



Relatório de Testes de Software

Exame

Marcos Vinicius
Murillo Godoi

**Ribeirão Preto
2021**

Relatório sobre Api Rest

Exame

Marcos Vinicius
Murillo Godoi

**Ribeirão Preto
2021**

RESUMO

Este documento tem como objetivo apresentar as características da Api Rest que criamos como objetivo do exame final. Para a criação da Api utilizamos o Spring Boot, e para a criação dos testes utilizamos o Framework Mockito.

SUMÁRIO

| | |
|--|----|
| 1 INTRODUÇÃO | 5 |
| 2 DESENVOLVIMENTO | 6 |
| 2.1 Model | 6 |
| 2.2 Rotas da Api..... | 7 |
| 2.3 Testes realizados na Api | 8 |
| 2.3.1Teste na rota de listagem de filmes..... | 8 |
| 2.3.2Teste na rota de cadastro de filmes | 9 |
| 2.3.3Teste na rota de atualização de filmes | 10 |
| 2.3.4 Teste na rota de listagem de filme por id..... | 11 |
| 2.3.5 Teste na rota de remoção de um filme | 11 |
| 2.4 Resultados dos Testes | 12 |
| 3 CONSIDERAÇÕES FINAIS..... | 13 |

1. INTRODUÇÃO

A Api Rest que criamos tem como o tema filmes, pois pensamos em um sistema de cadastramento de filmes de uma locadora, a partir da Api será possível cadastrar filmes, lista-los de uma forma geral ou a partir de um id, atualizar um filme, e também os salvar e excluí-los.

Os filmes terão como atributos id, nome, sinopse, ano de lançamento, classificação indicativa e gênero.

2. DESENVOLVIMENTO

2.1 Model

Essa é a classe model de filmes onde definimos qual será o nome de sua tabela no banco de dados e também quais serão os seus atributos que serão salvos.

```
@Entity
@Table(name="filmes")
@Builder
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Filme {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @NotNull
    private String nome;

    @NotNull
    private String sinopse;

    @NotNull
    private String ano;

    @NotNull
    private String classificacaoIndicativa;

    @NotNull
    private String genero;
}
```

Figura 1 – Classe Model

2.2 Rotas da Api

```
@CrossOrigin(origins="*")
@RestController
@RequestMapping(value="/filmes")
public class FilmeResource {

    @Autowired
    FilmeRepository filmeRepository;

    @GetMapping
    public List<Filme> listaFilmes(){
        return filmeRepository.findAll();
    }

    @PostMapping
    public Filme salvaFilme(@RequestBody @Valid Filme filme){ Replace this persistent entity with a simple
        return filmeRepository.save(filme);
    }

    @PutMapping("/{id}")
    public Filme atualizaFilme(@PathVariable(value="id") Long id, @RequestBody @Valid Filme filmeAtualizado){
        filmeAtualizado.setId(id);
        return filmeRepository.save(filmeAtualizado);
    }

    @GetMapping("/{id}")
    public Filme listaFilmeUnico(@PathVariable(value="id") Long id){
        return filmeRepository.findById(id);
    }

    @DeleteMapping("/{id}")
    public void deletaFilme(@PathVariable(value="id") Long id){
        filmeRepository.deleteById(id);
    }
}
```

Figura 2 – Rotas da Api

A base das rotas da nossa Api é “/filmes”, nós definimos cinco rotas, são elas:

1. A primeira rota da nossa Api é para listar todos os filmes para isso é preciso dar um GET na url base da nossa Api;
2. A segunda rota é para salvar um filme, para isso acontecer é preciso dar um POST em nossa url base passando como body um json contendo os atributos do filme a ser salvo;
3. A terceira rota é pra atualizar um filme, para isso é preciso dar um PUT em nossa url base passando o id do filme que será atualizado, e também mandar no body da request um objeto json contendo as informações que serão atualizadas no filme;
4. A quarta rota é para listar um filme por id, é preciso dar um GET na url base passando o id do filme que será listado;

5. A quinta rota é para deletar um filme por id, é preciso dar um DELETE na url base passando o id do filme que será deletado;

2.3 Testes Realizados na Api

Utilizamos o TDD para a criação de todos os testes feitos em nossa Api.

