AutomatedWebappTestsAPI

Introduction

This wiki page contains the API draft for the GSoC project "Automated webapp tests for MyFaces core + extensions" GSoC2010_AutomatedTests. Feel free to comment and contribute!

API draft

First idea

Using inheritance.

Example

The following class `ExampleTestCase` shows how a test case could look like with this new framework

```
@View(id = "test.xhtml", pageBeans = @PageBean(clazz = ExampleBean.class)) // this view will be accessed and
the beans will be installed
@Dependency.List({
    @Dependency(groupId = "javax.el", artifactId = "el-api", version = "2.2"), // for special dependencies
    @Dependency(StandardDependency.EL_API_2_2) // for standard dependencies
})
@RunWith(WebappTestRunner.class)
public class ExampleTestCase extends WebappTestCase
{

    @Inject
    private ExampleBean exampleBean; // specified as PageBean, thus it is a valid resource
    // NOTE that this ExampleBean instance will have to be a cglib proxy of the real bean class
    // in order to allow the expected method call testing

    public void testApplyInputValue()
    {
        // START first phase of test case: validating the view that "only" has been rendered
        // NOTE that facesContext and root are from the first (=rendering) request at this time
        assertNotNull(root.findComponent("fieldId"));
        // END first phase

        // START second phase of test case: setting values and clicking some button or link
        input("test value").into("fieldId"); // set values using the clientId
        click("buttonId");  // execute some action
        // END second phase

        // START third phase of test case: verifying the postback
        // NOTE that facesContext and root are from the second (=postback) request at this time
        assertThat(facesContext.getExternalContext().getRequestParameterMap().get("formId:fieldId")).is("test
value");
        assertThat(exampleBean.getInput()).is("test value").before(PhaseId.INVOKE_APPLICATION);
        expectCall(exampleBean.action()).in(PhaseId.INVOKE_APPLICATION);
        // for void methods:
        exampleBean.actionListener(null);
        expectLastCall().in(PhaseId.INVOKE_APPLICATION);
        // END third phase

        // now we could again set some input and click some button or link and do some other assertions...
    }

}
```

Base Class

The base class for every test case will be `WebappTestCase`, which contains all necessary methods to set up the test case, to do assertions and to input data into fields and click buttons just like a user would do. These methods will use a fluent API approach to ease usage. Furthermore any configuration will happen via annotations and resources will be injected using the @Inject annotation from javax.inject.

```
public abstract class WebappTestCase
{

    protected FacesContext facesContext; // will be set appropriately before test execution
    protected UIViewRoot root; // will be set appropriately before test execution
    // TODO maybe introduce a Page object which contains the current UIViewRoot
    // and some more information (like the HTML).

    private List<Dependency> _dependencies = new ArrayList<Dependency>();

    public List<Dependency> getDependencies()
    {
        return Collections.unmodifiableList(_dependencies);
    }

    // TODO should only happen once for the whole test class
    protected void setUpWebapp() throws Exception
    {
        // get all configuration (from local annotations + super-class annotations
        // to allow preconfigured Test super classes)
        Class<?> clazz = getClass();
        do
        {
            // single Dependency
            Dependency dependency = clazz.getAnnotation(Dependency.class);
            if (dependency != null)
            {
                _dependencies.add(dependency);
            }

            // aggregated Dependencies
            Dependency.List dependencies = clazz.getAnnotation(Dependency.List.class);
            if (dependencies != null)
            {
                _dependencies.addAll(Arrays.asList(dependencies.value()));
            }
        }
        while ((clazz = clazz.getSuperclass()) != WebappTestCase.class);
    }

    // TODO should only happen once for the whole test class
    protected void tearDownWebapp() throws Exception
    {
        _dependencies = null;
    }

    /*-------------------------------------------------------------
      Begin methods, that interact like the user would do
     -------------------------------------------------------------*/

    /**
     * Inputs the given value in the field specified by the into() method.
     * Typical usage: input("value").into("field-client-id");
     */
    public Input input(String value)
    {
        return new Input(value);
    }

    /**
     * Clicks the UICommand component with the given clientId
     * @param clientId
     */
    public void click(String clientId)
    {
        // TODO
    }

    /*-------------------------------------------------------------
      Begin methods, that verify expected behavior or values
```

```
 ----------------------------------------------------------------*/

public final <T> Assertable<T> assertThat(T injectedMethodCall)
{
    return new Assertable<T>(injectedMethodCall);
}

public final Call expectCall(Object methodCall)
{
    // TODO record call
    return new Call();
}

public final Call expectLastCall()
{
    // TODO record call
    return new Call();
}

/*-------------------------------------------------------------
  Begin classes needed for fluent API
 ----------------------------------------------------------------*/

public final class Assertable<T>
{

    private T _injectedMethodCall;
    private T _expectedValue;
    private PhaseAware _phaseAware;

    public Assertable(final T injectedMethodCall)
    {
        _injectedMethodCall = injectedMethodCall;
        // somehowe find out what method has been called on the injected resource
        // (most certainly by using cglib-proxies)
    }

    public PhaseAware is(final T expectedValue)
    {
        _expectedValue = expectedValue;
        _phaseAware = new PhaseAware();

        return _phaseAware;
    }

    // has to be called in the right phase by the test
    private boolean doAssert()
    {
        return _expectedValue == _injectedMethodCall
                || (_expectedValue != null
                        && _expectedValue.equals(_injectedMethodCall));
    }

}

public final class PhaseAware
{

    private boolean _after;
    private PhaseId _phase;

    public void after(PhaseId phase)
    {
        _after = true;
        _phase = phase;
    }

    public void before(PhaseId phase)
    {
        _after = false;
        _phase = phase;
```

```
        }

    }

    public final class Input
    {

        private String _input;

        public Input(String input)
        {
            _input = input;
        }

        public void into(String clientId)
        {
            // TODO input data into component
        }

    }

    public final class Call
    {

        private PhaseId _phase;

        public Call()
        {
            // TODO find out which method (most certainly by using cglib-proxies)
        }

        public void in(PhaseId phase)
        {
            _phase = phase;
        }

    }

}
```

