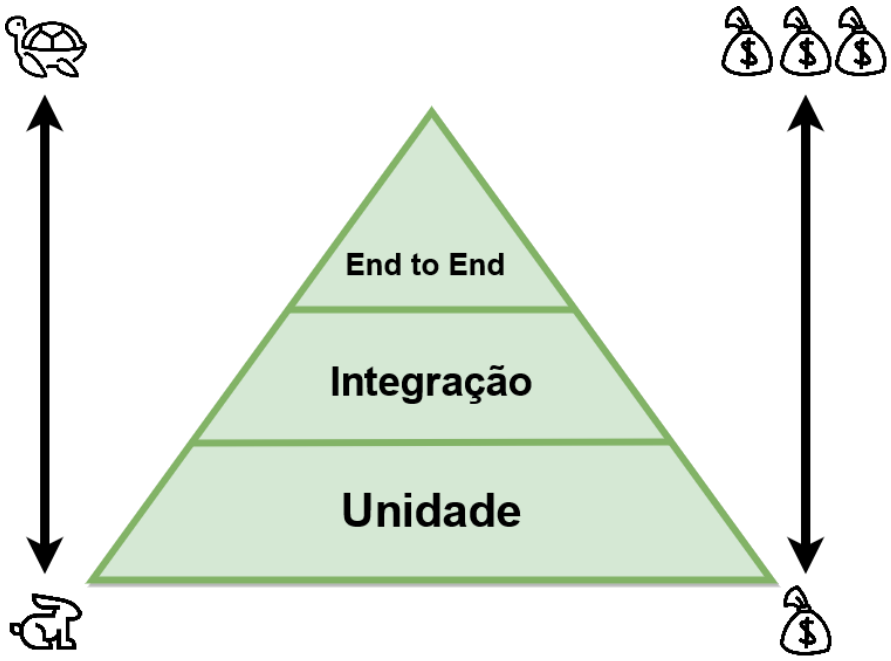
Teste de Software - JUnit 5 - Introdução

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 umas 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.



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, porquê além de ser muito rápido é o teste mais barato porquê 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 sí.

O Teste de Integração é um pouco mais lento e um pouco mais caro do que o Teste de Unidade porquê 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
@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 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.

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

Documentação: JUnit 5

3.1. Anotações do JUnit

JUnit 5	Descrição
@Test	A anotação @Test indica que o método deve ser executado como um teste.
@BeforeEach	A anotação @BeforeEach 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).
@BeforeAll	A anotação @BeforeAll 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).
@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).
@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).
@Disabled	A anotação @Disabled desabilita temporariamente a execução de um teste específico. Cada método que é anotado com @Disabled não será executado.
@DisplayName	Personaliza o nome do teste permitindo inserir um Emoji (tecla Windows + .) e texto.
@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.
@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** 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
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.

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 como vocês podem ver no exemplo nesse link .***

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 **Usuario** do Projeto 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.