

PRUEBAS EN LOS CONTROLADORES DE SPRING

DESCRIPCIÓN

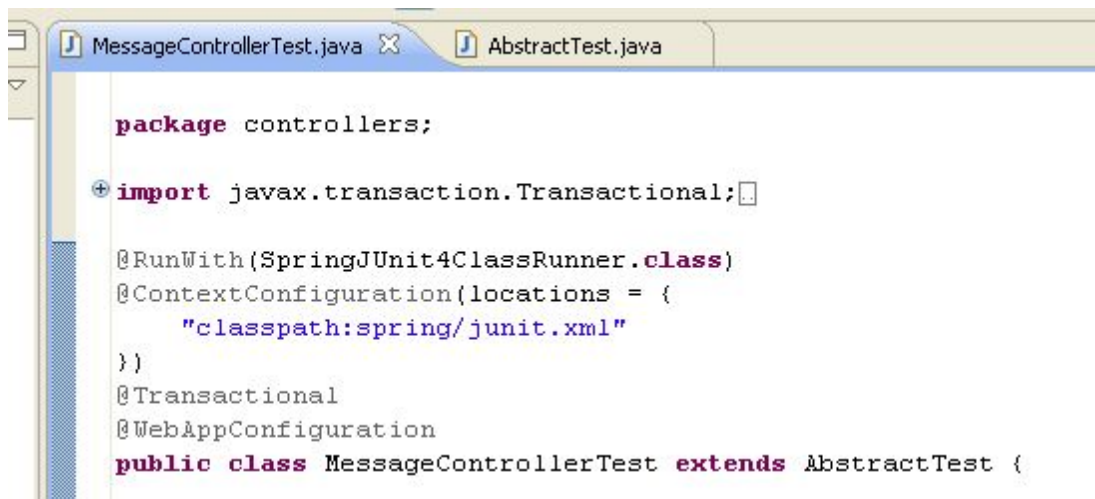
Prueba del controlador Message usando la librería Mockito en Spring. Para ello será necesario previamente incluirla en el archivo pom.xml de nuestro proyecto.

CREACIÓN DE LA CLASE TEST

La clase test de los controladores se incluyen en el mismo directorio que los test de servicios, *src/test/java*.

ANOTACIONES

Los anotaciones a utilizar siguen la misma estructura que las anotaciones que estamos acostumbrado en las pruebas unitarias de los servicios.



```
package controllers;

import javax.transaction.Transactional;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
    "classpath:spring/junit.xml"
})
@Transactional
@WebAppConfiguration
public class MessageControllerTest extends AbstractTest {
```

@RunWith

Invoca la clase que ejecutará las pruebas unitarias de dicha clase.

@ContextConfiguration

Define los metadatos que se usarán para determinar cómo cargar y configurar las pruebas unitarias.

