Como lanzar nuestro test Mockito

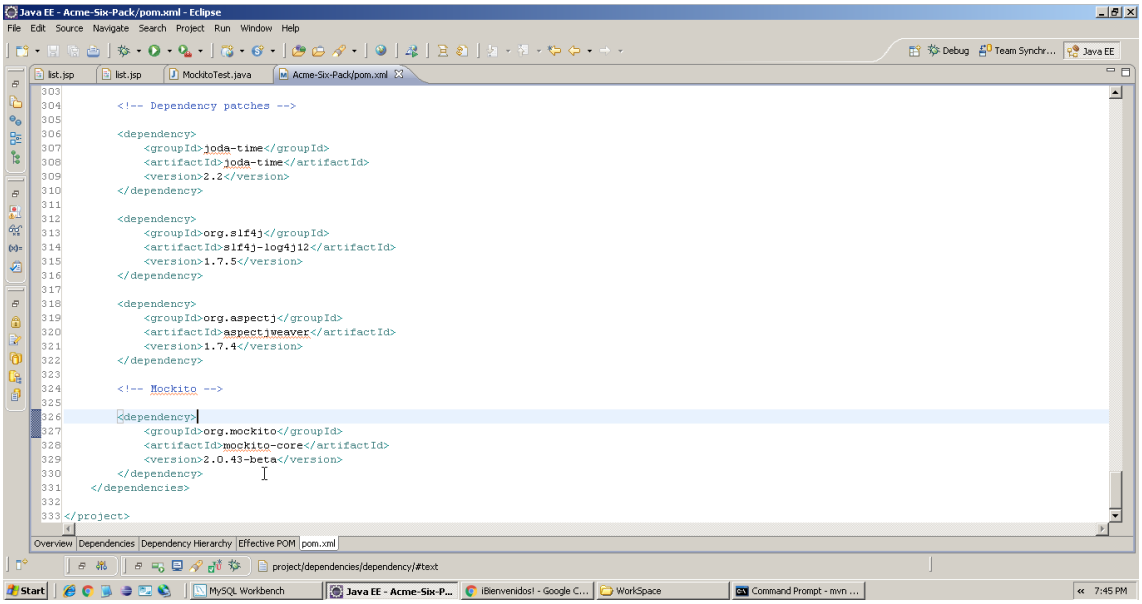
Los test de Mockito se lanzan como otros tests más de JUnit. No requiere nada especial ya que las librerías están incluidas como dependencias en Maven.

Información acerca del framework elegido

Hemos elegido el framework Mockito debido a su sencillez y la claridad con la que funciona. La versión elegida es una beta, en concreto la versión "2.0.43-beta" debido a ser la última estable y llevar cerca de 3 meses sin ser sustituida. El motivo de no usar una versión totalmente estable (1.10.19) es que la última actualización data de más de año y medio y los cambios entre la versión 1 y 2 son grandes.

Implementación de Mockito como dependencia

Lo primero que tuvimos que hacer es añadir las librerías de Mockito a nuestro proyecto. Para ello nos fuimos a la página oficial (http://mockito.org/) y seleccionamos que queríamos añadirla a nuestro proyecto desde maven central lo que nos condujo a las instrucciones (http://search.maven.org/#artifactdetails|org.mockito|mockito-core|2.0.43-beta|) . La implementación se realizó siguiendo los pasos ahí descritos, es decir, se accedió al archivo "pom.xml" en nuestro proyecto y se incluyó en la sección dependencies la librería.



Una vez añadido al archivo, solo habrá que lanzar maven o hacer un update y automáticamente descargará la librería.

Preparando test con Mockito

Para usarlo solo hay que crear una variable general del tipo MockMvc (la cual guardará todos los datos necesarios para la ejecución del test) y otra del tipo WebApplicationContext (la cual guardará lo relacionado con la aplicación Web que queremos ejecutar).

Posteriormente, en el método @Before de JUnit deberemos inicializar las anotaciones de Mockito mediante la orden "MockitoAnnotations.initMocks(this);", igualmente inicializaremos la variable creada anteriormente (en nuestro caso mockMvc) con la orden "MockMvcBuilders.standaloneSetup(firstController, secondController, …).build();"donde "firstController, secondController, …" serán los controladores que vamos a usar en los test. Mockito solo cargará estos controladores por lo que si se requiere alguno no incluido, los test mostrarán errores. Por último ejecutaremos los builders de mockito, concretamente los relacionados con las aplicaciones web usando la variable WebApplicationContext anteriormente declarada, la orden es "MockMvcBuilders.webAppContextSetup(wac);".

Todos estos pasos aplicados a nuestro proyecto se pueden observar en la siguiente imagen:

```
54    @Autowired
55    private WebApplicationContext wac;
56
57    private MockMvc mockMvc;
58
59
60    @Before
61    public void setup() {
62
63        MockitoAnnotations.initMocks(this);
64
65        mockMvc = MockMvcBuilders.standaloneSetup(registerController, welcomeController, loginController).build();
66
67        MockMvcBuilders.webAppContextSetup(wac);
68
69        unauthenticate();
70    }
```

Realización de test con Mockito

Realizar un test con Mockito no dista mucho de un test JUnit normal ya que en realidad es un test JUnit ejecutando unas sentencias de otra librería como podía ser el caso de spring o hibernate. Se crea un método público con el nombre del test y como característica hay que añadir a la declaración "throws Exception". Al igual que en otros test JUnit hay que poner justo antes de esta cabecera la anotación @Test en caso de que el test sea positivo. No se suelen realizar test que fallen a este nivel ya que el propio framework de Mockito te da la posibilidad de capturar el tipo de error que quieres obtener para los controladores.

