CAPACITAÇÕES
JAVA PARA A PDPJ-Br

TESTES AUTOMATIZADOS

CADERNO DE ATIVIDADES - TRILHA 1

Samantha Kelly Soares de Almeida

Unifor | JUSTIÇA DO TRABALHO Conselho Superior da Justiça do Trabalho | Tribunal Superior Eleitoral | STJ SUPERIOR TRIBUNAL DE JUSTIÇA | JUSTIÇA 40 | PNUD | JUSTIÇA FEDERAL Conselho da Justiça Federal | CNJ CONSELHO NACIONAL DE JUSTIÇA

ATIVIDADES DO CADERNO

Atividade 1

Objetivos: Vivenciar a prática de testes unitários, para uma calculadora, e desenvolver um teste unitário, para cada método presente na calculadora.

Requisitos: Não tem.

Programa Calculadora

```
public class Calculadora {
    public int soma(int a, int b) {
        return a + b;
    }
    public int subtracao(int a, int b) {
        return a - b;
    }
    public int multiplicacao(int a, int b) {
        return a * b;
    }
    public int divisao(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Divisão por zero não é permitida.");
        }
        return a / b;
    }
}
```

Passo 1: Configurar as dependências no Maven, adicionar JUnit ao seu arquivo pom.xml:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
 <dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
```

Passo 2: Criar o caso de teste, para cada método da calculadora.

Passo 3: Executar os testes no Eclipse, assegurando-se de que passaram.

Atividade 2

Objetivo: Aplicar essa convenção de nomenclatura em seus testes, garantindo que sejam fáceis de entender, que comuniquem claramente o que está sendo testado e sob quais condições.

Requisitos: Não tem.

Classes a serem testadas:

```java
public class ContaBancária {
    private double saldo;

    public ContaBancária(double saldoInicial) {
        this.saldo = saldoInicial;
    }

    public void depositar(double quantia) {
        if (quantia < 0) {
            throw new IllegalArgumentException("Quantia para depósito deve ser positiva.");
        }
        this.saldo += quantia;
    }

    public void sacar(double quantia) {
        if (quantia > this.saldo) {
            throw new IllegalArgumentException("Saldo insuficiente.");
        }
        this.saldo -= quantia;
    }

    public double obterSaldo() {
        return this.saldo;
    }
}
```

Elaborada pela autora, 2024.

Utilize a convenção “deve_fazer_isto_quando_isto”, para criar testes unitários, para a classe **ContaBancária**. Aqui estão algumas sugestões de cenários, para serem testados:

1. **Testes de Depósito**
   - Deve aumentar o saldo, quando depositar uma quantia positiva.
   - Deve lançar exceção, quando tentar depositar uma quantia negativa.
2. **Testes de Saque**
   - Deve diminuir o saldo, quando sacar uma quantia menor ou igual ao saldo.
   - Deve lançar exceção, quando tentar sacar uma quantia maior que o saldo.
3. **Testes de Obtenção de Saldo**
   - Deve retornar o saldo correto, após operações de depósito e saque.
   - Deve retornar zero, quando a conta é criada com saldo inicial zero.

Atividade 3

Objetivo: Vivenciar a prática de TDD, para os métodos de soma e subtração de uma calculadora simples.

Requisitos: Não tem.

Passo 1: Configurar as dependências no Maven, adicionar JUnit ao seu arquivo pom.xml:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
 <dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
```

Passo 2: Criar o caso de teste para a calculadora, mesmo que ela ainda não exista.

Exemplo para SOMA:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
 public class CalculadoraTest {
    @Test
   void testSoma() {
      Calculadora calculadora = new Calculadora();
```

```
    int resultado = calculadora.soma(2, 3);
    assertEquals(5, resultado);  // Esperado 2 + 3 = 5
  }
```

Crie o teste para a função SUBTRAIR.

Passo 3: Executar os métodos e assegurar-se de que os testes falham ao serem executados, pois a classe Calculadora ainda não foi implementada.

Passo 4: Implementar a função SUBTRAIR.

Exemplo Função SOMA:

