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 pode ser usada quando você desejar desabilitar temporariamente a execução de um teste específico. Cada método que é anotado com @Disabled não será executado.	*@Ignore*

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?

A partir de um projeto base (Agenda), criaremos testes nas 3 classes principais:

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

Para executarmos os testes, faremos algumas configurações específicas no módulo de testes do Spring em: **src/test** e alguns pequenos ajustes no arquivo **pom.xml**, como veremos a seguir.

Antes de prosseguir, assegure que o projeto não esteja em execução no Spring.

#01 Configurações iniciais

Dependências

No arquivo, **pom.xml**, vamos alterar a linha:

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

Para:

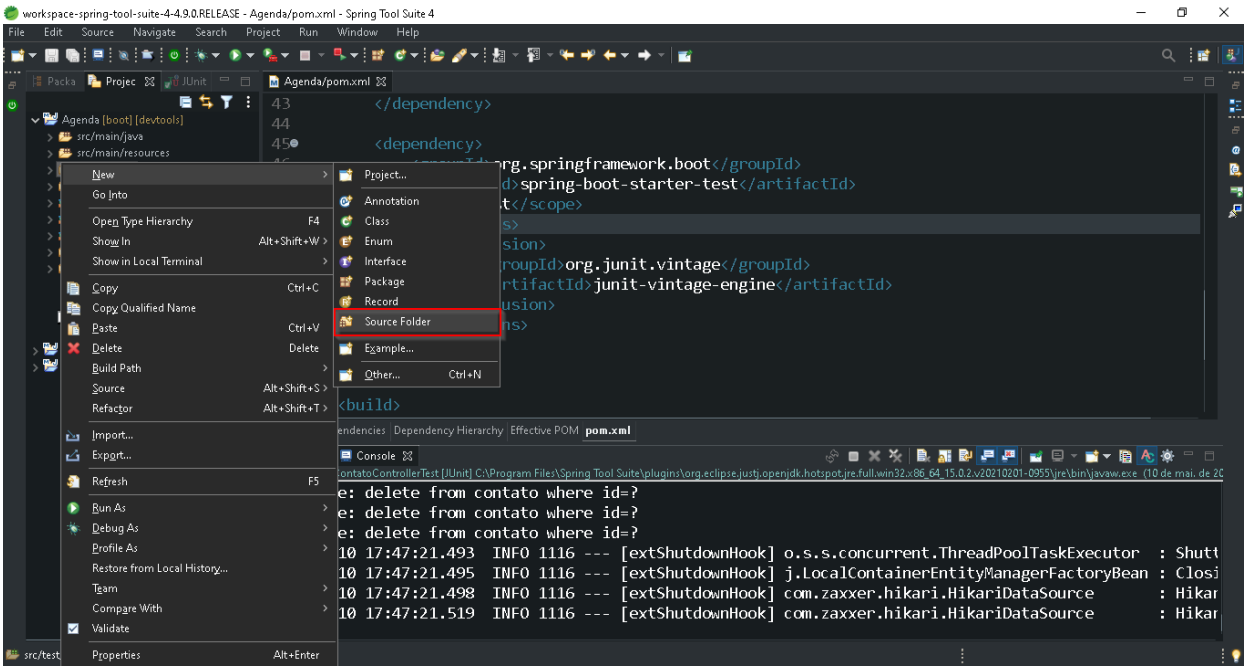
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>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

