Bootcamp Spring 3.0 - Cap. 02

Testes automatizados no back end

# Pré-requisitos

* Back end DSCatalog (capítulo 1)

# Competências

* Fundamentos de testes automatizados
  + Tipos de testes
  + Benefícios
  + TDD - Test Driven Development
  + Boas práticas e padrões
* JUnit
  + Básico (vanilla)
  + Spring Boot
    - Repositories
    - Services
    - Resources (web)
    - Integração
* Mockito & MockBean
  + @Mock
  + @InjectMocks
  + Mockito.when / thenReturn / doNothing / doThrow
  + ArgumentMatchers
  + Mockito.verify
  + @MockBean
  + @MockMvc

# Etapas

* Fundamentos + JUnit vanilla (exercício de fixação)
* Testes de repository (exercício de fixação)
* Testes de unidade com Mockito (exercício de fixação)
* Testes da camada web com MockMvc (exercício de fixação)
* Testes de integração
* Desafio TDD (desafio final para entregar)

# Fundamentos de testes automatizados

## Tipos de testes

### Unitário

Teste feito pelo desenvolvedor, responsável por validar o comportamento de unidades funcionais de código. Nesse contexto, entende-se como unidade funcional qualquer porção de código que através de algum estímulo seja capaz de gerar um comportamento esperado (na prática: métodos de uma classe). Um teste unitário não pode acessar outros componentes ou recursos externos (arquivos, bd, rede, web services, etc.).

### Integração

Teste focado em verificar se a comunicação entre componentes / módulos da aplicação, e também recursos externos, estão interagindo entre si corretamente.

### Funcional

É um teste do ponto de vista do usuário, se uma determinada funcionalidade está executando corretamente, produzindo o resultado ou comportamento desejado pelo usuário.

## Beneficios

* Detectar facilmente se mudanças violaram as regras
* É uma forma de documentação (comportamento e entradas/saídas esperadas)
* Redução de custos em manutenções, especialmente em fases avançadas
* Melhora design da solução, pois a aplicação testável precisa ser bem delineada

## TDD - Test Driven Development

É um **método de desenvolver software**. Consiste em um desenvolvimento guiado pelos testes.

Princípios / vantagens:

* Foco nos requisitos
* Tende a melhorar o design do código, pois o código deverá ser testável
* Incrementos no projeto têm menos chance de quebrar a aplicação

Processo básico:

1. Escreva o teste como esperado (naturalmente que ele ainda estará falhando)
2. Implemente o código necessário para que o teste passe
3. Refatore o código conforme necessidade

## Boas práticas e padrões

### Nomenclatura de um teste

* <AÇÃO> should <EFEITO> [when <CENÁRIO>]

### Padrão AAA

* **Arrange**: instancie os objetos necessários
* **Act**: execute as ações necessárias
* **Assert**: declare o que deveria acontecer (resultado esperado)

### Princípio da inversão de dependência (SOLID)

* Se uma classe A depende de uma instância da classe B, não tem como testar a classe A isoladamente. Na verdade nem seria um teste unitário.
* A inversão de controle ajuda na testabilidade, e garante o isolamento da unidade a ser testada.

### Independência / isolamento

* Um teste não pode depender de outros testes, nem da ordem de execução

### Cenário único

* O teste deve ter uma lógica simples, linear
* O teste deve testar apenas um cenário
* Não use condicionais e loops

### Previsibilidade

* O resultado de um teste deve ser sempre o mesmo para os mesmos dados
* Não faça o resultado depender de coisas que variam, tais como timestamp atual e valores aleatórios.

## 

## JUnit

### Visão geral

* <https://junit.org/junit5>
* O primeiro passo é criar uma classe de testes
* A classe pode conter um ou mais métodos com a annotation @Test
* Um método @Test deve ser void
* O objetivo é que todos métodos @Test passem sem falhas
* O que vai definir se um método @Test passa ou não são as “assertions” deste método
* Se um ou mais falhas ocorrerem, estas são mostradas depois da execução do teste

# 

# Exercício: JUnit vanilla

**Solução:** <https://youtu.be/EsfLKHOy_rg>

|  | Um financiamento possui três dados:   * **totalAmont**: valor total de dinheiro financiado * **income**: renda mensal do cliente que está financiando * **months**: número de meses do financiamento   E dois métodos:   * **entry**: entrada do financiamento, igual a 20% do valor total * **quota**: prestação mensal do financiamento (sem juros) |
| --- | --- |

Há ainda uma regra: o valor da prestação não pode ser maior que metade da renda mensal do cliente. A seguir alguns exemplos de financiamentos para ajudar a entender a regra:

| **Exemplo 1**: **{ totalAmount: 100000, income: 2000, months: 20 }**  Este exemplo é **INVÁLIDO** porque com esses dados a entrada seria 20000 e a prestação seria 4000. Porém a prestação não pode passar de 1000, que é a metade da renda do cliente.  **Exemplo 2**: **{ totalAmount: 100000, income: 2000, months: 80 }**  Já este exemplo é **VÁLIDO** porque a entrada seria 20000 e a prestação seria 1000. Neste caso, a prestação é menor ou igual a metade da renda do cliente, satisfazendo a regra. |
| --- |