```
public class Calculadora {
   public int soma(int a, int b) {
     return a + b;
  }
}
```

Passo 5: Depois de implementar a classe subtraiCalculadora, executar novamente os testes, para verificar se todos passam.

Atividade 4

Objetivo: Implementar um teste unitário para o método obterTodasCategorias, que deve verificar se as categorias retornadas para o usuário informado estão de acordo com o esperado.

Requisitos: Não tem.

Método a ser testado:

```
public List<Categoria> obterTodasCategorias(String userLogin) {
    return categoriaRepository.findAllCategoriasByUsuarioLogin(userLogin);
}
```

Elaborada pela autora, 2024.

Passo a passo:
1 – Criar uma classe de testes, no projeto de testes, atendendo ao padrão AAA.
2 – Executar a classe de testes, através do comando Run do Eclipse, para verificar se a construção foi executada com sucesso.

Observação: Caso o teste falhe, você deverá buscar o erro cometido, durante a construção do teste, e corrigi-lo, até que o teste fique com o comando de OK (cor verde), após ser executado no Eclipse.

Nota: Utilizar, como base, o teste do método deveObterTodasContas.

```
@Test
void deveObterTodasContas() {

    //Arrange
    String loginUsuario = "user@login.com";

    List<Conta> contasEsperadas = List.of(
            Conta.builder().nome("Conta Corrente").build(),
            Conta.builder().nome("Cartão Crédito").build());

    Mockito.when(contaRepositoryMock.findAllContasByUsuarioLogin(loginUsuario)).thenReturn(contasEsperadas);

    //Act
    List<Conta> contasObtidas = contaService.obterTodasContas(loginUsuario);

    //Assert
    Assertions.assertEquals(contasEsperadas, contasObtidas);
}
```

Elaborada pela autora, 2024.

Atividade 5

Objetivo: Implementar um teste unitário para o método obterCategoriaPorId, que deve verificar se a categoria retornada corresponde ao Id de categoria e o usuário informado como parâmetro.

Requisitos: Não tem.

Método a ser testado:

```
public Categoria obterCategoriaPorId(String idCategoria, String userLogin) {

    validacaoDadosUsuarioService.validarCategoriaDoUsuarioLogado(idCategoria, userLogin);

    return categoriaRepository.findById(idCategoria).orElseThrow();
}
```

Elaborada pela autora, 2024.

Passo a passo:
1 – Criar uma classe de testes, no projeto de testes, atendendo ao padrão AAA.
2 – Executar a classe de testes, através do comando Run do Eclipse, para verificar se a construção foi executada com sucesso.

Observação: Caso o teste falhe, você deverá buscar o erro cometido, durante a construção do teste, e corrigi-lo, até que o teste fique com o comando de OK (cor verde), após ser executado no Eclipse.

Nota: Utilizar como base o teste do método deveObterContaPorId.

```java
@Test
void deveObterContaPorId() {

    //Arrange
    String idConta = "123";
    String loginUsuario = "user@login.com";

    Conta contaEsperada = Conta.builder().nome("Cartão Crédito").build();
    Optional<Conta> contaEsperadaOpt = Optional.of(contaEsperada);
    Mockito.when(contaRepositoryMock.findById(idConta)).thenReturn(contaEsperadaOpt);

    //Act
    Conta contaResultado = contaService.obterContaPorId(idConta, loginUsuario);

    //Assert
    Assertions.assertEquals(contaEsperada, contaResultado);

    Mockito.verify(validacaoDadosUsuarioServiceMock)
            .validarContaDoUsuarioLogado(idConta, loginUsuario);
    Mockito.verify(contaRepositoryMock).findById(idConta);
}
```

Elaborada pela autora, 2024.

Atividade 6

Objetivo: Implementar um teste unitário para o método criarCategoria, que deve verificar se a categoria retornada corresponde à categoria criada, a partir dos dados informados como parâmetro.

Requisitos: Não tem.

Método a ser testado:

```
@Transactional
public Categoria criarCategoria(CategoriaRequestDTO categoriaDTO, String userLogin) {

    List<Categoria> categorias = categoriaRepository.findAllCategoriasByUsuarioLogin(userLogin);
    validarCategoriaComMesmoNome(categoriaDTO.nome(), categorias);