Nós mocamos dois filmes antes de serem realizados os testes

```
public class FilmeTest {  
    @Autowired  
    MockMvc mockMvc;  
    @Autowired  
    ObjectMapper objectMapper;  
  
    @MockBean  
    FilmeRepository filmeRepository;  
  
    Filme FILME_1 = new Filme(1L, "filme 1", "sinopse", "1992", "Livre", "Ação");  
    Filme FILME_2 = new Filme(2L, "filme 2", "sinopse", "1998", "+10", "Drama");  
}
```

Figura 3 – Filmes mocados

2.3.1 Teste na rota de listagem de filmes

Abaixo está o teste realizado na rota de listagem de filmes.

```
@Test  
Run Test | Debug Test  
public void DeveListarFilmes() throws Exception {  
    List<Filme> filmes = new ArrayList<>(Arrays.asList(FILME_1, FILME_2));  
  
    Mockito.when(filmeRepository.findAll()).thenReturn(filmes);  
    mockMvc.perform(MockMvcRequestBuilders  
        .get("/filmes")  
        .contentType(MediaType.APPLICATION_JSON))  
        .andExpect(MockMvcResultMatchers.status().isOk())  
        .andExpect(MockMvcResultMatchers.jsonPath("$", Matchers.hasSize(2)))  
        .andExpect(MockMvcResultMatchers.jsonPath("$[1].nome", Matchers.is("filme 2")));  
}
```

Figura 4 – Teste na rota de listagem de filmes

2.3.2 Teste na rota de cadastro de filmes

Abaixo está o teste realizado na rota de cadastro de filmes.

```
@Test
Run Test | Debug Test
public void DeveCriarFilmes() throws Exception {
    Filme filme = Filme.builder()
        .nome("filme 1")
        .sinopse("sinopse")
        .ano("1992")
        .classificacaoIndicativa("Livre")
        .genero("Ação")
        .build();

    Mockito.when(filmeRepository.save(filme)).thenReturn(filme);

    MockHttpServletRequestBuilder mockRequest = MockMvcRequestBuilders.post("/filmes")
        .contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.APPLICATION_JSON)
        .content(this.objectMapper.writeValueAsString(filme));

    mockMvc.perform(mockRequest)
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.jsonPath("$", Matchers.notNullValue()))
        .andExpect(MockMvcResultMatchers.jsonPath("$.nome", Matchers.is("filme 1")));
}
```

Figura 5 – Teste na rota de cadastro de filmes

2.3.3 Teste na rota de atualização do filme

Abaixo está o teste realizado na rota de atualização de filme.

```
@Test
Run Test | Debug Test
public void DeveAtualizarFilme() throws Exception {
    Filme atualizacaoFilme = Filme.builder()
        .nome("filme atualizado")
        .sinopse("sinopse")
        .ano("1992")
        .classificacaoIndicativa("Livre")
        .genero("Ação")
        .build();

    Filme filmeAtualizado = Filme.builder()
        .id(1L)
        .nome("filme atualizado")
        .sinopse("sinopse")
        .ano("1992")
        .classificacaoIndicativa("Livre")
        .genero("Ação")
        .build();

    Mockito.when(filmeRepository.findById(FILME_1.getId())).thenReturn(FILME_1);
    Mockito.when(filmeRepository.save(filmeAtualizado)).thenReturn(filmeAtualizado);

    MockHttpServletRequestBuilder mockRequest = MockMvcRequestBuilders.put("/filmes/" + FILME_1.getId())
        .contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.APPLICATION_JSON)
        .content(this.objectMapper.writeValueAsString(atualizacaoFilme));

    mockMvc.perform(mockRequest)
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.jsonPath("$", Matchers.notNullValue()))
        .andExpect(MockMvcResultMatchers.jsonPath("$.nome", Matchers.is("filme atualizado")));
}
```

Figura 6 – Teste na rota de atualização do filme

2.3.4 Teste na rota de listagem de filme por id

Abaixo está o teste realizado na rota de listagem de filme por id.

```
@Test
Run Test | Debug Test
public void DeveListarFilmeUnico() throws Exception {

    Mockito.when(filmeRepository.findById(FILME_1.getId())).thenReturn(FILME_1);

    mockMvc.perform(MockMvcRequestBuilders
        .get("/filmes/" + FILME_1.getId())
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.jsonPath("$", Matchers.notNullValue()))
        .andExpect(MockMvcResultMatchers.jsonPath("$.nome", Matchers.is("filme 1")));
}
```