@Transactions

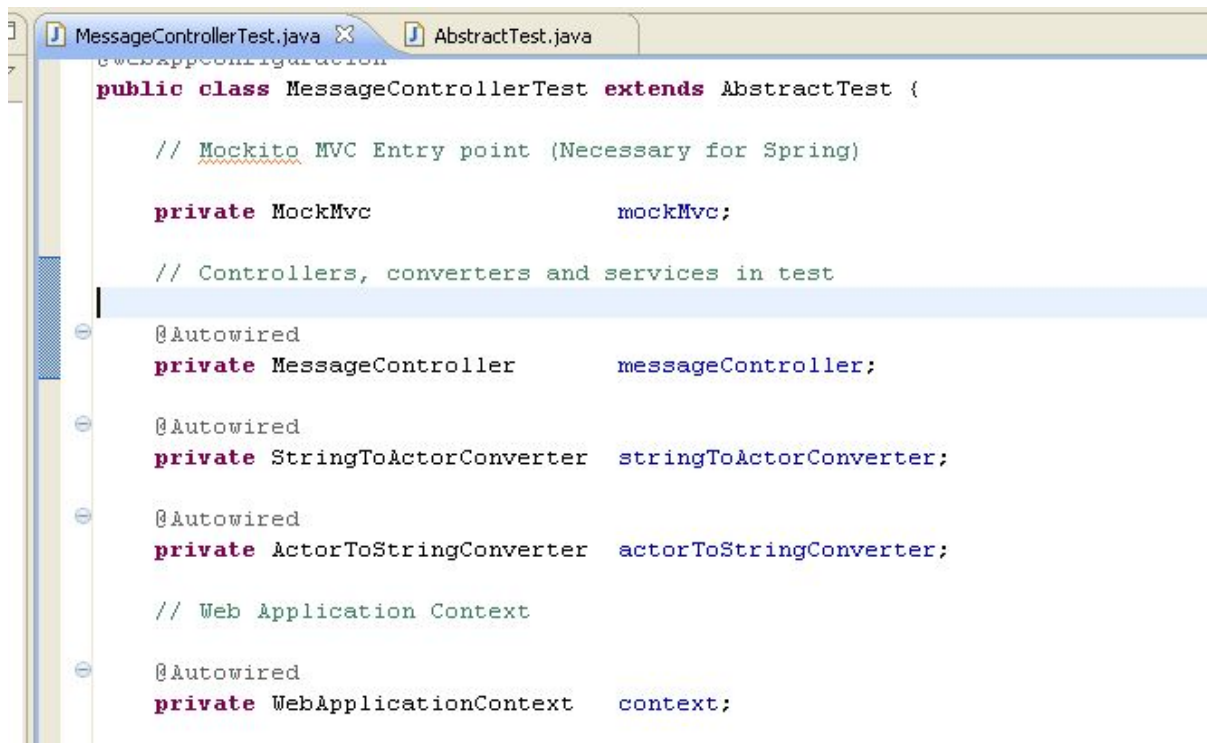
Hace que las acciones se conviertan en transacción y, en el caso de que una excepción sea lanzada, se reviertan los cambios.

DECLARACIÓN DE VARIABLES Y AUTOWIRED

El uso de Mockito para la creación de pruebas unitarias en los controladores implica que tendríamos que hacer uso de nuevas anotaciones de dicha librería.

En primer lugar tendremos que declarar un objeto MockMvc que posteriormente inicializaremos. MockMvc se encargará de realizar las acciones para comprobar que el controlador ha devuelto la respuesta deseada, cómo comprobar el código de respuesta, un valor presente en el código fuente, etc.

Por otro lado tendremos que declarar con la anotación @Autowired el controlador donde realizaremos las pruebas y los convertidores necesarios.



```
MessageControllerTest.java X AbstractTest.java
@ContextConfiguration
public class MessageControllerTest extends AbstractTest {

    // Mockito MVC Entry point (Necessary for Spring)
    private MockMvc mockMvc;

    // Controllers, converters and services in test
    @Mock
    private MessageController messageController;

    @Mock
    private StringToActorConverter stringToActorConverter;

    @Mock
    private ActorToStringConverter actorToStringConverter;

    // Web Application Context
    @Autowired
    private WebApplicationContext context;
```

MÉTODO DE CONFIGURACIÓN PREVIA

Al contrario que ocurre con las pruebas unitarias en los servicios, Mockito necesita inicializar la variable `MockMvc` con los convertidores necesarios y el controlador que vamos a testear.

Es importante el uso de la anotación `@Before`, ya que hará que Spring ejecute la configuración antes que los propios pruebas unitarios.

```
// Setting up the test environment

@Before
public void setup() {
    MockitoAnnotations.initMocks(this);
    final FormattingConversionService cs = new FormattingConversionService();
    cs.addConverter(this.stringToActorConverter); // Adding necessary converters
    cs.addConverter(this.actorToStringConverter);
    this.mockMvc = MockMvcBuilders.standaloneSetup(this.messageController).setConversionService(cs).build();
    MockMvcBuilders.webApplicationContextSetup(this.context);
}
```

PRUEBAS UNITARIAS

Comprobar que el formulario de envío de mensaje se muestra correctamente con “customer1”

Nuestro caso de uso debería permitir enviar un mensaje a otro Actor del sistema siempre que esté registrado como Customer o Administrator.