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

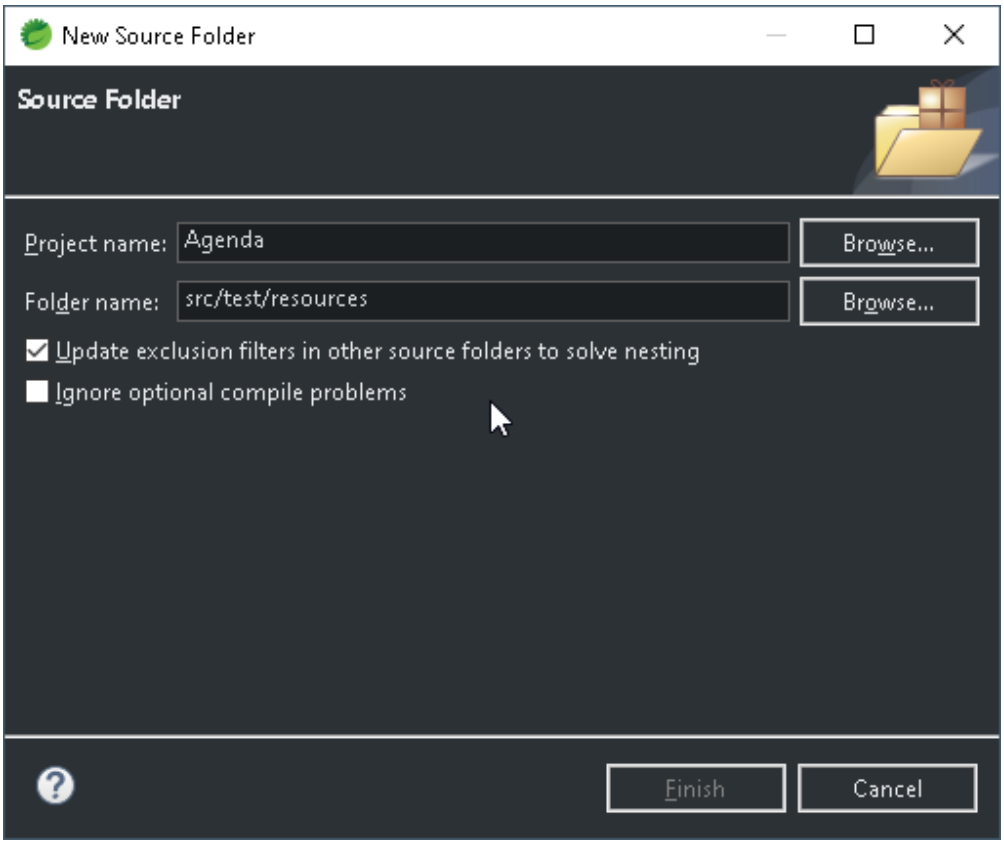
Banco de Dados para testes

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

1. No lado esquerdo superior, na Guia **Project**, na Package **src/test**, clique com o botão direito do mouse e clique na opção **New->Source folder**

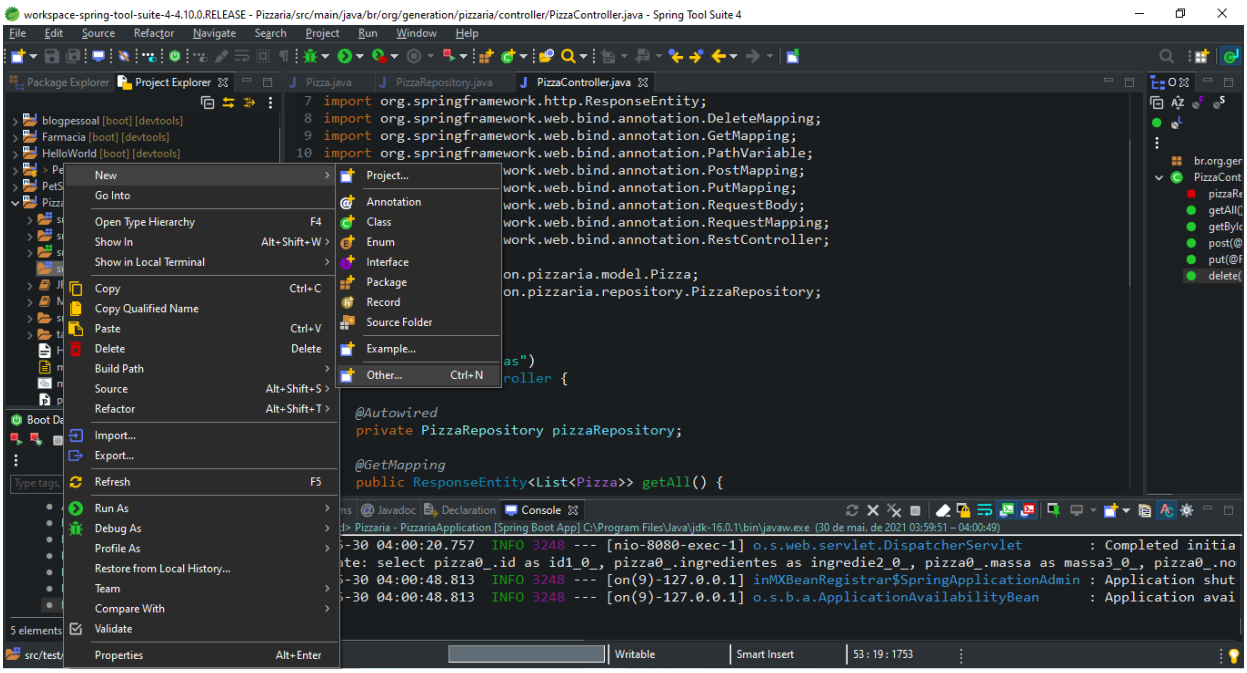


2. Em **Source Folder**, no item **Folder name**, informe o caminho como mostra a figura abaixo (**src/test/resources**), e clique em **Finish**:

