Prueba de controladores REST con Spring Test

En la guía anterior , `Books` se creó REST Controller for . Es una buena práctica mantener los casos de prueba al día con el desarrollo. Aquí exploremos cómo probar los métodos REST de la unidad.
Código de referencia: https://github.com/GlueCoders/springboot-guide/releases/tag/rest-with-mvc

Dependencia de POM

Incluya la siguiente dependencia en el `pom.xml` archivo.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <version>${spring.version}</version>
</dependency>
```

Esto incluirá bibliotecas para `Spring Test` , `Mockito` , `Hamcrest` y otras bibliotecas de terceros útiles necesarios durante las pruebas unitarias.

@WebMvcTest

Para probar Spring MVC Controllers `@WebMvcTest` se usa la anotación. Esta anotación busca todos los componentes relacionados con MVC y no incluirá `@Component` clases regulares . Esto a menudo se usa para probar una clase de controlador a la vez y se combina con el marco Mockito para burlarse de las dependencias. Spring tiene una `@MockBean` anotación que juega bien con la biblioteca Mockito.

Definición de clase de prueba

```
@RunWith(SpringRunner.class)
@WebMvcTest(Books.class)
public class BooksTest {
```

```
    @Autowired
    private MockMvc mvc;

    @MockBean
    private BookService bookService;
```

`@RunWith` define la clase de corredor que se usará para ejecutar casos de prueba, `SpringRunner` es la opción de facto ya que `Spring` se usa prácticamente para todo en la aplicación.
`@WebMvcTest` toma la clase de controlador que está bajo prueba. Esto iniciará el contexto de la aplicación web, no se utilizará el contenedor de servlet incorporado y esto sigue haciendo que las pruebas sean livianas.
`MockMvc` es la clase auxiliar que proporciona sintaxis para llamar a las clases de controlador como solicitudes HTTP. También define métodos para expectativas sobre la respuesta HTTP.
`@MockBean` crea un simulacro Mockito del `BookService` cual es una dependencia para el `Books` controlador. Al tener simulacros, podemos controlar el comportamiento de las dependencias sin llamar al método real y realmente solo centrarnos en las pruebas unitarias de la clase de controlador. Para controlar el comportamiento de `bookService` , un `Behavior` La clase anidada se ha definido en estos casos de prueba que se mostrarán más adelante.

Ayudante de MockMvc

Los siguientes son los usos de la `MockMvc` instancia `mvc` para realizar pruebas unitarias en los métodos REST.

OBTENER Método

El siguiente caso de prueba corresponde al `getAllBooks()` método en el `Books` controlador.

```
@Test
    public void getAllBooks_NoBooks() throws Exception {
        Behavior.set(bookService).hasNoBooks();
        mvc
                .perform(get("/books"))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8))
                .andExpect(content().json("[]"));
```

```
        verify(bookService, times(1)).getAllBooks();
    }
```

`.perform(..` es análogo a hacer una solicitud HTTP al controlador REST.
`get("/books")` define que se debe realizar una solicitud HTTP GET en la ruta '/ books'.
`.andExpect(..` se puede usar para establecer expectativas sobre la respuesta HTTP recibida de la clase de controlador, como por ejemplo `status().isOk()` , el código de respuesta HTTP debe serlo `200` .
`content().json([])` significa que el cuerpo de la respuesta debe coincidir con el contenido json que se proporciona en el `json()` método.
`content().contentType(...` significa que el `Content-Type` encabezado debe coincidir con el valor dado en el método.
`Behavior` es una clase personalizada que configura métodos Mockito `bookService` .

Método POST

El siguiente caso de prueba corresponde al `addBook()` método en el `Books` controlador.

```
@Test
    public void addBook_Positive() throws Exception {
        Behavior.set(bookService).returnSame();
        String bookContent = mapper.writeValueAsString(effectiveJavaBook);
        mvc
                .perform(post("/books")
                        .content(bookContent)
                        .contentType(MediaType.APPLICATION_JSON_UTF8))
                .andExpect(status().isCreated())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8))
                .andExpect(content().json(bookContent));
        verify(bookService, times(1)).addBook(effectiveJavaBook);
    }
```

`post(..` define hacer una solicitud HTTP POST junto con el contenido y el tipo de contenido definido en los métodos.
`mapper` Esta es la instancia `Jackson ObjectMapper` que permite la serialización y deserialización de POJOs hacia y desde JSON.

Método DELETE

El siguiente caso de prueba corresponde al `deleteBook()` método en el `Books` controlador.

```
@Test
    public void deleteBook() throws Exception {
        mvc
                .perform(delete("/books/1234567899"))
                .andExpect(status().isOk());
        verify(bookService, times(1)).deleteBook(anyLong());
    }
```

`delete("/books...")` define hacer una solicitud HTTP DELETE.

Uso de Mockito

Exploremos también algunos de los métodos Mockito que se están utilizando en esta clase.

verificar()

`verify` El método se utiliza para verificar la cantidad de veces que se ha llamado a un método simulado. Esto es especialmente útil para probar declaraciones condicionales, ya que si el caso de prueba no espera que el flujo de código ingrese una cierta declaración de condición, entonces los métodos dentro de ese bloque no deberían llamarse ni una sola vez.
Su firma es `verify(T mock, VerificationMode mode)` donde `T` corresponde al bean simulado y se `mode` refiere al número de veces que se deben llamar los métodos. También se puede usar para hacer coincidir los parámetros recibidos por el método simulado como se muestra a continuación

```
verify(bookService, times(1)).getBookByISBN(effectiveJavaBook.getIsbnCode());
```

Esto significa que el método `getBookByISBN` de bean solo `bookService` debe llamarse una vez durante la ejecución del caso de prueba. Además, el parámetro recibido por el método se verifica dando el argumento exacto como aquí `effectiveJavaBook.getIsbnCode()` .

cuando () doReturn () thenReturn ()

Estas construcciones se utilizan para establecer expectativas en frijoles simulados. Veamos primero un ejemplo:

```
when(bookService.getBookByISBN(book.getIsbnCode())).thenReturn(book);
when(bookService.addBook(book)).thenReturn(book);
when(bookService.getAllBooks()).thenReturn(Collections.emptyList());
when(bookService.addBook(any())).thenAnswer(invocationOnMock -> invocationOnMock.getArguments()[0]);
```

El primer ejemplo define que si `getBookByISBN` se invoca con algunos `isbnCode` , debería devolver el mismo libro, ya `isbnCode` que también se obtiene de la misma instancia de `book` .
El segundo ejemplo define que `addBook` debería devolver la `book` instancia si se llama con la misma `book` instancia.
El tercer ejemplo define que `getAllBooks` debería devolverse `emptyList` si se invoca.
El cuarto ejemplo define que `addBook` debe devolver la misma instancia que recibe como parámetro. Tenga en cuenta que aquí no hay una `book` instancia predefinida , por lo que esto funcionará para cualquier argumento pasado `addBook` , sin embargo, en el segundo ejemplo, solo funcionará para esa instancia de la `book` que se ha utilizado en la `when(..` cláusula.

Se `BooksTest` puede hacer referencia a toda la clase en el `src/tests/java` directorio bajo `org.gluecoders.library.rest` paquete.

Anterior TOC Siguiente