

# How to Unit Test the User Interface of Web Applications

A tutorial for ApacheCon US 2005

© 2005 Thomas Dudziak

[tomdz@apache.org](mailto:tomdz@apache.org)

# Contents

Introduction.....	3
The sample web application.....	4
Overview of the approach.....	7
Lesson 0: Setting up the tests.....	9
Goal:.....	9
Lesson 1: The login.....	15
Goal:.....	15
Lesson 2: Clicking a link.....	23
Goal:.....	23
Detour: Logging.....	28
Lesson 3: Database-driven testing.....	31
Goal:.....	32
Detour: Using Spring in the test case.....	32
Back on track: Writing the test.....	40
Lesson 4: Testing AJAX-enabled web pages.....	47
Detour: AJAX in the sample web application.....	49
Back on track: Writing the test.....	53
Lesson 5: Testing back button & double posting.....	58
Goal:.....	59
Summary and outlook.....	65

## Introduction

It is general knowledge that testing is crucial to the success of a software development project. Given the complexity of today's software systems, there will be errors in them no matter how good the team and process is. And you do want to find them as early as possible, not let your customer encounter them during normal operation.

To this end, several levels of testing are usually applied to the different parts of the system, and at different levels of abstraction.

For instance, unit tests are meant to test units of code, usually single methods or classes, that perform a certain function, and not more. These tests are white or gray box tests in the case that the test author knows how the unit works, and can therefore determine what the typical and boundary cases are. This kind of tests is usually implemented in code and performed automatically which means they can be fast. Thus, they can be run often. Methodologies like Test Driven Development and techniques like Refactoring rely on building comprehensive unit tests that are run often.

Functional testing on the other hand means ascertaining that the software performs what the customer wants. Where unit tests are the developers way of making sure that the individual building blocks do what they are supposed to, functional tests are written from the user's perspective.

Functional tests are thus often derived from the functional requirements, and it is not unusual that a set of functional tests is used by the customer for acceptance criteria.

One problem with functional tests for systems with a graphical user interface, is that they often need to be performed manually. E.g. checklists are defined which then are executed by a tester step by step. This of course is rather labor-intensive, tedious and error-prone, which in turn often leads to a reduction in the amount of functional testing performed.

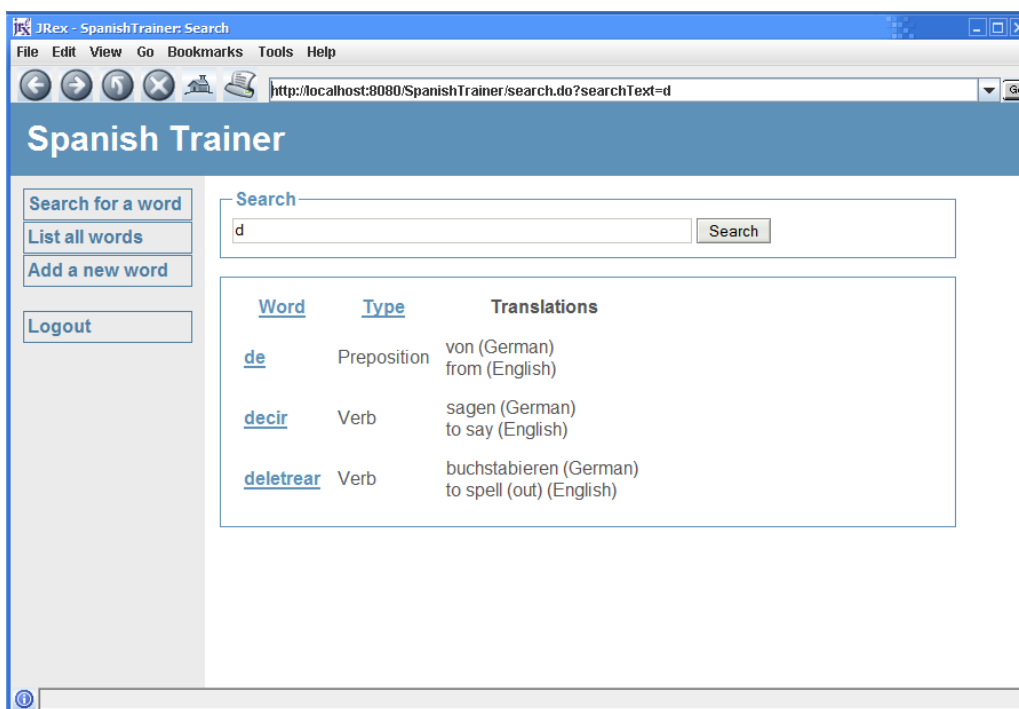
Rather, functional tests should be approached like unit tests. They should be defined as early as possible, and be run as often as possible. This is the point where automation in the style of unit tests can help. If you can let the functional tests run automatically – even if they take some time – then you will let them run more often.

This tutorial will present a technique for automating the functional tests in a unit test-like way. Several lessons will show how to use the *webappunit* framework for testing typical functions of web applications.

