Testing Pages

Here's a bit of what I've found out while writing tests for Wicket. The code is using Wicket 1.2 beta 3, but most of it should apply to 1.1.1 as well.

- To avoid code duplication, you can extend WicketTester to start in the application path you want through the WicketTester(String path) constructor. It might also be useful to add common initialization for all tests (like basic services and such) and provide an accessible method to add your standard authorization strategy for tests that need it (not all my tests do, in fact most don't). For instance:

```
public class AppTester extends WicketTester {
   public AppTester() {
       super("/admin");
   }
   public void initialize() { // not always called
       getSecuritySettings().setAuthorizationStrategy(new AdminAuthStrategy());
       ServiceInitializer.initializeDevelopmentServices();
   }
   protected ISessionFactory getSessionFactory() {
       return new AdminSessionFactory(this);
   }
}
```

- Similarly, you may extend JUnit's testcase (when it makes sense) to instantiate your tester.
- There are 2 things I usually test on a page: proper **navigation** and proper **data binding**. Data binding also includes correct **data validation** and **error messaging**. The following is an easy way to test the result of **attribute modifiers** (see also the mailing for few pointers to the source code of AjaxResponseTarget.respondComponent() as an example):

```
// e.g., Your TestCase -class
private final static Method getReplaceModelMethod;

static {
  try {
    getReplaceModelMethod = AttributeModifier.class.getDeclaredMethod("getReplaceModel");
    getReplaceModelMethod.setAccessible(true);
  } catch (Exception e) {
    e.printStackTrace();
    throw new RuntimeException(e);
  }
}

public void assertAttribute(String message, String expected, Component component, String attribute) {
  AttributeModifier behavior = getAttributeModifier(component, attribute);
  if (behavior != null) {
    try {
      IModel<?> model = (IModel<?>) getReplaceModelMethod.invoke(behavior);
      assertEquals(message, expected, model.getObject().toString());
      return;
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }

  fail("Attribute not found.");
}
```

- For **data binding tests**, I usually separate them in 2 parts:
  - **Data->HTML** (rendering): for this test, set your data on the database (through your services if you can) or pass it in the constructor of your class, then check if the correct components were generated and the data was correctly bound to the models of those components (when applicable. For instance:

```
    public void testRender() {
        // add some values directly to the database, to check the rendering (services are not
flexible enough for me)
        // template here is a Spring JDBCTemplate, which is quite useful for running direct SQL
queries without much red-tape
        template.execute(
                 "update config set value='sampleuser' where name='userName';" +
                 "update config set value='samplepass' where name='password';" +
                 "update config set value='true' where name='allowOffsiteAccess';" +
                 "update config set value='sampleiprange,anotherip' where name='trustedIPRange';" +
                 "commit;");
        // start the page - my page is package protected for better encapsulation, so I need an
ITestPageSource
        // if the page takes parameters, just pass them to the constructor instead of setting them
        // through the database (that allows for better reuse)
        // [tester] here is an instance of my AppTester described above
        tester.startPage(new ITestPageSource() {
            public Page getTestPage() { return new ChangeOffSiteAccessPage(); }
        });
        // check that the right page was rendered (no unexpected redirect or intercept)
        tester.assertRenderedPage(ChangeOffSiteAccessPage.class);
        // assert that there's no error message
        tester.assertNoErrorMessage();
        // check that the right components are in the page
        tester.assertComponent("feedback", FeedbackPanel.class);
        tester.assertComponent("form", Form.class);
        // ok, now check not only that the component is present, but also that the model object
        // contains the correct value (was correctly bound)
        tester.assertComponent("form:username", TextField.class);
        final TextField usernameField = (TextField) tester.getComponentFromLastRenderedPage("form:
username");
        assertEquals("sampleuser", usernameField.getModelObject());
        ... other components
    }
```

- **HTML->Data** (form submitting): this test verifies that the data you set on the form components gets sent to the correct data objects, and that the correct data validation and type conversion gets done. The code for those tests is typically centered around the FormTester. Below is an example of it:

```
    public void testInvalidLogin() {
        // create the form tester object, mapping to its wicket:id
        FormTester form = tester.newFormTester("form");
        // set the parameters for each component in the form
        // notice that the name is relative to the form - so it's 'username', not 'form:username'
as in assertComponent
        form.setValue("username", "test");
        // unset value is empty string (wicket binds this to null, so careful if your setter does
not expect nulls)
        form.setValue("password", "");
        // slight pain in the butt, for RadioGroups the value string is a bit complicated
        // I believe it's the pageversion followed by the complete component name (not the
relative, now) then the id for the choice itself
        // the easiest way is to render the page once and then copy & paste
        // pageversion didn't seem to have an effect, so I always replace it by 0
        form.setValue("offSiteAccessEnabled", "0:form:offSiteAccessEnabled:Yes");
        // another one to pay attention: listviews
        // here I have a 3 column iteration through a listview with 10 rows iterating through
another listview
        // so it's the listview followed by the row id followed by the inner component in the
listview
        form.setValue("addressRow:0:addressColumn:0:mask", "");
        // all set, submit
        form.submit();
        // check if the page is correct: in this case, I'm expecting an error to take me back to
the same page
        tester.assertRenderedPage(ChangeOffSiteAccessPage.class);
        // check if the error message is the one expected (you should use wicket's
internationalization for this)
        // if you're not expecting an error (testing for submit successful) use
assertNoErrorMessage() instead
        tester.assertErrorMessages(new String[] { "A Login and Password are required to enable
offsite access." });
    }
```

- For **navigation** tests, in most cases you can use the **assertRenderedPage(Class)** method. You can follow links by submitting the forms or clicking the link. Below are a couple of examples:

```
    public void testCancelLink() {
        tester.clickLink("form:cancelButton");
        tester.assertRenderedPage(ChangeSystemParametersPage.class);
        ... clicked on cancel, verify that the data was not changed in any way
    }
    public void testGenerateQueryReport() throws UnsupportedEncodingException {
        // this one is a bit more interesting, as this page does a download-on-submit
        ... prepare the data for the test
        // prepare the form, fill the data, and submit
        final FormTester form = prepareFormTester();
        form.setValue("dataType", "0:form:dataType:query");
        form.submit();
        // check that the submit created a download link
        final MockHttpServletResponse servletResponse = tester.getServletResponse();
        // in this case it's a CSV report, so just convert the whole thing to a string
        final String report = new String(servletResponse.getBinaryContent(), servletResponse.
getCharacterEncoding());
        // compare it
        assertEquals("QUERY COUNT\n\n", report);
    }
```

- A few random testing tips:
  - Code reuse is important, but test code clarity is essential. No one tests for tests, so if tests are too complex due to heavy inheritance or intricate execution paths, they will contain bugs themselves, defeating the purpose of testing in the first place.
  - Test execution time is important too, if tests take 20 minutes to run you won't run them often (if ever). Try to avoid tests that rely heavily on the database, or create mock objects for the services those rely on.

- In my code, most pages have package (default) scope. This decreases the apparent complexity of the system (you only need to know a few pages at a time), decreases coupling and makes the packages more cohesive. However, Wicket's relatively heavy use of reflection will cause runtime exceptions sometimes. Just be sure to touch all pages/panels during tests (unit tests AND manual tests).