usuario> resposta = testRestTemplate
        .withBasicAuth("admin@email.com.br", "admin123")
        .exchange("/usuarios/alterar", HttpMethod.PUT, request, Usuario.class);

        assertEquals(HttpStatus.OK, resposta.getStatusCode());

    }

}
```

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

Observe que o método start(), anotado com a anotação **@BeforeAll**, inicializa três objetos do tipo Usuário:

1. **usuario**: Não foi passado o Id, porquê este objeto será utilizado para testar o método Post.
2. **usuarioUpdate**: Foi passado o Id, porquê o objeto será utilizado para testar o método Put.
3. **usuarioAdmin**: Como o nosso Blog Pessoal está com a Camada de Segurança Basic implementada, precisaremos de um usuário para efetuar login na API. Utilizaremos este objeto para criar o usuário administrador antes de executar os testes.

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

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.

Vamos analisar a requisição do método Post:

```
    @Test
    public void deveRealizarPostUsuario() {
1)      HttpEntity<Usuario> request = new HttpEntity<Usuario>(usuario);

2)      ResponseEntity<Usuario> resposta = testRestTemplate
        .exchange("/usuarios/cadastrar", HttpMethod.POST, request, Usuario.class);

3)      assertEquals(HttpStatus.CREATED, resposta.getStatusCode());

    }
```

1. Cria um objeto HttpEntity recebendo o objeto da Classe Usuario. Nesta etapa, o processo é equivalente ao que o Postman faz: Transforma os atributos que você passou via JSON num objeto da Classe Usuario.
2. Cria a Requisição HTTP passando 4 parâmetros:

- **A URI**: Endereço do endpoint (/usuarios/cadastrar);
- **O Método HTTP**: Neste exemplo o método POST;
- **O Objeto HttpEntity**: Neste exemplo o objeto é da Classe Usuario;
- **O Tipo de Resposta esperada**: Neste exemplo será do tipo Usuario (Usuario.class).

3. Através do método de asserção **AssertEquals()**, comparamos se a resposta da requisição (Response), é a resposta esperada (CREATED -> 201).

Vamos analisar a requisição do método GET:

```
    @Test
    public void deveMostrarTodosUsuarios() {
1)      ResponseEntity<String> resposta = testRestTemplate
        .withBasicAuth("admin@email.com.br", "admin123")
        .exchange("/usuarios/all", HttpMethod.GET, null, String.class);

2)      assertEquals(HttpStatus.OK, resposta.getStatusCode());
    }
```

Observe que no Método GET não é necessário criar o Objeto request da Classe **HttpEntity**, porquê o GET não envia um Objeto na sua Requisição. Lembre-se que ao criar uma request do tipo GET no Postman você não passa nenhum parâmetro além da URL do endpoint.

1. Cria a Requisição HTTP passando 4 parâmetros:

- **A URI**: Endereço do endpoint (/usuarios/all);
- **O Método HTTP**: Neste exemplo o método GET;
- **O Objeto da requisição**: Neste exemplo ele será nulo (null);
- **O Tipo de Resposta esperada**: Como o objeto da requisição é nulo, a resposta esperada será do tipo String (String.class).

2. Através do método de asserção **AssertEquals**, comparamos se a resposta da requisição (Response), é a resposta esperada (OK -> 200).

Vamos analisar a requisição do método Put:

```
    @Test
    public void deveRealizarPutUsuario() {
1)      HttpEntity<Usuario> request = new HttpEntity<Usuario>(usuarioUpdate);

2)      ResponseEntity<Usuario> resposta = testRestTemplate
        .withBasicAuth("admin@email.com.br", "admin123")
        .exchange("/usuarios/cadastrar", HttpMethod.PUT, request, Usuario.class);

3)      assertEquals(HttpStatus.OK, resposta.getStatusCode());

    }
```

1. Cria um objeto HttpEntity recebendo o objeto da Classe Usuario. Nesta etapa, o processo é equivalente ao que o Postman faz: Transforma os atributos que você passou via JSON num objeto da Classe Usuario.
2. Cria a Requisição HTTP passando 4 parâmetros:

- **A URI**: Endereço do endpoint (/usuarios/alterar);
- **O Método HTTP**: Neste exemplo o método PUT;
- **O Objeto HttpEntity**: Neste exemplo o objeto é da Classe Usuario;
- **O Tipo de Resposta esperada**: Neste exemplo será do tipo Usuario (Usuario.class).

3. Através do método de asserção **AssertEquals()**, comparamos se a resposta da requisição (Response), é a resposta esperada (OK -> 200).

A implementação do Método **DELETE** é semelhante ao Método GET.

Importante

O Blog Pessoal está com o **Spring Security** habilitado, com autenticação do tipo **Basic** na API, logo o Objeto **testRestTemplate** dos Métodos http GET e PUT, que estão protegidos, deverá passar um usuário e uma senha válida para realizar os testes através da opção **withBasicAuth(user, password)**.

Exemplo:

```
ResponseEntity<String> resposta = testRestTemplate
.withBasicAuth("admin@email.com.br", "admin123")
.exchange("/usuarios/all", HttpMethod.GET, null, String.class);
```

Faça algumas alterações nos dados dos objetos e/ou escreva outros testes para praticar.