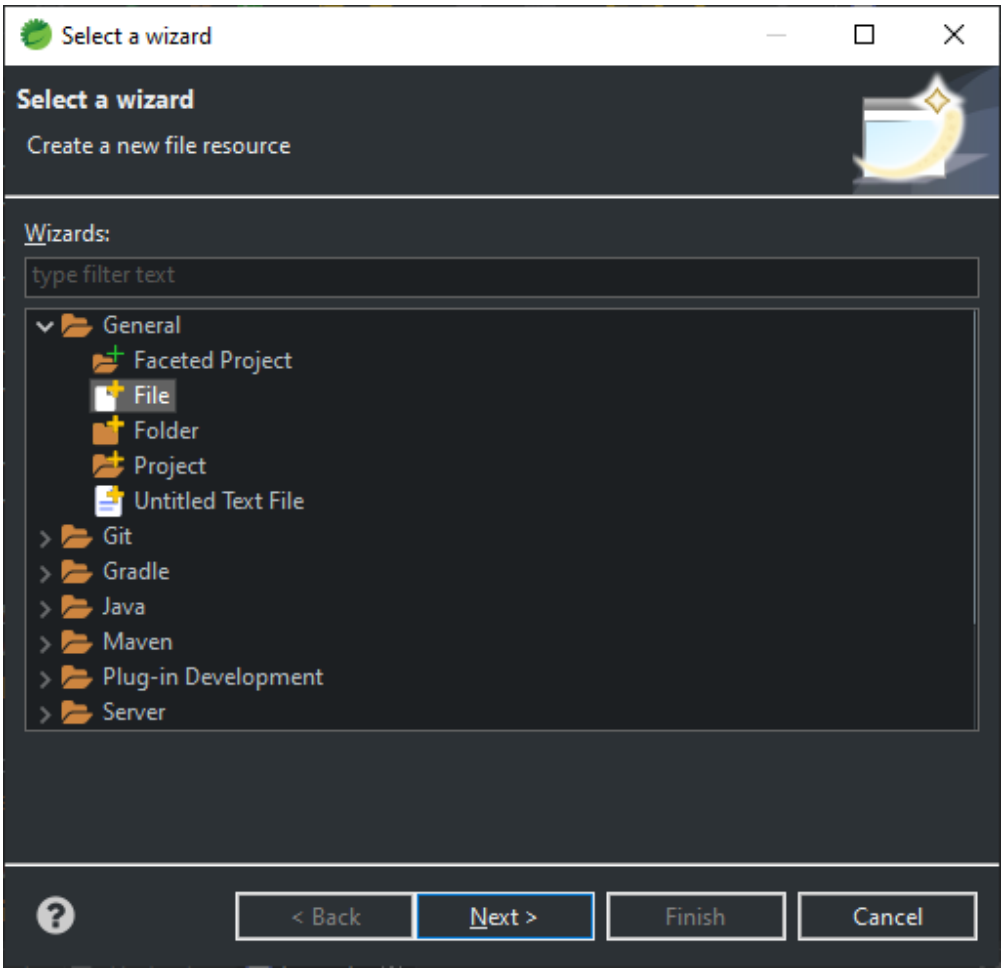


3. Na nova Source Folder (**src/test/resources**) , crie o arquivo **application.properties**, para configurarmos a conexão com o Banco de Dados de testes

