



# **Teste de Software**

Prof. Dr. Bruno Queiroz Pinto

# Testes Automatizados utilizando **Mockito**


- ✓ As classes de um projeto podem depender de outras classes que executam algum serviço externo.
  - Por exemplo: Uma classe Service depende da classe Repository que acessa um banco de dados.
- ✓ Às vezes, queremos testar apenas a lógica daquela classe de maneira isolada, mas pelo fato de ela depender de outra classe, não é possível isolar essas dependências e testá-la individualmente.
- ✓ Solução: fazer integração ou usar um **Mock**.

# Testes Automatizados utilizando **Mockito**

- ✓ Essa característica também é essencial para projetos que utilizam TDD. Como testar uma classe que depende de outras, mesmo essas ainda não existindo?
  - **Utilizar Stubs/Mocks.**
- ✓ Cenário 1:
  - Classes Calculadora e CalculadoraService;
- ✓ Cenário 2:
  - Classes ClientRepository e ClientService;

# Testes Automatizados utilizando **Mockito**


## ✓ Cenário 1:

- Github: <https://github.com/brunoqp78/calculadora-mocks-modelo>
- Classes Calculadora e CalculadoraService;
- Criar testes para a classe Calculadora; 
- Criar testes para a classe CalculadoraService;
  - ✓ CalculadoraService depende de Calculadora?

# Testando CalculadoraService com Mockito

```
public class CalculadoraService {  
    @Autowired  
    private Calculadora calc;  
  
    public double calculo(double n1, double n2) {  
        return calc.somar(n1, n2)*10;  
    }  
}
```

Depende de  
Calculadora




# Testes Automatizados Spring Boot – Projeto Client

```
@Service
public class ClientService {

    @Autowired
    private ClientRepository repository;

    @Transactional(readOnly = true)
    public Page<ClientDT0> findAllPaged(PageRequest pageRequest)
    {
        Page<Client> list = repository.findAll(pageRequest);
        return list.map(x -> new ClientDT0(x));
    }
}
```



A blue rectangular box on the right side of the slide contains the text "ClientService depende ClientRepository". A red arrow points from this box to the "repository" variable in the code, which is also enclosed in a red rectangular box.

# O que são Mocks?

- ✓ Objetos que simulam os comportamentos dos objetos reais são o que chamamos de mock objects.
- ✓ Mock objects são objetos que fingem ser outros objetos.
- ✓ Ele serve para cenários em que queremos testar a lógica e os algoritmos de uma classe que tem dependência de outra classe, mas isolando essas dependências.

# O que são Mocks?

- ✓ Com os Mocks, conseguimos escrever um teste de unidade em vez de ter que usar um teste de integração, ou seja, que vai se integrar às dependências.



# Testando CalculadoraService com Mockito

```
public class CalculadoraService {  
    @Autowired  
    private Calculadora calc;  
  
    public double calculo(double n1, double n2) {  
        return calc.somar(n1, n2)*10;  
    }  
}
```

Depende de  
Calculadora

Para testar a classe CalculadoraService não iremos utilizar o objeto real, nós iremos criar um mock dessa classe.

# Testando CalculadoraService com Mockito

```
package org.brsoft.test.process;

import org.brsoft.service.CalculadoraService;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
public class CalculadoraServiceTest {
    // define qual classe receberá os Mocks criados
    @InjectMocks
    private CalculadoraService process;
}
```

Necessário para o Mockito

- criar o objeto de classe a ser testada que receberá a injeção de mocks. Para fazer isso, usamos a notação `@InjectMocks`.

**`@InjectMocks` marca um campo no qual a injeção deve ser realizada.**

# Vamos para a prática?


<code>@SpringBootTest</code>	Carrega o contexto da aplicação
<code>@SpringBootTest</code> <code>@AutoConfigureMockMvc</code>	Carrega o contexto da aplicação Trata as requisições sem subir o servidor
<code>@WebMvcTest</code>	Carrega o contexto, porém somente da camada web
<code>@ExtendWith(SpringExtension.class)</code>	Não carrega o contexto, mas permite usar os recursos do Spring com JUnit
<code>@DataJpaTest</code>	Carrega somente os componentes relacionados ao Spring Data JPA. Cada teste é transacional e dá rollback ao final.

