Teste de integração com o Spring Boot MockMVC

Certified Developer
The Ultimate Tech Degree

DigitalHouse >

Escrever testes de integração com MockMVC

Devemos começar estabelecendo o contexto inicial da classe de teste, levantando a aplicação conforme ela é executada no contexto de desenvolvimento e injetando todas as dependências necessárias.

```
@SpringBootTest

@AutoConfigureMockMvc
public class HelloWorldIntegrationTest {

    @Autowired
    private MockMvc mockMvc;
```

Carrega o contexto completo da aplicação Spring.

Permite a injeção de um objeto MockMVC totalmente configurado.

Injeta a dependência necessária.

Testar um método GET e verificar o conteúdo da resposta

Faremos uma solicitação (request) para a URL: http://localhost:8080/sayHello e a saída esperada é:

```
{
    "id": 1,
    "message": "Hello World!"
}
```

perform() executará o método de solicitação GET, que retorna um ResultActions. Neste objeto podemos obter o response, content, HTTP status e Header.

```
@Test
public void testHelloWorldOutput() throws Exception {
   MvcResult mvcResult =
            this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHello"))
           .andDo(print()).andExpect(status().isOk())

.andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello World!"))
           .andReturn();

   Assertions.assertEquals("application/json",
mvcResult.getResponse().getContentType());
}
```

andDo(print()) imprime o request e response no console. Útil para obter detalhes em caso de erro.

```
orldOutput() throws Exception {
    MvcResult m    sult =
            this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHello"))
            .andDo(print()).andExpect(status().isOk())

.andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello World!"))
            .andReturn();

    Assertions.assertEquals("application/json",
mvcResult.getResponse().getContentType());
}
```

andExpect(MockMvcResultMatchers.status().isOk()) verifica se a resposta (response) é HTTP status OK (200).

```
@T
public void testHelloWo          t() throws Exception {
   MvcResult mvcResult =
            this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHello"))
           .andDo(print()).andExpect(status().isOk())

.andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello World!"))
           .andReturn();

   Assertions.assertEquals("application/json",
mvcResult.getResponse().getContentType());
}
```


andExpect(MockMvcResultMatchers.jsonPath(“$.message”).value(“Hello World!!!”)) verifica se o conteúdo corresponde à saída esperada. o jsonPath extrai parte dessa resposta para fornecer o valor a ser verificado.
rows Exception {
MvcRequestBuilders.get("/sayHello"))
tus().isOk())
.andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello World!"))
.andReturn();
andReturn() retorna o objeto MvcResult completo caso seja necessário verificar algo que não tenha sido verificado nos métodos anteriores.
Assertions.assertEquals("
mvcResult.getResponse().getCo
}

Testar um método GET com um PathVariable

Faremos uma solicitação (request) para a URL: http://localhost:8080/sayHello/George e a saída esperada é:

```
{
    "id": 1,
    "message": "Hello George!"
}
```

MockMvcRequestBuilders.get("/sayHello/{name}", "George") ele executará o método GET com sua PathVariable no caminho da URL.

```
@Test
public void testHelloGeorgeOutput() throws Exception {
   this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHello/{name}", "George"))
           .andDo(print()).andExpect(status().isOk())
           .andExpect(content().contentType("application/json"))
           .andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello
George!"));
}
```

Testar um método GET com QueryParam

Faremos uma solicitação (request) para a URL: http://localhost:8080/sayHelloWithParam?name=George e a saída esperada é:

```
{
    "id": 1,
    "message": "Hello George!"
}
```

```
@Test
public void testHelloWithParamGeorgeOutput() throws Exception {

this.mockMvc.perform(MockMvcRequestBuilders
")
                .param("name", "George"))
            .andDo(print()).andExpect(status().isOk())
            .andExpect(content().contentType("application/json"))

.andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello
George!"));
}
```

param("name", "George") adiciona um Query Parameter na solicitação GET.

Testar um método POST e verificar o conteúdo da resposta

Faremos uma solicitação (request) para a URL: http://localhost:8080/sayHelloPost e o body de entrada é:

```
{
    "name”: “George”
}
```

E a saída esperada é:

```
{
    "id": 1,
    "message": "Hello George!"
}
```

```
@Test
public void testHelloPostGeorgeOutput() throws
    NameDTO payloadDTO = new NameDTO("George");
    ObjectWriter writer = new ObjectMapper().
            configure(SerializationFeature.WRAP_ROOT_VALUE, false).
            writer().withDefaultPrettyPrinter();
    String payloadJson = writer.writeValueAsString(
            mockMvc.perform(MockMvcRequestBuilders.post("/helloPost")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(payloadJson))
            .andDo(print()).andExpect(status().isOk())
            .andExpect(content().contentType("application/json"))
            .andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello
George!"));
}
```

Incorpora o ObjectMapper, o qual é usado para converter um objeto do tipo DTO em uma String com sua representação em JSON.

contentType(MediaType.APPLICATION.JSON) define o formato de payload de entrada.

content(payloadJson) adiciona o payload no formato JSON no POST request.

Testar um método POST e verificar o conteúdo completo da resposta

Faremos uma solicitação (request) para a URL: http://localhost:8080/sayHelloPost e o body de entrada é:

```
{
    "name”: “George”
}
```

E a saída esperada é:

```
{
    "id": 1,
    "message": "Hello George!"
}
```

```java
@Test
public void testHelloPostGeorgeOutput() throws Exception {
   NameDTO payloadDTO = new NameDTO("George");
   HelloDTO responseDTO = new HelloDTO(1, "Hello George!");

   ObjectWriter writer = new ObjectMapper()
           .configure(SerializationFeature.WRAP_ROOT_VALUE, false)
           .writer();

   String payloadJson = writer.writeValueAsString(payloadDTO);
   String responseJson = writer.writeValueAsString(responseDTO);

   MvcResult response = this.mockMvc.perform(MockMvcRequestBuilders.post("/sayHelloPost")
               .contentType(MediaType.APPLICATION_JSON)
               .content(payloadJson))
           .andDo(print()).andExpect(status().isOk())
           .andExpect(content().contentType("application/json"))
           .andReturn();

   Assertions.assertEquals(responseJson, response.getResponse().getContentAsString());
}
```

Anotações para testes de integração

@WebMvcTest: utilizando para teste MockMVC. Desabilita a configuração automática e permite uma configuração determinada, por exemplo, do Spring Security.

@MockBean: Permite a simulação de Beans.

@InjectMocks: Permite a injeção de Beans.

@ExtendWith: Normalmente, a extensão SpringExtension.class é fornecida, e inicializa o contexto de teste do Spring.

@ContextConfiguration: Permite que você carregue uma classe de configuração personalizada.

@WebAppConfiguration: Permite carregar o contexto web da aplicação.

DigitalHouse>