En la siguiente imagen se puede observar la cabecera de uno de nuestros test para entender el funcionamiento básico de Mockito.

```
@Test
public void mockTestGet() throws Exception{
    RequestBuilder requestBuilder;
    requestBuilder = MockMvcRequestBuilders.get("/welcome/index.do", "");

    mockMvc.perform(requestBuilder)
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.view().name("welcome/index"))
        ;
}
```

Dentro del método hay que crear una variable del tipo RequestBuilder la cual inicializaremos con la siguiente orden “MockMvcRequestBuilders.get("urlACargar", "");” donde definiremos el tipo de petición http a realizar (get, post, put …) dependiendo de la orden con la que llamemos a “MockMvcRequestBuilders”. En “urlACargar”, debemos poner la url a la que deseamos hacer la petición (puesta en la cabecera de cada método en las clases incluidas en el paquete Controllers).

Tras definir la petición que queremos que ejecute Mockito hacia el controlador, le diremos que la ejecute mediante “mockMvc.perform(requestBuilder)” y con ese comando habrá ejecutado lo que le hayamos pedido en el requestBuilder. Justo en la misma orden debemos de poner los parámetros “.andExpect(MockMvcResultMatchers)” que denotarán la respuesta que esperamos de realizar el perform del requestBuilder.

Mediante MockMvcResultMatchers se pueden chequear muchas parámetros como por ejemplo:

- MockMvcResultMatchers.status() . Denota el estado http esperado. Algunos ejemplos son el “.isOk()” o “.isMovedTemporarily()”.
- MockMvcResultMatchers.view().name(“viewName”). Denota la vista que deberíamos haber recibido como respuesta. Esta vista usa los mismos nombres que se han definido en los “tiles.xml” del proyecto.
- MockMvcResultMatchers.model(). Denotan comprobaciones en el modelo devuelto como que no haya ningún error “.hasNoErrors()” o que contenga un atributo con un parámetro en concreto “.attribute(“paramInUrlView”, “containParamUrlView”)”. Estas comprobaciones también te permiten comprobar el contenido de las cookies o la fecha del ordenador remoto.

Con esto quedaría realizado un test estándar hacía los controladores con Mockito.

Test detallado con Mockito

A continuación se muestra un test realizado con Mockito:

```
93
94      @Test
95      public void mockTestRegister() throws Exception(
96          RequestBuilder requestBuilder;
97          requestBuilder =
98                  MockMvcRequestBuilders.post("/customer/create", "save")
99                  .contentType(MediaType.APPLICATION_FORM_URLENCODED)
100                 .param("name", "admin")
101                 .param("surname", "admin")
102                 .param("phone", "admin")
103                 .param("username", "userExample1")
104                 .param("password", "admin")
105                 .param("repeatedPassword", "admin")
106                 .param("acceptTerm", "true")
107                 .param("_acceptTerm", "on")
108                 .param("_createCreditCard", "on")
109                 .param("createCreditCard", "true")
110                 .param("_createSocialIdentity", "on")
111                 .param("save", "")
112                 .sessionAttr("actorForm", actorFormService.createForm())
113                 ;
114
115         mockMvc.perform(requestBuilder)
116             .andDo(MockMvcResultHandlers.print())
117             .andExpect(MockMvcResultMatchers.status().isMovedTemporarily())
118             .andExpect(MockMvcResultMatchers.view().name("redirect:../security/login.do"))
119             .andExpect(MockMvcResultMatchers.model().hasNoErrors())
120             .andExpect(MockMvcResultMatchers.model().attribute("messageStatus", "customer.commit.ok"))
121             ;
122
123         authenticate("userExample1");
124
```

```
125         requestBuilder =
126                 MockMvcRequestBuilders.post("/welcome/index.do", "")
127                 .contentType(MediaType.APPLICATION_FORM_URLENCODED)
128                 ;
129
130         mockMvc.perform(requestBuilder)
131             .andDo(MockMvcResultHandlers.print())
132             .andExpect(MockMvcResultMatchers.status().isOk())
133             .andExpect(MockMvcResultMatchers.view().name("welcome/index"))
134             .andExpect(MockMvcResultMatchers.model().hasNoErrors())
135         ;
136     }
```

En él se puede apreciar que realizar una petición post es algo más complejo ya que hay que denotar el parámetro asociado al “submit” (en nuestro ejemplo “save”), el tipo de contenido que vamos a enviar (MediaType.APPLICATION_FORM_URLENCODED) y todos los parámetros que deseemos enviar. Cabe destacar que los checkBox aparecen dos veces (una con guion bajo y otra sin él) ya que es la información que realmente se manda en una petición post en una web y que hay una orden muy recomendable de usar que es “sessionAttr()” que denota el tipo de entidad que se va a realizar. Es útil ya que nos preconfigura los parámetros que tendríamos al haber accedido al formulario.

Más adelante, al hacer el “.perform” también podemos encontrar otra orden (.andDo(MockMvcResultHandlers.print())) no comentada hasta ahora. Esta orden nos permite ver los parámetros que se están enviando y recibiendo al realizar las operaciones. Es muy útil por ver exactamente la respuesta recibida y partiendo de ahí poder añadir más resultados esperados y mejorar la eficacia de los test.