Primero realizamos la autenticación con el método *authenticate* de *AbstractTest* (el mismo que usamos para testear los servicios), usando el nombre de usuario de algún Customer o Administrator ya registrado en la base de datos.

Con el constructor de *RequestBuilder* le diremos a Mockito cual es la URL de la llamada GET. El segundo parámetro solo se especificará cuando se trate de una llamada POST y especifica el parámetro que envía la llamada (por ejemplo, un botón con valor “send” que envía el formulario).

Una vez realizada dicha llamada esperamos que el controlador devuelve la vista con código de respuesta 200 y con el formulario para el envío de un mensaje. Dichas acciones son las que especificaremos usando el método *perform* del objeto *MockMvc* que hemos inicializado anteriormente.

```

/*
 * Test case: A user (customer/admin) must be able to send a message to another user if it is well formed.
 *
 * A GET request is made to /message/send.do -> The user is prompted with the message form.
 *
 * A POST request is made to /message/send.do with param "send" and the user input was valid ->
 * The user is redirected to a list with his/her sent messages.
 *
 * A POST request is made to /message/send.do with param "send" and the user input was invalid ->
 * The user is redirected to the same form asking him/her to input valid data.
 */

@Test
public void getSendTest() throws Exception {
    this.authenticate("customer1");

    RequestBuilder requestBuilder;

    requestBuilder = MockMvcRequestBuilders.get("/message/send", "");

    this.mockMvc.perform(requestBuilder)
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.view()
            .name("message/send"));

    this.unauthenticate();
}

```

Enviar mensaje con “customer1”

Misma idea que la anterior prueba, pero en este caso se trata de una llamada POST donde tendremos que especificar los valores del formulario que enviaremos. En el constructor de RequestBuilder especificamos *contentType* para indicarle que se trata de datos de formulario y posteriormente los parámetros a enviar para realizar la prueba, asegurándonos que se tratan de todos los campos necesarios para enviar un mensaje (de lo contrario obtendremos un mensaje de error).

Con el objeto MockMvc obtendremos la respuesta a dicha petición. Para que la respuesta sea correcta tendremos que haber sido redirigido a la vista de los mensajes enviados y que el mensaje de redirección en el correcto.

```

@Test
public void postSendValidTest() throws Exception {
    final RequestBuilder requestBuilder;

    this.authenticate("customer1");

    requestBuilder = MockMvcRequestBuilders.post("/message/send", "send")
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .param("title", "test")
        .param("text", "test")
        .param("attachments", "")
        .param("recipient", "102")
        .param("send", "");

    this.mockMvc.perform(requestBuilder)
        .andExpect(MockMvcResultMatchers.model().hasNoErrors())
        .andExpect(MockMvcResultMatchers.model().attribute("message", "message.commit.ok"))
        .andExpect(MockMvcResultMatchers.status().isMovedTemporarily())
        .andExpect(MockMvcResultMatchers.view().name("redirect:/message/listSent.do"))
        .andExpect(MockMvcResultMatchers.redirectedUrl("/message/listSent.do?message=message.commit.ok"));

    this.unauthenticate();
}

```

Enviar mensaje incorrecto con “customer1”

Mismo procedimiento que el anterior caso, pero enviando en el formulario valores que cumplan alguna restricción del objeto.

En este caso esperamos que el usuario recibe un error y vuelve a la vista del formulario para corregir los errores.

```
@Test
public void postSendInvalidTest() throws Exception {
    final RequestBuilder requestBuilder;

    this.authenticate("customer1");

    requestBuilder = MockMvcRequestBuilders.post("/message/send", "send")
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .param("title", "")
        .param("text", "")
        .param("attachments", "")
        .param("recipient", "102")
        .param("send", "");

    this.mockMvc.perform(requestBuilder)
        .andExpect(MockMvcResultMatchers.model().hasErrors())
        .andExpect(MockMvcResultMatchers.view().name("message/send"))
        .andExpect(MockMvcResultMatchers.status().isOk());

    this.unauthenticate();
}
```