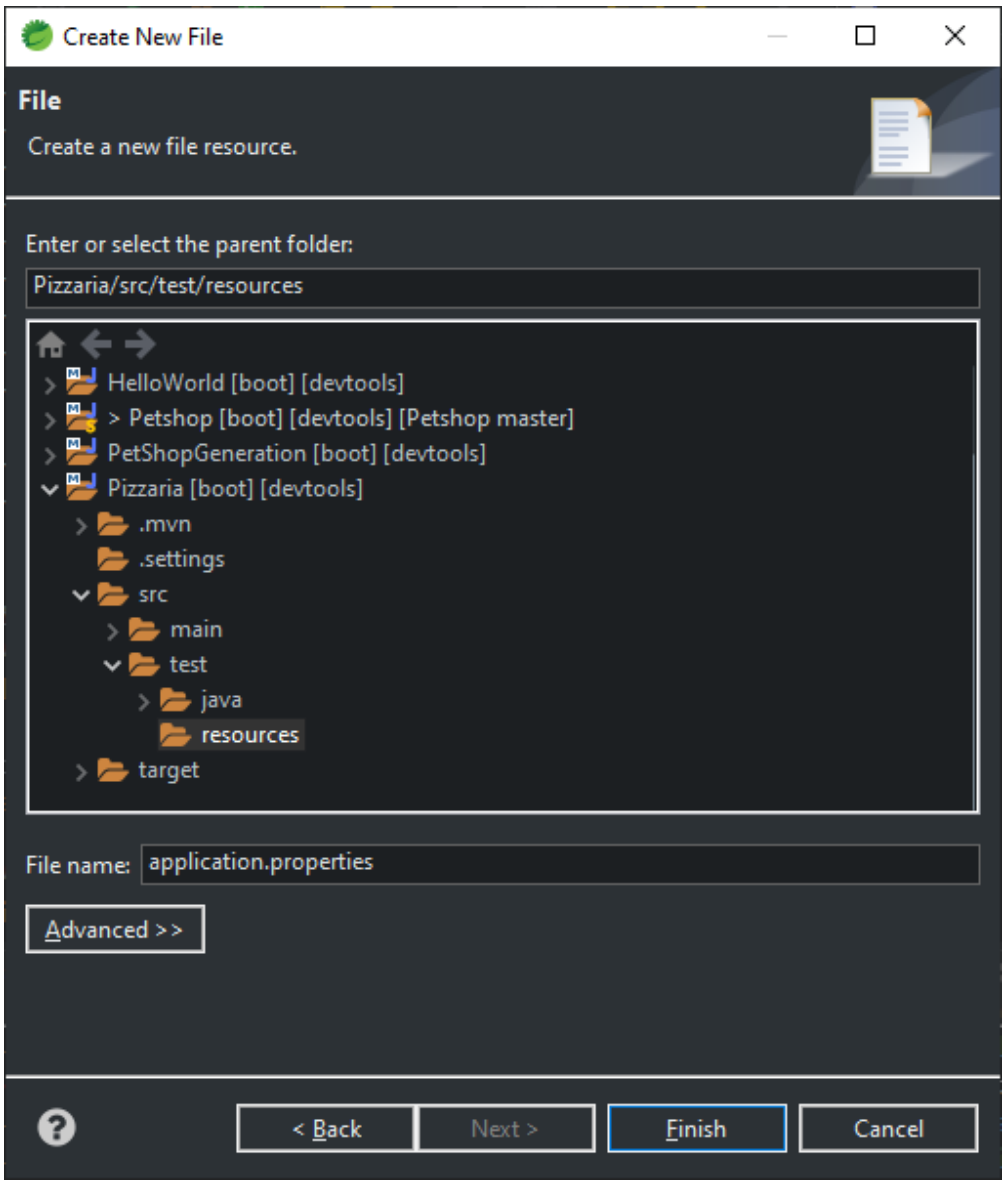
4. No lado esquerdo superior, na Guia **Project**, na Package **src/test/resources**, clique com o botão direito do mouse e clique na opção **New->Other**.