    UserDetails usuario = usuarioRepository.findByLogin(userLogin);
    Categoria c = new Categoria();
    c.setNome(categoriaDTO.nome());
    c.setUsuario((Usuario) usuario);

    return categoriaRepository.save(c);
}
```

Elaborada pela autora, 2024.

Passo a passo:
1 – Criar uma classe de testes, no projeto de testes, atendendo ao padrão AAA.
2 – Executar a classe de testes, através do comando Run do Eclipse, para verificar se a construção foi executada com sucesso.

Observação: Caso o teste falhe, você deverá buscar o erro cometido, durante a construção do teste, e corrigi-lo, até que o teste fique com o comando de OK (cor verde), após ser executado no Eclipse.

Nota: Utilizar, como base, o teste do método deveCriarConta.

```java
@Test
void deveCriarConta() {

    //Arrange
    String loginUsuario = "user@login.com";
    String nomeNovaConta = "Conta Corrente";

    ContaRequestDTO novaContaDto = new ContaRequestDTO( id: null, nomeNovaConta);

    List<Conta> contasExistentes = List.of(Conta.builder().nome("Cartão Crédito").build());
    Mockito.when(contaRepositoryMock.findAllContasByUsuarioLogin(loginUsuario)).thenReturn(contasExistentes);

    Usuario usuario = Usuario.builder().login(loginUsuario).id("1234").build();
    Mockito.when(usuarioRepositoryMock.findByLogin(loginUsuario)).thenReturn(usuario);

    Conta contaEsperada = Conta.builder().nome(nomeNovaConta).usuario(usuario).build();
    Mockito.when(contaRepositoryMock.save(Mockito.any(Conta.class))).thenReturn(contaEsperada);

    //Act
    Conta contaResultado = contaService.criarConta(novaContaDto, loginUsuario);

    //Assert
    Assertions.assertEquals(contaEsperada, contaResultado);
}
```

Elaborada pela autora, 2024.

Atividade 7

Objetivo: Implementar um teste unitário, para o método atualizarCategoria, que deve verificar se a categoria retornada corresponde à categoria atualizada, a partir dos dados informados como parâmetro.

Requisitos: Não tem.

Método a ser testado:

```java
@Transactional
public Categoria atualizarCategoria(String idCategoria, CategoriaRequestDTO categoriaDTO, String userLogin) {

    validacaoDadosUsuarioService.validarCategoriaDoUsuarioLogado(idCategoria, userLogin);

    List<Categoria> categorias = categoriaRepository.findAllCategoriasByUsuarioLogin(userLogin);
    validarCategoriaComMesmoNome(categoriaDTO.nome(), categorias);

    Categoria categoria = categoriaRepository.findById(idCategoria).orElseThrow();
    categoria.setNome(categoriaDTO.nome());

    return categoriaRepository.save(categoria);
}
```

Elaborada pela autora, 2024.

Passo a passo:

1 – Criar uma classe de testes, no projeto de testes, atendendo ao padrão AAA.
2 – Executar a classe de testes, através do comando Run do Eclipse, para verificar se a construção foi executada com sucesso.

Observação: Caso o teste falhe, você deverá buscar o erro cometido, durante a construção do teste, e corrigi-lo, até que o teste fique com o comando de OK (cor verde), após ser executado no Eclipse.

Nota: Utilizar, como base, o teste do método deveAtualizarConta.

```
@Test
void deveAtualizarConta() {

    //Arrange
    String loginUsuario = "user@login.com";
    String nomeNovaConta = "Conta Conjunta";

    ContaRequestDTO novaContaDto = new ContaRequestDTO( id: null, nomeNovaConta);

    Conta cartaoCredito = Conta.builder().nome("Cartão Crédito").build();
    Conta contaCorrente = Conta.builder().nome("Conta Corrente").build();
    List<Conta> contasExistentes = List.of(cartaoCredito, contaCorrente);
    Mockito.when(contaRepositoryMock.findAllContasByUsuarioLogin(loginUsuario)).thenReturn(contasExistentes);

    Usuario usuario = Usuario.builder().login(loginUsuario).id("1234").build();
    String idConta = "id_Conta";
    Mockito.when(contaRepositoryMock.findById(idConta)).thenReturn(Optional.of(contaCorrente));