Preconfigured base classes

Because of the fact that `WebappTestCase` will read the configuration annotations from all its super classes (stopping by itself), it is very easy to create preconfigured test cases e.g. for extension testing. `ExtensionTestCase` shows an example.

```
@Dependency(groupId = "org.apache.myfaces", artifactId = "myfaces-extension", version = "1.0.0")
public abstract class ExtensionTestCase extends WebappTestCase
{

    // in this way we can provide preconfigured superclasses for extension tests

}
```

Second idea

Using separate configuration classes and an annotation to bind a config to a test class and also some helper classes instead of inheritance.

Example

The following class `ExampleTestCase` shows how a test case could look like with this new framework (second idea)

```
@View(id = "test.xhtml", pageBeans = @PageBean(clazz = ExampleBean.class)) // this view will be accessed and
the beans will be installed
@Configuration(SomeConfiguration.class) // the configuration class contains the Dependencies
@RunWith(WebappTestRunner.class)
public class ExampleTestCase
{

    @Inject
    private ExampleBean exampleBean; // specified as PageBean, thus it is a valid resource
    // NOTE that this ExampleBean instance will have to be a cglib proxy of the real bean class
    // in order to allow the expected method call testing

    @Inject
    private WebappTester tester; // will provide all necessary methods to control the test case

    public void testApplyInputValue()
    {
        // START first phase of test case: validating the view that "only" has been rendered
        // NOTE that facesContext and root are from the first (=rendering) request at this time
        tester.assertNotNull(root.findComponent("fieldId"));
        // END first phase

        // START second phase of test case: setting values and clicking some button or link
        tester.input("test value").into("fieldId"); // set values using the clientId
        tester.click("buttonId");  // execute some action
        // END second phase

        // START third phase of test case: verifying the postback
        // NOTE that facesContext and root are from the second (=postback) request at this time
        tester.assertThat(facesContext.getExternalContext().getRequestParameterMap().get("formId:fieldId")).is
("test value");
        tester.assertThat(exampleBean.getInput()).is("test value").before(PhaseId.INVOKE_APPLICATION);
        tester.expectCall(exampleBean.action()).in(PhaseId.INVOKE_APPLICATION);
        // for void methods:
        exampleBean.actionListener(null);
        tester.expectLastCall().in(PhaseId.INVOKE_APPLICATION);
        // END third phase

        // now we could again set some input and click some button or link and do some other assertions...
    }

}
```

Base Class

There will be no base class for this idea, but some Configuration classes and a WebappTester to provide the needed functionality (from the superclass of the first idea).

Preconfigured base classes

As there are no base classes, configuration classes will be used instead. However, there's no reason why a pre-configured base class couldn't be created which delegates calls to the helper classes.

Configuration

The framework uses a config class to get configuration options like the JSF implementation (MyFaces or Mojarra) and the related version of it. These configuration options will have standard values and will be configurable via System properties (which are very easy to set using maven).

It should be possible to provide ConfigurationTestSuits.

Example:

```
@ConfigurationTestSuite(testSuite = AllMyFacesTests.class,
  testConfigurations = {MyFacesJettyConfig.class, MyFacesTomcatConfig.class, MyFacesEmbeddedGlassfishConfig.
class})
public class AllTestsWithCommonConfigs extends ConfigurationTest
{
}
```

... means that all tests bundled by the test-suite AllMyFacesTests should be executed with the given configurations.

Dependencies

- JUnit 4.x
- HtmlUnit (for accessing the pages)
- javax.inject (for @Inject)
- CGLib (for proxies to verify expected method calls)
- MyFaces core API (for the server-side tests --> FacesContext etc.)
- Arquillian (to start containers) ???? Should we use it ???? --> maybe use something different.

Ideas

Feel free to post your ideas and opinions here!