5. Na pasta **General**, selecione a opção **File** e clique em **Next**.



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



7. Insira neste arquivo as seguintes linhas:

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost/db_testagenda?
createDatabaseIfNotExist=true&serverTimezone=UTC&useSSl=false
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.show-sql=true
```

Observe que o nome do Banco de dados possui a palavra **test** para indicar que será apenas para a execução dos testes.

#02 Criando os Testes no STS

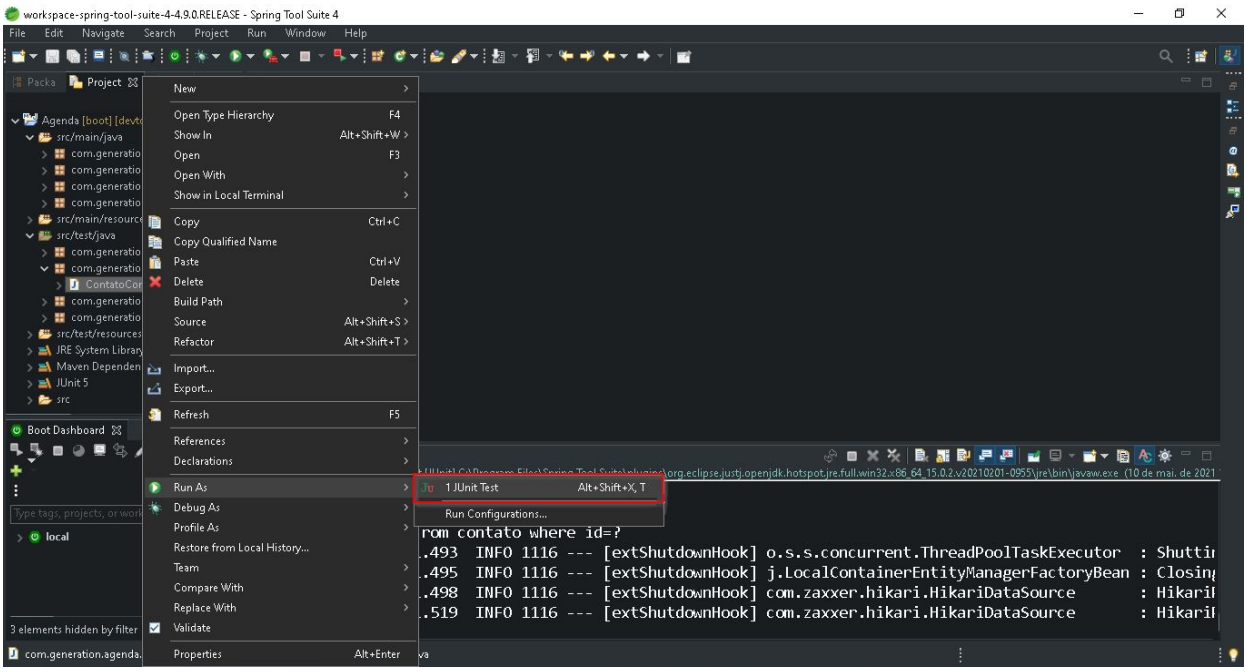
Na Source Folder de Testes (**src/test/java**) , observe que existe uma estrutura de pacotes identica a Source Folder Principal (**src/main/java**). Será a Source Folder de Testes que você criará os seus testes.