Testes na Repository

Testes na Service

# Testando CalculadoraService com Mockito

```
public class CalculadoraServiceTest {  
    // define qual classe receberá os Mocks criados  
    @InjectMocks  
    private CalculadoraService process;  
  
    @Mock  
    private Calculadora calc;  
}
```



- usada para criar e injetar instâncias simuladas . Não são criados objetos reais, em vez disso, o mockito cria um mock para a classe.

@Mock

- permite a criação abreviada de objetos necessários para o teste.
- minimiza o código de criação de simulação repetitiva.
- torna a classe de teste mais legível.

# Testando CalculadoraService com Mockito

```
public class CalculadoraServiceTest {  
    // define qual classe receberá os Mocks criados  
    @InjectMocks  
    private CalculadoraService process;  
  
    private Calculadora calc = Mockito.mock(Calculadora.class);  
}
```

## Outra forma



@Mock é alternativa a Mockito.mock(classToMock).

- Ambos alcançam o mesmo resultado.
- @Mock é geralmente considerado “ mais limpo ”.

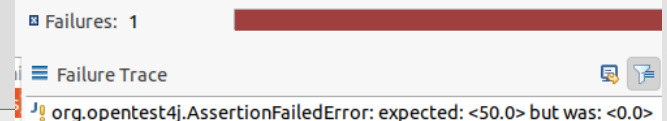
# Testando CalculadoraService com Mockito

```
public class CalculadoraServiceTest {
    @InjectMocks
    private CalculadoraService process;
    @Mock
    private Calculadora calc;

    @Test
    public void testaCalculoN1MaiorN2() {
        double numero1 = 3;
        double numero2 = 2;
        double resultadoEsperado = 50;
        double resultadoObtido = process.calculo(n1, n2);
        Assertions.assertEquals(resultadoEsperado, resultadoObtido);
    }
}
```

- O código acima utiliza o Mock original gerado. Observe que ele não passa no teste, pois um Mock original retorna o valor default 0.0 em um método que deveria retornar o valor 5.0, considerando a entrada fornecida.

Ou seja, precisamos configurar o nosso Mock.



# Testando CalculadoraService com Mockito

```
:
:
@Test
public void testaCalculoN1MaiorN2() {
    double numero1 = 3;
    double numero2 = 2;
    double resultadoEsperado = 50;
    Mockito.when(calc.somar(n1, n2)).thenReturn(5.0);
    double resultadoObtido = process.calculo(n1, n2);
    Assertions.assertEquals(resultadoEsperado, resultadoObtido);
}
}
```

- O comando Mockito.when é utilizado para ensinar o objeto Mock o que ele deve fazer em certas situações. O when basicamente determina qual método esperamos que seja chamado no futuro e com quais parâmetros.
- O thenReturn diz qual será o valor devolvido quando o método do comando when for chamado.

# Testando CalculadoraService com Mockito

```
:
:
@Test
public void testaCalculoN1MaiorN2() {
    double numero1 = 3;
    double numero2 = 2;
    double resultadoEsperado = 50;
    Mockito.when(calc.somar(n1, n2)).thenReturn(5.0);
    double resultadoObtido = process.calculo(n1, n2);
    Assertions.assertEquals(resultadoEsperado, resultadoObtido);
    Mockito.verify(calc, Mockito.times(1)).somar(numero1, numero2);
}
}
```

- O comando Mockito.verify verifica se um determinado método (somar) do objeto mock (calc) foi executado uma determinada quantidade de vezes(Mockito.times(1)).