Projeto começado: <https://github.com/acenelio/exercicio-testes-java>

Você deve implementar os seguintes testes para validar esta classe (total = 10 testes):

**Construtor**

* Deve criar o objeto com os dados corretos quando os dados forem válidos
* Deve lançar IllegalArgumentException quando os dados não forem válidos

**setTotalAmount**

* Deve atualizar o valor quando os dados forem válidos
* Deve lançar IllegalArgumentException quando os dados não forem válidos

**setIncome**

* Deve atualizar o valor quando os dados forem válidos
* Deve lançar IllegalArgumentException quando os dados não forem válidos

**setMonths**

* Deve atualizar o valor quando os dados forem válidos
* Deve lançar IllegalArgumentException quando os dados não forem válidos

**entry**

* Deve calcular corretamente o valor da entrada

**quota**

* Deve calcular corretamente o valor da prestação

# Annotations usadas nas classes de teste

| @SpringBootTest | Carrega o contexto da aplicação (teste de integração) |
| --- | --- |
| @SpringBootTest  @AutoConfigureMockMvc | Carrega o contexto da aplicação (teste de integração & web)  Trata as requisições sem subir o servidor |
| @WebMvcTest(Classe.class) | Carrega o contexto, porém somente da camada web (teste de unidade: controlador) |
| @ExtendWith(SpringExtension.class) | Não carrega o contexto, mas permite usar os recursos do Spring com JUnit (teste de unidade: service/component) |
| @DataJpaTest | Carrega somente os componentes relacionados ao Spring Data JPA. Cada teste é transacional e dá rollback ao final. (teste de unidade: repository) |

# Fixtures

É uma forma de organizar melhor o código dos testes e evitar repetições.

| **JUnit 5** | **JUnit 4** | **Objetivo** |
| --- | --- | --- |
| @BeforeAll | @BeforeClass | Preparação antes de todos testes da classe (método estático) |
| @AfterAll | @AfterClass | Preparação depois de todos testes da classe (método estático) |
| @BeforeEach | @Before | Preparação antes de cada teste da classe |
| @AfterEach | @After | Preparação depois de cada teste da classe |

# Mockito vs @MockBean

<https://stackoverflow.com/questions/44200720/difference-between-mock-mockbean-and-mockito-mock>

| @Mock  private MyComp myComp;  ou  myComp = Mockito.mock(MyComp.class); | Usar quando a classe de teste não carrega o contexto da aplicação. É mais rápido e enxuto.  @ExtendWith |
| --- | --- |
| @MockBean  private MyComp myComp; | Usar quando a classe de teste carrega o contexto da aplicação e precisa mockar algum bean do sistema.  @WebMvcTest  @SpringBootTest |

# Exercícios: testes de repository

**Solução:** <https://youtu.be/qm3K1dkzJBM>

Favor implementar os seguintes testes em ProductRepositoryTests:

* findById deveria
  + retornar um Optional<Product> não vazio quando o id existir
  + retornar um Optional<Product> vazio quando o id não existir

# Exercícios: testes de unidade com Mockito

**Solução:** <https://youtu.be/KvXL5HgX5Jg>

Favor implementar os seguintes testes em ProductServiceTests:

* findById deveria
  + retornar um ProductDTO quando o id existir
  + lançar ResourceNotFoundException quando o id não existir
* update deveria *(dica: você vai ter que simular o comportamento do getOne)*
  + retornar um ProductDTO quando o id existir
  + lançar uma ResourceNotFoundException quando o id não existir

# Exercícios: testes da camada web com MockMvc

**Solução:** <https://youtu.be/Vmt-LEShc7Y>

Favor implementar os seguintes testes em ProductResourceTests:

* insert deveria
  + retornar “created” (código 201), bem com um ProductDTO
* delete deveria
  + retornar “no content” (código 204) quando o id existir
  + retornar “not found” (código 404) quando o id não existir

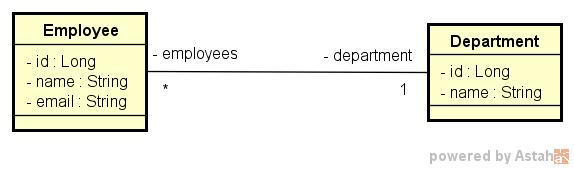
# 

# Desafio TDD resolvido

Implemente as funcionalidades necessárias para que os testes do projeto abaixo passem:

<https://github.com/devsuperior/bds01>

Este é um sistema de funcionários e departamentos com uma relação N-1 entre eles:



# Desafio TDD para entregar

**TAREFA: TDD Event-City**

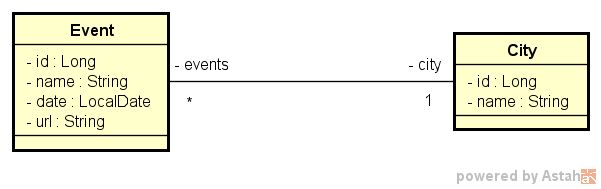
Implemente as funcionalidades necessárias para que os testes do projeto abaixo passem:

<https://github.com/devsuperior/bds02>

Collection do Postman:

<https://www.getpostman.com/collections/c347ea3428d6b199b391>

Este é um sistema de eventos e cidades com uma relação N-1 entre eles:



Mínimo para aprovação: 5/7