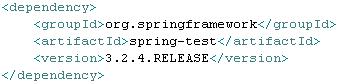
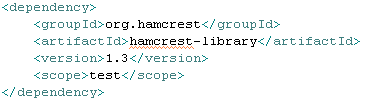
**DELIVERABLE 09 ITEM 5: TESTING THE CONTROLLERS**

In our projects, we have been testing every service to check that they are working as expected. The main purpose of these tests is to detect error in our code and fix them as soon as possible, so that the cost of these corrections is lower. However, as mentioned, we are only testing the services, and that is just a small part of the whole project. It would be convenient to test the controllers and the views too. For this project, we will be testing our controllers, using Spring MVC Test, a testing framework included in Spring. The required dependency is already included in our pom.xml file.



For this example, we will only test one controller, the AnnouncementUserController to be precise. Before we start explaining how we have implemented the tests, it is important to note that another dependency is needed to perform them. This dependency is called Hamcrest, and it includes the necessary methods to check and compare different elements.



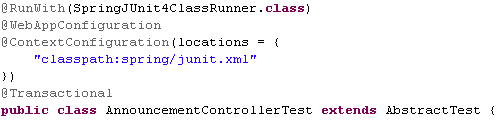
After including these dependencies and updating the project (so that Maven downloads the necessary files), we are ready to start developing our tests.

The first thing we need to do is to create a new package inside our tests package. We will name it “controllers”, in the same way that other packages are called “services” or “repositories”. We will store our test here.

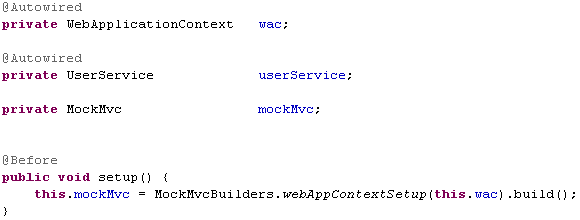
Then, we must create a new class. We named it “AnnouncementControllerTest”. This class will be run using JUnit, as we will explain later in this report. This class will extend “AbstractController”, to reuse the methods that allow us to simulate that a user is logged in, that is, the methods “authenticate” and “unauthenticate”. Note, however, that we will not be using “unauthenticate”, as we will always authenticate using “null” when we want to log out. This class needs several annotations in order to work properly. They are as follows:

* RunWith: this annotation specifies the class that needs to be used to run the tests. We will use “SpringJUnit4ClassRunner”, which is the same class that we are using to test our services.
* WebAppConfiguration: this indicates that the web application context must be loaded in the test.
* ContextConfiguration: this specifies the configuration of the application context where the tests will run. For our controller tests, we can reuse the same configuration that we included in the service tests, that is, an XML file called “junit.xml”.
* Transactional: this annotation tells Spring that the transaction support should be enabled for the test. This will ensure that any change that is made during the tests will be reverted after their completion.

This is what the class header (excluding imports) should look like.



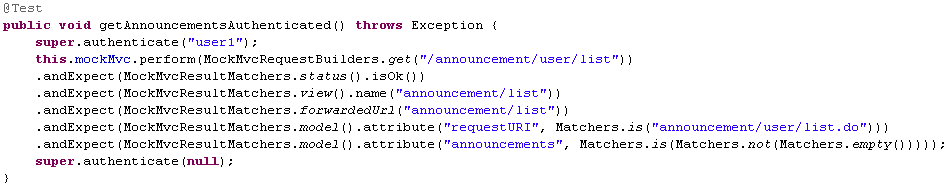
With that part done, the next thing that we need to add is the list of required elements for our tests. First, we need to autowire the web application context. Then, we would autowire every service that is required during the tests; in this case, the only service that was needed is the “UserService”. Finally, we need to create an object of type MockMvc. This object will be the main entry point of the tests, that is, the object that we will be using to perform each different test. After creating it, we mush initialize it. In order to do that, we will call a method named “webAppSetup”, that will take as a parameter the web application context that we autowired before, and after that we will use “build”. This small task will be included in a method called “setup”, that will be preceded by the annotation “Before” to specify that it must be executed before running the actual tests. The following image illustrates what we have explained in this paragraph.



