Pruebas de integración con Spring Boot MockMVC

DigitalHouse>

Certified Tech Developer
The Ultimate Degree

Escribir tests de integración con MockMVC

Se comienza por establecer el contexto inicial de la clase de testeo, levantando la aplicación tal cual se ejecuta en el contexto de desarrollo e inyectando todas las dependencias que se requieran.

```
@SpringBootTest
@AutoConfigureMockMvc
public class HelloWorldIntegrationTest {

    @Autowired
    private MockMvc mockMvc;
```

Levanta el contexto completo de la aplicación Spring.

Permite la inyección de un objeto MockMVC completamente configurado.

Inyecta la dependencia requerida.

Testear un método GET y verificar el contenido de la respuesta

Se hará un pedido (request) a la URL: http://localhost:8080/sayHello y la salida esperada es:

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

perform() va a efectuar el método GET request, que devuelve ResultActions. A este objeto se le podrán efectuar las assertions sobre la response, content, HTTP status o header.

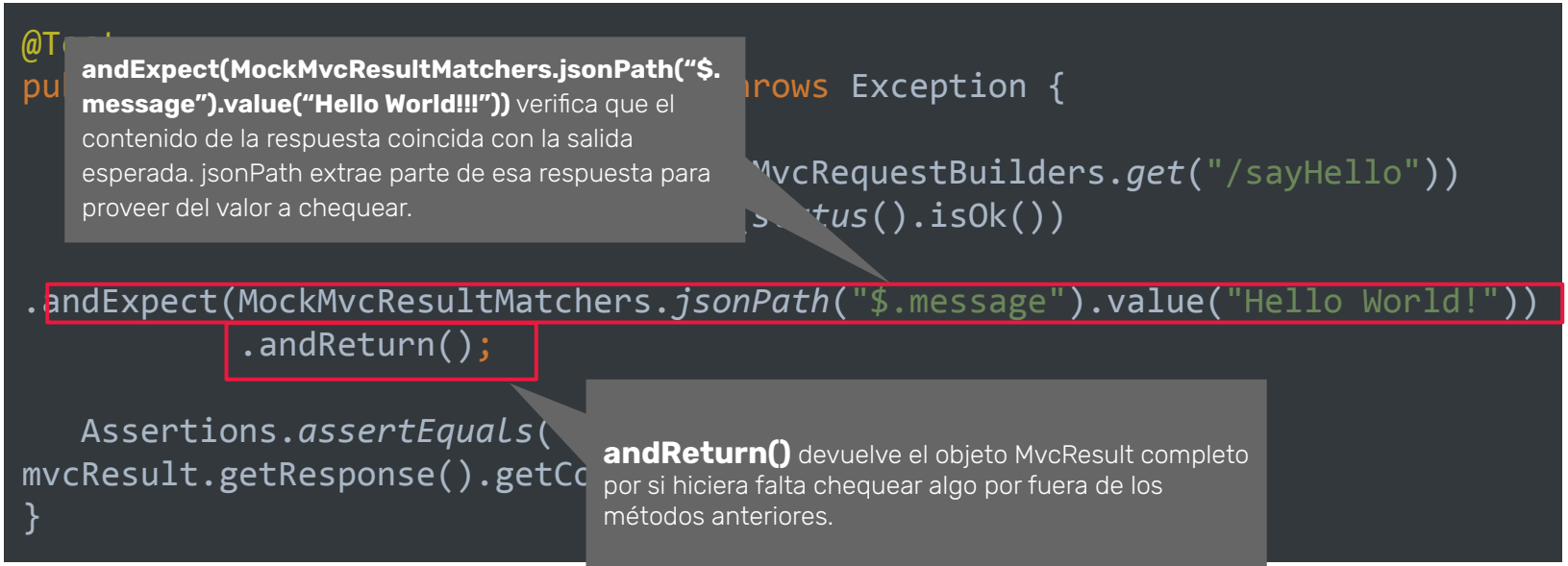
```
@Test
public void testHelloWorldOutput() throws Exception {
   MvcResult mvcResult =
            this.mockMvc.perform(MockMvcRequestBuilders.get("/sayHello"))
           .andDo(print()).andExpect(status().isOk())

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

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

andDo(print()) imprime request y response por consola. Útil para obtener detalles en caso de error.

```
elloWorldOutput() 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.status().isOk()) verifica que la respuesta (response) sea HTTP status OK (200).

```
@Test
public void testHelloOutput() 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());
}
```


@T
pu
rows Exception {
MvcRequestBuilders.get("/sayHello"))
tus().isOk())
.andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello World!"))
.andReturn();
Assertions.assertEquals(
mvcResult.getResponse().getCo
}
andExpect(MockMvcResultMatchers.jsonPath("$.message").value("Hello World!")) verifica que el contenido de la respuesta coincida con la salida esperada. jsonPath extrae parte de esa respuesta para proveer del valor a chequear.
andReturn() devuelve el objeto MvcResult completo por si hiciera falta chequear algo por fuera de los métodos anteriores.

Testear un método GET con un PathVariable

Se hará un pedido (request) a la URL: http://localhost:8080/sayHello/George y la salida esperada es:

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

MockMvcRequestBuilders.get("/sayHello/{name}", "George") va a efectuar el método GET request con su PathVariable en el path de la 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 un método GET con QueryParam

Se hará un pedido (request) a la URL: http://localhost:8080/sayHelloWithParam?name=George y la salida esperada es:

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

```
@Test
public void testHelloWithParamGeorgeOutput(

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") va a agregar el parámetro Query en el request GET.

Testear un método POST y verificar el contenido de la respuesta

Se hará un pedido (request) a la URL: http://localhost:8080/sayHelloPost y el body de entrada es:

```
{
    "name”: “George”
}
```

Y la salida de esperada es:

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

```
@Test
public void testHelloPostGeorgeOutput() th
   NameDTO payloadDTO = new NameDTO("Georg

   ObjectWriter writer = new ObjectMapper().
           configure(SerializationFeature
                   withDefaultPrettyPrir
                   n = writer.writeValue

                   form(MockMvcRequestBuilders.post("/sayHelloPost")
                   ntentType(MediaType.APPLICATION_JSON)
                   .content(payloadJson))
           .andDo(print()).andExpect(status().isOk())
           .andExpect(content().contentType("application/json"))

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

Se incorpora el ObjectMapper, que se utiliza para convertir un objeto de tipo DTO en un String con su representación en JSON.

contentType(MediaType.APPLICATION.JSON) especifica el formato del payload de entrada.

content(payloadJson) agrega el payload en formato Json al POST request.

Testear un método POST y verificar el contenido completo de la respuesta

Se hará un pedido (request) a la URL: http://localhost:8080/sayHelloPost y el body de entrada es:

```
{
    "name”: “George”
}
```

Y la salida de esperada es:

```
{
    "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());
}
```

Anotaciones para tests de integración

@WebMvcTest: Se utiliza para pruebas MockMVC. Deshabilita la autoconfiguración y permite una configuración determinada, por ejemplo, de Spring Security.
@MockBean: Permite la simulación de Beans.
@InjectMocks: Permite la inyección de Beans.
@ExtendWith: Usualmente se proporciona la extensión SpringExtension.class, inicializa el contexto de testeo Spring.
@ContextConfiguration: Permite cargar una clase de configuración custom.
@WebAppConfiguration: Permite cargar el contexto web de la aplicación.

DigitalHouse>