




# Testes automatizados

■ Course	 <a href="#">Spring Boot 3</a>
☀ Confidence	Not Confident
📅 Next Review	@11 de agosto de 2023
🕒 Last Edited	@14 de agosto de 2023 10:04
📌 Curso	Spring Boot 3: Documente teste e prepare uma API para o deploy

## Tópicos

- Testes com Spring Boot
- Configurando o banco de testes
- Para saber mais: testes com in-memory database
- Testando o repository
- Testando erro 400
- Testando código 200
- Códigos testáveis
- Faça como eu fiz: testes da classe MedicoController
- O que aprendemos?

---

## Testes automatizados com Spring Boot

O que vamos testar?

*Controller* → *API*

*Repository* → *Queries*

---

## Configurando o banco de testes

- A anotação `@DataJpaTest` é utilizada para testar uma interface Repository
- A anotação `@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)`

- Com isso, estamos indicando que queremos fazer uma configuração do banco de dados de teste e pedindo para não substituir as configurações do banco de dados pelas do banco em memória.
- A anotação `@ActiveProfiles("test")` Indica que o arquivo de configuração que o spring deve carregar seria o `application-test.properties`.

---

## Para saber mais: testes com in-memory database

Como citado no vídeo anterior, podemos realizar os testes de interfaces *repository* utilizando um banco de dados em memória, como o **H2**, ao invés de utilizar o mesmo banco de dados da aplicação.

Caso você queira utilizar essa estratégia de executar os testes com um banco de dados em memória, será necessário incluir o H2 no projeto, adicionando a seguinte dependência no arquivo `pom.xml`:

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

E também deve remover as anotações `@AutoConfigureTestDatabase` e `@ActiveProfiles` na classe de teste, deixando-a apenas com a anotação `@DataJpaTest`:

```
@DataJpaTest
class MedicoRepositoryTest {

    //resto do código permanece igual

}
```

Você também pode **apagar** o arquivo `application-test.properties`, pois o Spring Boot realiza as configurações de `url`, `username` e `password` do banco de dados H2 de maneira automática.

---

## Testando o repository

- A anotação `@DisplayName` é utilizada para descrever o teste de forma mais completa, pelo fato que essa descrição vai ser uma **String** exemplo:

```
@Test
@DisplayName("Deveria devolver null quando unico medico cadastrado nao esta disponivel na data")
void escolherMedicoAleatorioLivreNaDataCenario1()
```

---

## Códigos testáveis

Ao longo desta aula, aprendemos a escrever **testes automatizados** na aplicação utilizando o JUnit em conjunto com o Spring Boot.

Considerando o que foi ensinado, escolha as alternativas que indicam afirmativas **verdadeiras** em relação aos testes automatizados.

- Ao se utilizar o Swagger UI, torna-se desnecessário escrever testes automatizados no projeto.
  - Embora o Swagger UI permita a execução de testes na API, ele não invalida a necessidade de se escrever testes automatizados no projeto.
- Alguns componentes do Spring não precisam ser testados.
  - Alguns componentes, como *interfaces repository*, que não possuem métodos de consultas, não precisam de testes automatizados.
- Podemos mesclar testes de unidade com testes de integração.
  - Para alguns componentes, como classes controller, podemos escrever testes de unidade; já para outros, como as *interfaces repository*, testes de integração são os mais recomendados.
- Devemos testar todas as classes do projeto.
  - Nem todas as classes têm códigos de regras de negócio, validações, algoritmos ou integrações.

---

## Faça como eu fiz: testes da classe MedicoController

Agora é com você! Faça o mesmo procedimento que eu fiz na aula, porém, agora criando os testes automatizados para o método `cadastrar` da classe `MedicoController`.

---

## O que aprendemos?

Nesta aula, você aprendeu como:

- Escrever **testes automatizados** em uma aplicação com Spring Boot;

- Escrever testes automatizados de uma interface *Repository*, seguindo a estratégia de usar o mesmo banco de dados que a aplicação utiliza;
  - Sobrescrever propriedades do arquivo `application.properties`, criando outro arquivo chamado `application-test.properties` que seja carregado apenas ao executar os testes, utilizando para isso a anotação `@ActiveProfiles`;
  - Escrever testes automatizados de uma classe Controller, utilizando a classe `MockMvc` para simular requisições na API;
  - Testar cenários de erro 400 e código 200 no teste de uma classe controller.
-