# Testando CalculadoraService com Mockito

```
public class CalculadoraServiceTest {
    private double numero1, numero2, somaN1N2;
    @InjectMocks
    private CalculadoraService servico;
    @Mock
    private Calculadora calculadora;

    @BeforeEach
    public void configuraMocks() {
        // assign
        numero1 = 3;
        numero2 = 2;
        somaN1N2 = 5;
        Mockito.when(calculadora.somar(numero1, numero2)).thenReturn(somaN1N2);
        Mockito.when(calculadora.somar(numero1, numero1)).thenReturn(numero1);
        Mockito.when(calculadora.somar(numero2, numero1)).thenReturn(numero2);
    }

    @Test
    public void testaCalculoN1MaiorN2() {
        // assign
        double resultadoEsperado = 50;
        // act
        double resultadoObtido = servico.calculo(numero1, numero2);
        // assert
        assertEquals(resultadoEsperado, resultadoObtido);
    }

    @Test
    public void testaCalculoN1igualN2() {
        // assign
        double resultadoEsperado = 30;
        // act
        double resultadoObtido = servico.calculo(numero1, numero1);
        // assert
        assertEquals(resultadoEsperado, resultadoObtido);
        Mockito.verify(calculadora, Mockito.times(1)).somar(numero1, numero1);
    }

    @Test
    public void testaCalculoN1menorN2() {
        // assign
        double resultadoEsperado = 20;
        // act
        double resultadoObtido = servico.calculo(numero2, numero1);
        // assert
        assertEquals(resultadoEsperado, resultadoObtido);
        Mockito.verify(calculadora, Mockito.times(1)).somar(numero2, numero1);
    }
}
```

Otimizando



# Testando CalculadoraService com Mockito

Nova regra, soma retorna Exception quando N1 ou N2 menores ou iguais a 0.

```
@BeforeEach
public void configuraMocks() {
    // assign
    numero1 = 3;
    numero2 = 2;
    somaN1N2 = 5;
    Mockito.when(calculadora.somar(numero1, numero2)).thenReturn(somaN1N2);
    Mockito.when(calculadora.somar(numero1, numero1)).thenReturn(numero1);
    Mockito.when(calculadora.somar(numero2, numero1)).thenReturn(numero2);
    Mockito.doThrow(InvalidParameterException.class).when(calculadora).somar(0, numero1);
    Mockito.doThrow(InvalidParameterException.class).when(calculadora).somar(numero1, 0);
    Mockito.doThrow(InvalidParameterException.class).when(calculadora).somar(0, 0);
}

@Test
public void testaCalculoN1Zero() {
    // act e assert
    assertThrows(InvalidParameterException.class, () -> {servico.calculo(0, numero1)});
    Mockito.verify(calculadora, Mockito.times(1)).somar(0, numero1);
}

@Test
public void testaCalculoN2Zero() {
    // act e assert
    assertThrows(InvalidParameterException.class, () -> {servico.calculo(numero1, 0)});
    Mockito.verify(calculadora, Mockito.times(1)).somar(numero1, 0);
}

@Test
public void testaCalculoN1N2Zero() {
    // act e assert
    assertThrows(InvalidParameterException.class, () -> {servico.calculo(0, 0)});
    Mockito.verify(calculadora, Mockito.times(1)).somar(0, 0);
}
```

# Testando CalculadoraService com Mockito

## Outros métodos de configuração do Mockito

`Mockito.doNothing().when(calculadora).metodoRetornoVoid();`

- utilizado no caso de sucesso de um método que retorna void.
- exemplo uma exclusão correta no banco de dados.
- `Mockito.doNothing().when(repository).deleteById(existingId);`

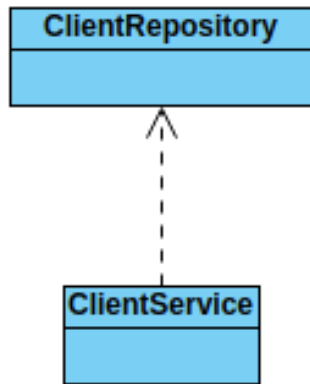
`Mockito.doCallRealMethod().when(calculadora).algumMetodoVoid();`

- utilizado quando queremos chamar o método original da classe.

`Mockito.doReturn(5.0).when(calculadora).somar(numero1, numero2);`

- outra forma de : `Mockito.when(calculadora.somar(numero1, numero2)).thenReturn(5.0);`

## Vamos para a prática? - Projeto Client



A classe **ClientService** depende das funcionalidades da classe **ClientRepository**.