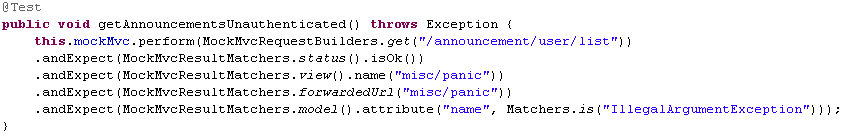
Now we can start writing our tests. As with every other JUnit test, each method will include the annotation “Test”. Moreover, in order to help understand what the tests do, we have written a brief Javadoc for each of them. Each test will include “throws Exception” in its header. For this example, we have developed tests for two of the different methods available in the AnnouncementUserController: “list”, which retrieves the list of announcements from the rendezvouses that a given user has RSVPd; and “create”, that initializes an empty announcement for a given rendezvous.

Please note that the screenshots that we will provide in this report will be slightly different from what we will include in the deliverable itself. The explanation for this is that because of how our Eclipse IDE is set up, it will try to “beautify” our code each time that we save a file; however, for these tests, it will do the exact opposite and make them harder to read, by chaining consecutive method calls in fewer lines and replacing static method imports with static calls, thus calling the same static classes several times. For the screenshots, we will write the tests using as many lines as we need, but we will keep the static calls as they help determine which class each call belongs to.

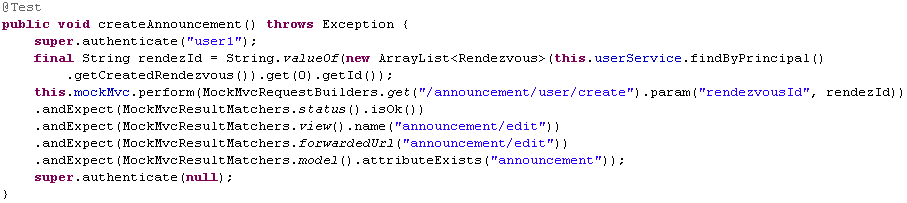
The first test that we have implemented is the listing of announcements for a user that is logged in. To simulate the user, we will use the “authenticate” method as we mentioned before. After that, we use the previously defined MockMvc object to perform a request. Since we need to use the GET method, we have to call “get” including the controller URL (that is, the URL that will be mapped to the controlled thanks to the “RequestMapping” annotations). Then, we can start chaining consecutive calls to a method named “andExpect”, that will perform assertions in a similar way to that of the “Assert” methods used in the service tests. For this test, we check that the HTTP status code is OK (which is the same as checking that it is 200), that the redirected view and the forwarded URL are both “announcement/list”, that the “requestURI” attribute needed for DisplayTag is correctly configured to “announcement/user/list.do”, and that the returned list of announcements is not empty. Then, we simulate that the user logs out.



For the second test, we want to check that an actor that is not authenticated cannot access this controller. After performing the request without previously logging in, we check that the redirected view is “misc/panic” and that an “IllegalArgumentException” is thrown. Note that a “panic view” is actually a correct response from the server, so the HTTP status code will still be OK.



Finally, for the creation test, we will check that a logged in user can start creating an announcement for a rendezvous that they have created. First, we need to retrieve the ID of any of their rendezvouses; we will use the first one. To perform the request, we need to add this ID as a parameter, and to do that we can either use the “param” method or directly add the parameter to the GET request. After that, we check that the user is redirected to “announcement/edit”, and that an announcement is included in the model.



With that, we have created some simple controller tests that will help us detect bugs in our code. To run them, just right click the class in the package view, go to “Run as…” and select “JUnit test”. These tests could be extended to check more elements from the model or more methods from the same controller or different ones, but, for this example, we think the checks that we included are enough.