    Conta contaEsperada = Conta.builder().nome(nomeNovaConta).usuario(usuario).build();
    Mockito.when(contaRepositoryMock.save(Mockito.any(Conta.class))).thenReturn(contaEsperada);

    //Act
    Conta contaResultado = contaService.atualizarConta(idConta, novaContaDto, loginUsuario);

    //Assert
    Assertions.assertEquals(contaEsperada, contaResultado);
}
```

Elaborada pela autora, 2024.

Atividade 8

Objetivo: Implementar um teste unitário, para o método deletarCategoria, que deve verificar se as que são responsáveis por deletar a categoria foram chamadas corretamente.

Requisitos: Não tem.

Método a ser testado:

```
@Transactional
public void deletarConta(String idConta, String userLogin) {

    validacaoDadosUsuarioService.validarContaDoUsuarioLogado(idConta, userLogin);

    Conta conta = contaRepository.findById(idConta).orElseThrow();
    if (conta.getLancamentos() != null && !conta.getLancamentos().isEmpty()) {
        throw new NegocioException("Não é possivel remover essa conta, pois ela possui lançamentos associados.");
    }
    contaRepository.deleteById(idConta);
}
```

Elaborada pela autora, 2024.

Passo a passo:
1 – Criar uma classe de testes, no projeto de testes, atendendo ao padrão AAA.
2 – Executar a classe de testes, através do comando Run do Eclipse, para verificar se a construção foi executada com sucesso.

Observação: Caso o teste falhe, você deverá buscar o erro cometido, durante a construção do teste, e corrigi-lo, até que o teste fique com o comando de OK (cor verde), após ser executado no Eclipse.

Validações do Teste:
Passo 1: Validar se a categoria pertence ao usuário.
Passo 2: Verificar se o repositório responsável pela deleção da categoria foi chamado.

Nota: Utilizar, como base, o teste do método deveDeletarConta.

```java
@Test
void deveDeletarConta() {

    //Arrange
    String loginUsuario = "user@login.com";
    String idConta = "id_Conta";

    Mockito.when(contaRepositoryMock.findById(idConta)).thenReturn(Optional.of(Conta.builder().build()));

    Mockito.doNothing().when(contaRepositoryMock).deleteById(idConta);

    //Act
    contaService.deletarConta(idConta, loginUsuario);

    //Assert
    Mockito.verify(validacaoDadosUsuarioServiceMock).validarContaDoUsuarioLogado(idConta, loginUsuario);
    Mockito.verify(contaRepositoryMock).deleteById(idConta);
}
```

Elaborada pela autora, 2024.

Atividade 9

Objetivo: Implementar um teste unitário de exceção, para o método validarCategoriaComMesmoNome, que deve verificar se uma exceção de negócio é retornada, ao passar um nome de categoria e uma lista de categorias na qual esse nome deve ser verificado se já existe.

Requisitos: Não tem.

```java
protected void validarCategoriaComMesmoNome(String nomeCategoria, List<Categoria> categorias) {

    boolean possuiCategoriaComMesmoNome = categorias.stream()
            .anyMatch(c -> c.getNome().equals(nomeCategoria));
    if (possuiCategoriaComMesmoNome) {
        throw new NegocioException("Categoria com nome informado já existente.");
    }
}
```

Elaborada pela autora, 2024.

Passo a passo:
1 – Criar uma classe de testes, no projeto de testes, atendendo ao padrão AAA.
2 – Executar a classe de testes, através do comando Run do Eclipse, para verificar se a construção foi executada com sucesso.

Observação: Caso o teste falhe, você deverá buscar o erro cometido, durante a construção do teste, e corrigi-lo, até que o teste fique com o comando de OK (cor verde), após ser executado no Eclipse.

Nota: Utilizar, como base, o teste do método deveLancarErroAoValidarContaMesmoNome.

Atividade 10

Objetivo: Implementar um teste unitário, para o método findAllCategoriasByUsuarioLogin, que deve verificar se todas as categorias inseridas no teste são retornadas.

Requisitos: Não tem.