Para testar a classe **Service** iremos criar um **MOCK** da classe **Repository**.

# Testes Automatizados Spring Boot – Projeto Client

```
@Service
public class ClientService {



    @Autowired
    private ClientRepository repository;

    @Transactional(readOnly = true)
    public Page<ClientDT0> findAllPaged(PageRequest pageRequest)
    {
        Page<Client> list = repository.findAll(pageRequest);
        return list.map(x -> new ClientDT0(x));
    }
}
```



A blue rectangular box on the right side of the slide contains the text "ClientService depende ClientRepository". A red arrow points from this box to the "repository" variable in the code, which is also enclosed in a red rectangular box.

# Vamos para a prática?

▼  > com.iftm.client.tests.services  
▶  ClientServiceTests.java

Criar o package services dentro de tests.

Criar a classe de testes:  
ClientServiceTests.java

Classe responsável em testar a classe de serviço ClientService

# Vamos para a prática?

```
package com.iftm.client.tests.services;

import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
public class ClientServiceTests {

}
```

Não carrega o contexto,  
mas permite usar os  
recursos do Spring com  
JUnit

# Vamos para a prática?

<code>@SpringBootTest</code>	Carrega o contexto da aplicação
<code>@SpringBootTest</code> <code>@AutoConfigureMockMvc</code>	Carrega o contexto da aplicação Trata as requisições sem subir o servidor
<code>@WebMvcTest</code>	Carrega o contexto, porém somente da camada web
<code>@ExtendWith(SpringExtension.class)</code>	Não carrega o contexto, mas permite usar os recursos do Spring com JUnit
<code>@DataJpaTest</code>	Carrega somente os componentes relacionados ao Spring Data JPA. Cada teste é transacional e dá rollback ao final.

Testes na Repository

Testes na Service



# Vamos para a prática?

```
package com.iftm.client.tests.services;

import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.mockito.Mock;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import com.iftm.client.repositories.ClientRepository;
import com.iftm.client.services.ClientService;

@ExtendWith(SpringExtension.class)
public class ClientServiceTests {

    @InjectMocks
    private ClientService servico;

    @Mock
    private ClientRepository repositorio;
}
```

Classe que irá utilizar o mock.

Classe que será simulada

# Vamos para a prática?

@BeforeEach

```
void setUp() throws Exception {  
    idExistente = 1L;  
    idNaoExistente = 1000L;  
    idDependente = 4L;
```

No projeto ainda não foi implementado

```
Mockito.doNothing().when(repositorio).deleteById(idExistente);  
Mockito.doThrow(EmptyResultDataAccessException.class).when(repositorio).deleteById(idNaoExistente);  
Mockito.doThrow(DataIntegrityViolationException.class).when(repositorio).deleteById(idDependente);  
}
```

@Test

```
public void apagarNaoDeveFazerNadaQuandoIdExiste() {  
    Assertions.assertDoesNotThrow(() -> {servico.delete(idExistente);});  
    Mockito.verify(repositorio, Mockito.times(1)).deleteById(idExistente);  
}
```

# Vamos para a prática?

```
@Test
public void apagarNaoDeveFazerNadaQuandoIdExiste() {
    Assertions.assertDoesNotThrow(()->{servico.delete(idExistente)});
    Mockito.verify(repositorio, Mockito.times(1)).deleteById(idExistente);
}
```

```
@Test
public void apagarGeraExcecaoQuandoIdNaoExiste() {
    Assertions.assertThrows(ResourceNotFoundException.class, ()->{servico.delete(idNaoExistente)});
    Mockito.verify(repositorio, Mockito.times(1)).deleteById(idNaoExistente);
}
```

```
@Test
public void apagarGeraExcecaoQuandoIdTemDependencia() {
    Assertions.assertThrows(DatabaseException.class, ()->{servico.delete(idDependente)});
    Mockito.verify(repositorio, Mockito.times(1)).deleteById(idDependente);
}
```