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

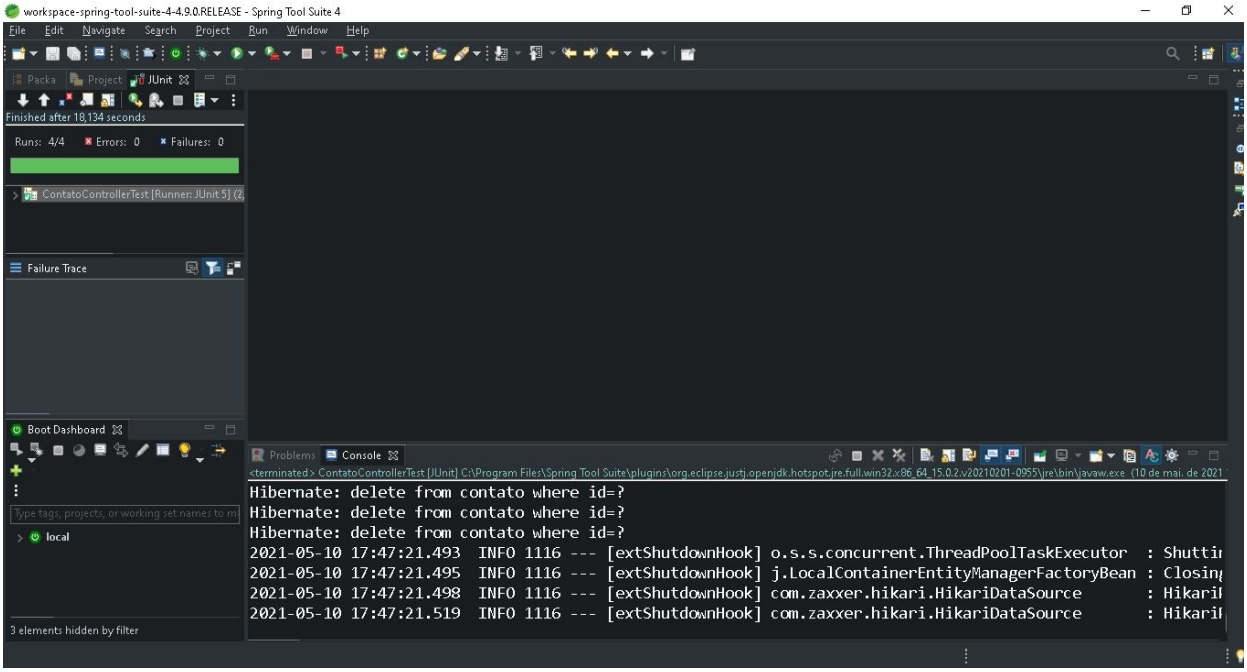
Exemplo:
ContatoRepository -> ContatoRepositoryTest.

#03 Executando os Testes no STS

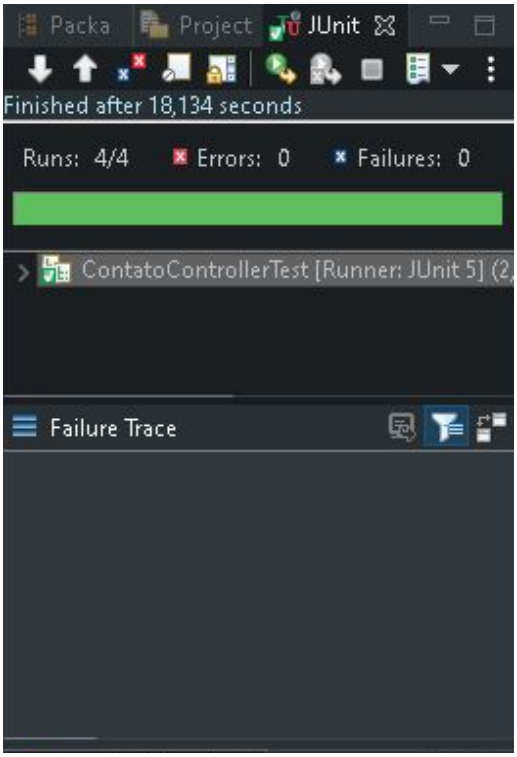
1. No lado esquerdo superior, na Guia **Project**, na Package **src/test/java**, clique com o botão direito do mouse sobre um dos testes e clique na opção **Run As->JUnit Test**.



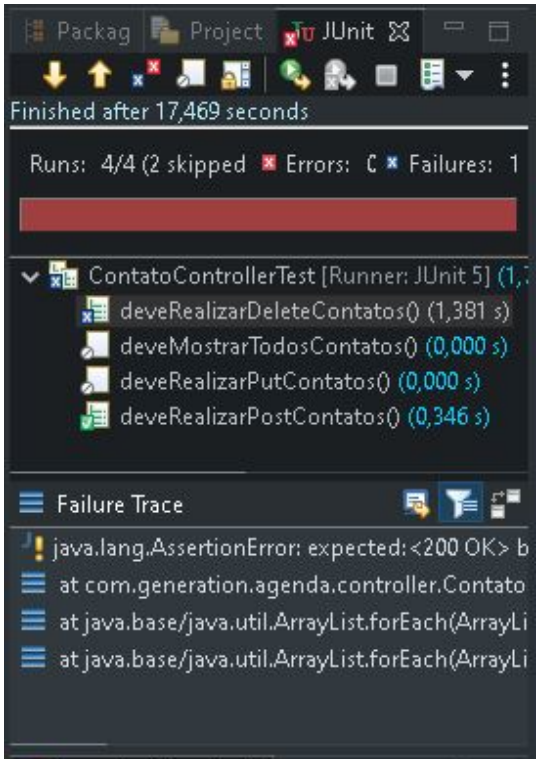
2) Para acompanhar os testes, ao lado da Guia **Project**, clique na Guia **JUnit**.



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 Testes na Camada Model (Entity)

Contato

