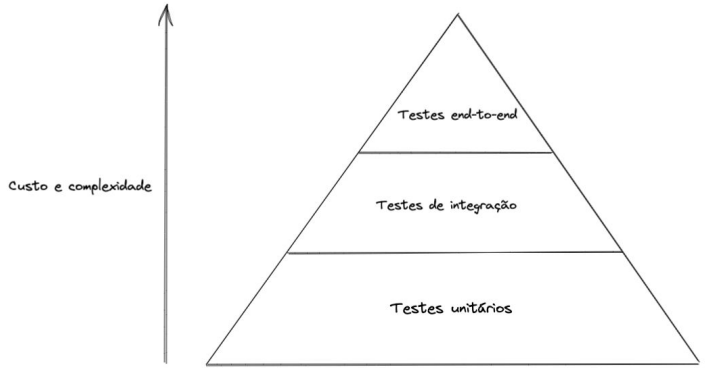
Introdução

Nesse módulo nós estudamos sobre testes automatizados.

Testes servem para garantir a qualidade das aplicações, prevenindo comportamentos inesperados e bugs em ambiente de produção.

Vamos relembrar o que estudamos?

Pirâmide de testes



Quando falamos de testes em aplicações, nos referimos essencialmente a códigos que testam o próprio código e comportamento da aplicação. Existem diferentes tipos de testes, mas os mais comuns são testes **unitários**, **integração** e **end-to-end** (**ponta-a-ponta**).

Esses tipos de testes se diferenciam em seus objetivos, complexidade e custo de execução. Testes mais simples como de unidade levam de milissegundos a alguns segundos para serem executados. Por outro lado, testes mais complexos e abrangentes como end-to-end podem levar horas.

Na ilustração anterior é mostrada a famosa **pirâmide de testes**, cujo objetivo é demonstrar a abrangência dos três tipos mais famosos. Na base da pirâmide estão os testes unitários, demonstrando que uma aplicação deve estar mais coberta com esse tipo de teste. No topo, está a mais custosa classe de testes: a end-to-end.

Esse artigo (em inglês) no blog de **Martin Fowler** detalha de forma prática a Pirâmide de Testes:
https://martinfowler.com/articles/practical-test-pyramid.html

Teste Unitários

Testes unitários se referem a testes que validam as menores partes de uma aplicação: os métodos de cada classe. De forma simples, todo método público de toda classe deve ser testado.

```
public class Product {
  private double price;

  private double maxDiscount;

  public double getPrice() {
     return price;
  }

  public void setPrice(double price) {
     this.price = price;
  }

  public void setMaxDiscount(double maxDiscount) {
     this.maxDiscount = maxDiscount;
  }

  public double getPriceWithDiscount(double discount) {
     if (discount > maxDiscount)
        return price - (price * maxDiscount);
     else
        return price - (price * discount);
  }
}
```

A classe anterior possui o método "**getPriceWithDiscount**" que implementa a regra de negócio para calcular o preço com base em um desconto concedido. O desconto concedido não pode ser maior do que o percentual máximo de desconto.

Com essa lógica de negócio implementada, devemos criar testes unitários para esse método:

```
public class ProductTest {
  @Test
  public void shouldCalculateTotalPriceWithDiscount() {
     Product product =  new Product();
     product.setPrice( 100.00);
     product.setMaxDiscount( 0.2);
     double priceWithDiscount =
product.getPriceWithDiscount( 0.1);
     assertEquals( 90, priceWithDiscount);
  }

  @Test
  public void shouldCalculatePriceWithMaxDiscount() {
     Product product =  new Product();
     product.setPrice( 100.0);
     product.setMaxDiscount( 0.2);
     double priceWithDiscount =
product.getPriceWithDiscount( 0.3);
     assertEquals( 80, priceWithDiscount);
  }
}
```

Para isso criamos dois testes:

- Cálculo do preço com desconto menor do que o desconto máximo permitido;
- Cálculo do preço com desconto máximo aplicado. Nesse caso, o desconto máximo permitido é de 20%.

Testes de Integração

Testes de integração são usados quando queremos validar um fluxo que é formado pela interação de diversas classes. No contexto de desenvolvimento de APIs, os testes de integração são usados para testar o endpoints da aplicação:

```java
@SpringBootTest
@DirtiesContext(classMode =
DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
@AutoConfigureTestDatabase(replace =
AutoConfigureTestDatabase.Replace.ANY)
@AutoConfigureMockMvc
public class ProductControllerTests {
  @Autowired
  private MockMvc mockMvc;

  @Test
  public void shouldCreateANewProduct()  throws Exception {
     String requestBody =  "{\n" +
           "    \"description\": \"Pilha\",\n" +
           "    \"supplier\": \"Duracell\",\n" +
           "    \"price\": 13.99,\n" +
           "    \"maxDiscount\": 0.10\n" +
           "}";

     // Testa apenas se o status HTTP é 200 (OK)
     mockMvc.perform(MockMvcRequestBuilders
                 .post( "/products")
                 .content(requestBody)
                 .contentType( "application/json"))
           .andExpect(status().isCreated());
  }
}
```

No exemplo acima temos um teste de integração que testa a requisição de criação de um produto.

Testes end-to-end

Os **testes end-to-end** são mais complexos e são usados para validar o comportamento de um aplicação como se estivesse sendo usada por um usuário real.

Também são conhecidos por testes de interface, pois envolvem simulação de cliques, aberturas de janelas e outros comportamentos.

Um exemplo de uso de **testes end-to-end** seria para realizar o fechamento de uma compra em um e-commerce, onde é testado desde o acesso ao catálogo de produtos, passando pela criação do carrinho de compra, entrada do endereço, informações de pagamento até chegar ao fechamento.

Testes end-to-end também são muito usados no contexto de microsserviços, onde é validado a comunicação das APIs umas com as outras.