```
public interface CategoriaRepository extends JpaRepository<Categoria, String> {

    List<Categoria> findAllCategoriasByUsuarioLogin(String userId);
}
```

Elaborada pela autora, 2024.

Passo a passo:
1 – Criar uma classe de testes, no projeto de testes, atendendo ao padrão AAA.
2 – Executar a classe de testes, através do comando Run do Eclipse, para verificar se a construção foi executada com sucesso.

Observação: Caso o teste falhe, você deverá buscar o erro cometido, durante a construção do teste, e corrigi-lo, até que o teste fique com o comando de OK (cor verde), após ser executado no Eclipse.

Validações do Teste:
Passo 1: Validar se a categoria pertence ao usuário.
Passo 2: Verificar se o repositório responsável pela deleção da categoria foi chamado.

Nota: Utilizar, como base, o teste do método deveObterAsContasDoUsuario.

```java
@Test
public void deveObterAsContasDoUsuario() {

    // Arrange
    String userLogin = "joao@teste.com";
    String userPassword = "senha_do_joao";
    var user = Usuario.builder().login(userLogin).password(userPassword).role(UserRole.ADMIN).build();

    String nomeContaCorrente = "Conta Corrente";
    Conta contaCorrente = Conta.builder().nome(nomeContaCorrente).usuario(user).build();
    String nomeCartaoCredito = "Cartão Crédito";
    Conta contaCartaoCredito = Conta.builder().nome(nomeCartaoCredito).usuario(user).build();

    usuarioRepository.save(user);
    contaRepository.save(contaCorrente);
    contaRepository.save(contaCartaoCredito);

    // Act
    List<Conta> contasUsuario = contaRepository.findAllContasByUsuarioLogin(userLogin);

    // Assert
    Assertions.assertEquals( expected: 2, contasUsuario.size());
}
```

Elaborada pela autora, 2024.

Atividade 11

Objetivo: Implementar um teste unitário, para o método createCategoria, que deve verificar se o retorno HTTP é igual a 201 (CREATED) e o objeto JSON retornado está de acordo com o esperado.

Requisito: Não tem.

```java
@PostMapping
public ResponseEntity<CategoriaResponseDTO> createCategoria(@RequestBody @Valid CategoriaRequestDTO categoria, Authentication authentication) {

    Categoria categoriaCriada = categoriaService.criarCategoria(categoria, authentication.getName());

    return ResponseEntity.status(HttpStatus.CREATED).body(new CategoriaResponseDTO(categoriaCriada));
}
```

Elaborada pela autora, 2024.

Passo a passo:
1 – Criar uma classe de testes, no projeto de testes, atendendo ao padrão AAA.
2 – Executar a classe de testes, através do comando Run do Eclipse, para verificar se a construção foi executada com sucesso.

Observação: Caso o teste falhe, você deverá buscar o erro cometido, durante a construção do teste, e corrigi-lo, até que o teste fique com o comando de OK (cor verde), após ser executado no Eclipse.

Validações do Teste:
Passo 1: Validar se a categoria pertence ao usuário.
Passo 2: Verificar se o repositório responsável pela deleção da categoria foi chamado.

Nota: Utilizar, como base, o teste do método deveInserirUmaConta.

```
@Test
@WithUserDetails("usuarioTeste")
public void deveInserirUmaConta() throws Exception {

    // Arrange
    ContaRequestDTO contaRequestDTO = new ContaRequestDTO( id: null,  nome: "Conta Corrente");

    Conta conta = Conta.builder().id("1234").nome("Conta Corrente").build();
    Mockito.when(contaService.criarConta(ArgumentMatchers.any(ContaRequestDTO.class), ArgumentMatchers.any(String.class)))
            .thenReturn(conta);

    // Converte o objeto para JSON
    String jsonContent = objectMapper.writeValueAsString(contaRequestDTO);

    mockMvc.perform(
                    // Act
                    post( urlTemplate: "/api/contas")
                            .with(SecurityMockMvcRequestPostProcessors.csrf())
                            .header( name: "Authorization",  ...values: "Bearer " + "fake-token-jwt")
                            .accept(MediaType.APPLICATION_JSON)
                            .contentType(MediaType.APPLICATION_JSON)
                            .content(jsonContent)
            )
            // Assert
            .andExpect(MockMvcResultMatchers.status().isCreated())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.id").value( expectedValue: "1234"))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.nome").value( expectedValue: "Conta Corrente"));
}
```

Elaborada pela autora, 2024.