```
@Entity
public class Contato {

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

    @NotEmpty(message="O DDD deve ser preenchido")
    private String ddd;

    @NotEmpty(message="O Telefone deve ser preenchido")
    private String telefone;

    @NotEmpty(message="O Nome deve ser preenchido")
    private String nome;

    public Contato() {}

    public Contato(Long id, String nome, String ddd, String telefone) {
        this.id = id;
        this.nome = nome;
        this.ddd = ddd;
        this.telefone = telefone;
    }

    // Métodos Getter's and Setter's

}
```

ContatoTest

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

    public Contato contato;
```

```
    @Autowired
    private final Validator validator = Validation
    .buildDefaultValidatorFactory()
    .getValidator();

    @BeforeEach
    public void start() {
        contato = new Contato(null, "Gabriel", "011y", "9xxxxxxx9");
    }

    @Test
    public void testValidationAtributos(){

        contato.setNome("João");
        contato.setDdd("011");
        contato.setTelefone("21837559");

        Set<ConstraintViolation<Contato>> violations = validator
        .validate(contato);
        System.out.println(violations.toString());
        assertTrue(violations.isEmpty());
    }
}
```

Oberve que o método **start()** foi anotado com a anotação**@BeforeEach** para inicializar o objeto da Classe Contato antes de iniciar o teste.
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, caso contrário as mensagens de erro serão exibidas no Console.

#05 Testes na Camada Repository

ContatoRepository

```
@Repository
public interface ContatoRepository extends JpaRepository<Contato, Long> {
```

```
    public Contato findByNome(String nome);
    public List<Contato> findAllByNomeIgnoreCaseContaining(String nome);

}
```

ContatoRepositoryTest

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

    @Autowired
    private ContatoRepository contatoRepository;

    @BeforeAll
    public void start() {
        Contato contato = new Contato(null, "Chefe", "0y", "9xxxxxxx9");
        if (contatoRepository.findByNome(contato.getNome()) == null)
            contatoRepository.save(contato);

        contato = new Contato(null, "Novo Chefe", "0y", "8xxxxxxx8");
        if (contatoRepository.findByNome(contato.getNome()) == null)
            contatoRepository.save(contato);

        contato = new Contato(null, "chefe Mais Antigo", "0y", "7xxxxxxx7");
        if (contatoRepository.findByNome(contato.getNome()) == null)
            contatoRepository.save(contato);

        contato = new Contato(null, "Amigo", "0z", "5xxxxxxx5");
        if (contatoRepository.findByNome(contato.getNome()) == null)
            contatoRepository.save(contato);
    }

    @Test
    public void findByNomeRetornaContato() throws Exception {

        Contato contato = contatoRepository.findByNome("Chefe");

        Assert.assertTrue(contato.getNome().equals("Chefe"));
    }

    @Test
    public void findAllByNomeIgnoreCaseRetornaTresContatos() {

        List<Contato> contatos = contatoRepository
```

```
        .findAllByNomeIgnoreCaseContaining("chefe");

        Assert.assertEquals(3, contatos.size());
    }

    @AfterAll
    public void end() {
        contatoRepository.deleteAll();
    }
}
```

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.
@DataJpaTest	Esta anotação desabilitará a configuração automática completa e, em vez disso, aplicará apenas a configuração relevante aos testes JPA, ou seja, concentra os testes no Spring Data JPA.

O método start(), anotado com a anotação **@BeforeAll**, inicializa 4 objetos do tipo contato e executa em todos o método método **findByNome()** para verificar se existe o contato antes de salvar no banco de dados.

No método **findByNomeRetornaContato()**, o teste verifica se existe algum contato cujo nome seja “chefe”, através da assertion **AssertTrue()**. Se existir, passa no teste.

No método **findAllByNomeIgnoreCaseRetornaTresContatos()**, o teste verifica se o numero de contatos que contenham no nome a palavra “chefe” é igual 3, através da assertion **AssertEquals()**. O método **size()** pertence a Collection List e retorna o tamanho do List.

#06 Testes na Camada Controller

ContatoController

```
@RestController
@RequestMapping("/contatos")
public class ContatoController {

    @Autowired
    private ContatoRepository contatoRepository;

    @GetMapping
    public ResponseEntity<List<Contato>> getAll() {
        List<Contato> contatos = contatoRepository.findAll();
        return ResponseEntity.ok(contatos);
    }

    @GetMapping("/contato/{id}")
    public ResponseEntity<Contato> getById(@PathVariable Long id) {
        return contatoRepository.findById(id)
                .map(resp -> ResponseEntity.ok(resp))
                .orElse(ResponseEntity.badRequest().build());
    }

    @PostMapping("/inserir")
    public ResponseEntity<Contato> post(@RequestBody Contato contato) {
        contato = contatoRepository.save(contato);
        return ResponseEntity.status(HttpStatus.CREATED).body(contato);
    }

    @PutMapping("/alterar")
    public ResponseEntity<Contato> put(@RequestBody Contato contato) {
        contato = contatoRepository.save(contato);
        return ResponseEntity.status(HttpStatus.OK).body(contato);
```