Figura 7 – Teste na rota de listagem de filme por id

2.3.5 Teste na rota de remoção de um filme

Abaixo está o teste realizado na rota de remoção de um filme.

```
@Test
Run Test | Debug Test
public void DeveRemoverFilmeUnico() throws Exception {
    Mockito.when(filmeRepository.findById(FILME_1.getId())).thenReturn(FILME_1);

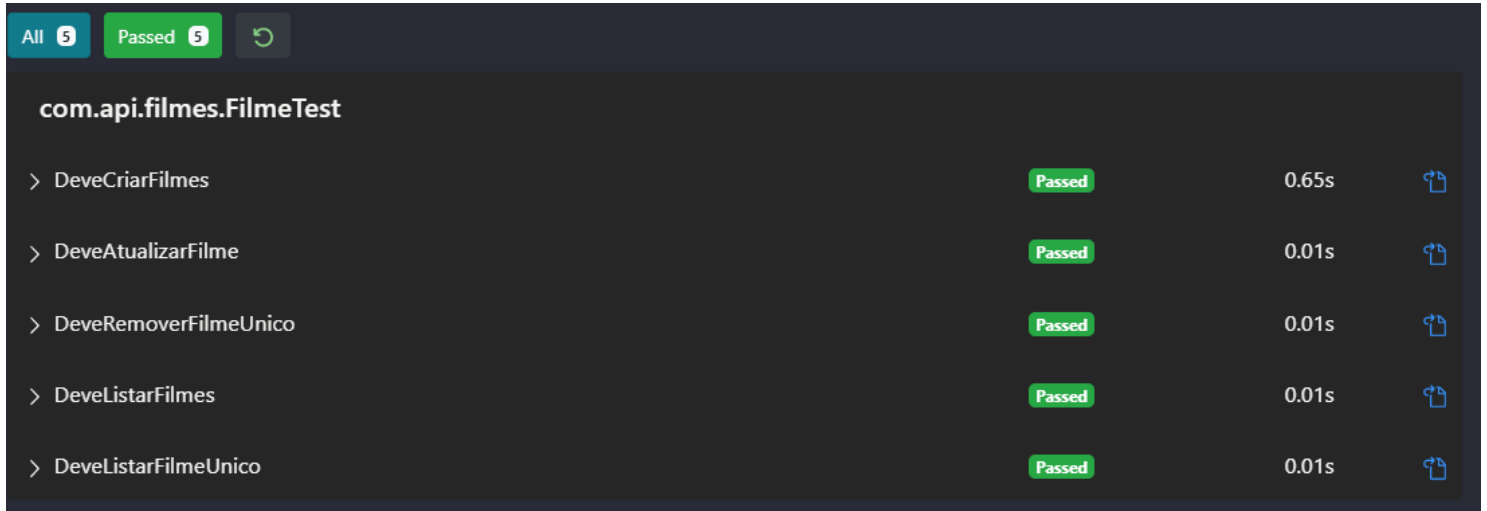
    mockMvc.perform(MockMvcRequestBuilders
        .delete("/filmes/" + FILME_1.getId())
        .andExpect(MockMvcResultMatchers.status().isOk()));
}
```

Figura 8 – Teste na rota de remoção de um filme

2.4 RESULTADOS DOS TESTES

Durante o desenvolvimento do projeto, ao todo, foram criados cinco testes e todos estes passaram sem qualquer problema.

Abaixo está a imagem que demonstra o sucesso de todos os casos de testes realizados no projeto.



The screenshot shows a test runner interface with a dark theme. At the top, there are two tabs: 'All' with a count of 5 and 'Passed' with a count of 5. Below the tabs, the package name 'com.api.filmes.FilmeTest' is displayed. A list of five tests follows, each with a 'Passed' status, a duration, and a copy icon.

| Test Name | Status | Duration | Action |
|-------------------------|--------|----------|--------|
| > DeveCriarFilmes | Passed | 0.65s | Copy |
| > DeveAtualizarFilme | Passed | 0.01s | Copy |
| > DeveRemoverFilmeUnico | Passed | 0.01s | Copy |
| > DeveListarFilmes | Passed | 0.01s | Copy |
| > DeveListarFilmeUnico | Passed | 0.01s | Copy |

Figura 9 – Resultados dos testes

3. CONSIDERAÇÕES FINAIS

Durante e após o desenvolvimento deste exame, ficou ainda mais evidente para nós a importância do desenvolvimento orientado a testes (TDD). Durante a aplicação destes conceitos em nosso projeto, percebemos que foi possível eliminar muitos códigos desnecessários e com a utilização de baby steps o nosso código ficou mais limpo e simples, e também percebemos a simplicidade do Spring Boot para construção e testes de API.