```
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        contatoRepository.deleteById(id);
    }
}
```

ContatoControllerTest

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

    @Autowired
    private TestRestTemplate testRestTemplate;

    private Contato contato;
    private Contato contatoupd;

    @BeforeAll
    public void start() {
        contato = new Contato(null, "Maria", "21", "44451198");
        contatoupd = new Contato(2L,"Maria da Silva", "23", "995467892");
    }

    @Test
    public void deveRealizarPostContatos() {
        HttpEntity<Contato> request = new HttpEntity<Contato>(contato);

        ResponseEntity<Contato> resposta = testRestTemplate
        .exchange("/contatos/inserir", HttpMethod.POST, request, Contato.clas
        Assert.assertEquals(HttpStatus.CREATED, resposta.getStatusCode());
    }

    @Disabled
    @Test
    public void deveMostrarTodosContatos() {
        ResponseEntity<String> resposta = testRestTemplate
        .exchange("/contatos/", HttpMethod.GET, null, String.class);
        Assert.assertEquals(HttpStatus.OK, resposta.getStatusCode());
    }
```

```
    @Disabled
    @Test
    public void deveRealizarPutContatos() {
        HttpEntity<Contato> request = new HttpEntity<Contato>(contatoupd);

        ResponseEntity<Contato> resposta = testRestTemplate
        .exchange("/contatos/alterar", HttpMethod.PUT, request, Contato.class
        Assert.assertEquals(HttpStatus.OK, resposta.getStatusCode());
    }

    @Disabled
    @Test
    public void deveRealizarDeleteContatos() {
        ResponseEntity<String> resposta = testRestTemplate
        .exchange("/contatos/3", HttpMethod.DELETE, null, String.class);
        Assert.assertEquals(HttpStatus.OK, resposta.getStatusCode());
    }
}
```

O teste da Camada Controller é um pouco diferente 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 dois objetos do tipo contato. A diferença é que no primeiro objeto não foi passado o Id, porquê este objeto será utilizado para testar o método Post. No segundo objeto o Id foi passado, porquê o objeto será utilizado para testar o método Put.

Para simular as Requisições e Respostas, utilizaremos alguns objetos:

Objetos	**Descrição**
TestRestTemplate()	É um cliente para escrever testes de integração 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 HttpStatus

TestRestTemplate.exchange(URI, HttpMethod, RequestType, ResponseType)	O método excahange executa uma solicitação 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 deveRealizarPostContatos() {

1)      HttpEntity<Contato> request = new HttpEntity<Contato>(contato);

2)      ResponseEntity<Contato> resposta = testRestTemplate
        .exchange("/contatos/inserir", HttpMethod.POST, request, Contato.clas
3)      Assert.assertEquals(HttpStatus.CREATED, resposta.getStatusCode());

}
```

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

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

3. Através do método assertion **AssertEquals()**, comparamos se a resposta da requisição (Response), é a resposta esperada (CREATED -> 201).
   O Método PUT é semelhante ao Método POST.

Vamos analisar a requisição do método GET:

```
    @Test
    public void deveMostrarTodosContatos() {
1)      ResponseEntity<String> resposta = testRestTemplate
```

```
        .exchange("/contatos/", HttpMethod.GET, null, String.class);
2)      Assert.assertEquals(HttpStatus.OK, resposta.getStatusCode());
    }
```

Antes de executar o Método GET, retire a anotação @Disabled.

Observe que no Método GET não é necessário criar o Objeto request da Classe **HttpEntity**, porquê o GET não envia um Objeto na sua Request. 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;
- **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 assertion **AssertEquals**, comparamos se a resposta da requisição (Response), é a resposta esperada (OK -> 200).
   O Método DELETE é semelhante ao Método GET.

Importante

Caso você tenha habilitado o **Spring Security** com autenticação do tipo **BasicAuth** na API, o Objeto **testRestTemplate** deverá passar o usuário e a senha para realizar os testes.

Exemplo:

```
ResponseEntity<String> resposta = testRestTemplate
.withBasicAuth("usuario", "senha")
.exchange("/v1/fornecedor", HttpMethod.GET, null, String.class);
```

Agora que você aprendeu a fazer testes no Spring, pratique escrevendo outros testes nas classes dos seus projetos.

O Código do projeto Agenda está disponível em:
https://github.com/conteudoGeneration/Spring-com-J-unit.git