## The sample web application

For the purpose of this tutorial, we'll be using a sample web application. A year ago, I was in Spain for a longer period of time, in order to learn Spanish. But after I got back home, I quickly got overwhelmed again with work and so failed to continue learning Spanish (part of the reason might also have been my laziness, but I digress).

Being a software developer, the obvious solution was to create a software that would help me in learning Spanish, and so the *SpanishTrainer* web application was born. Not only would it help me better my language skills, but, I figured, I could also use it as a vehicle to test all the new and fancy stuff in web and application development, especially for Java-based systems, such as AJAX, ORM, Spring etc.



*Sample web application*

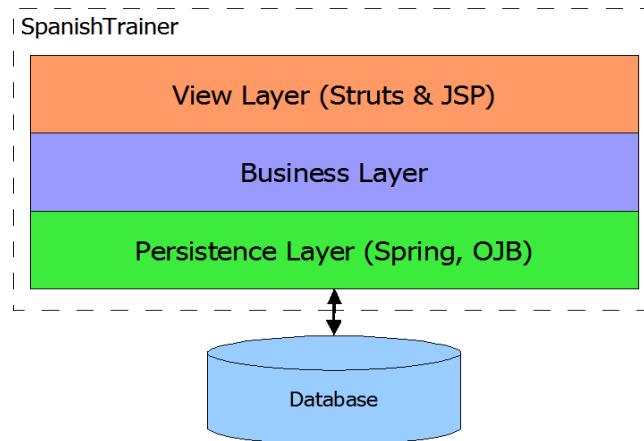
While the testing approach presented in this tutorial is largely language independent, we'll be discussing it using a Java-based web application for two reasons:

First, the testing infrastructure is readily available for Java web applications whereas for other languages or environments, e.g. Ruby or .Net, it has yet to be created.

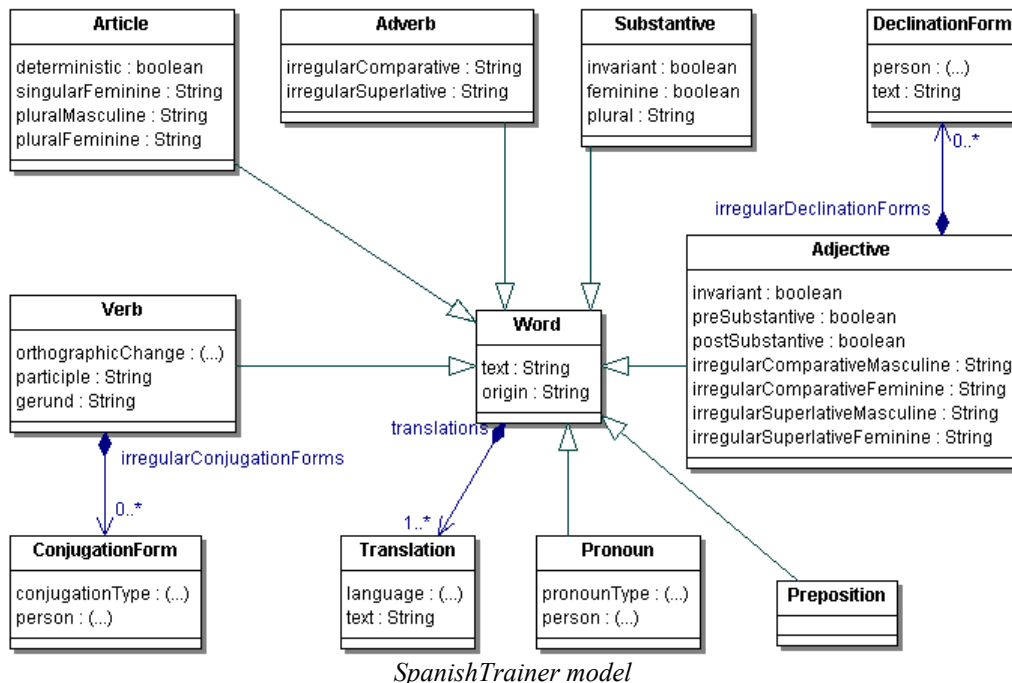
Second, one important aspect in testing web applications is the ability to access the web server during the tests. This is most easily done when using an embedded web server. While it certainly is possible to embed web servers like Apache or IIS into applications, it is not at all easy to access the runtime information of the web application that we might be interested in. Not so for Servlet containers such as Jetty or Tomcat. Here we can access almost any information whenever we so desire.

In short, the SpanishTrainer web application (currently) is a Struts-based web application that uses the help of Spring to simplify the infrastructure work, and employs OJB as the ORM layer for storing and retrieving data from the database.

As the internals of the sample web application are not really relevant to the functional testing shown here, we'll give only a brief overview of its structure and the used tools/libraries.



At the core of the business layer is the grammatical model of the Spanish language. While not complete yet, it is powerful enough to describe most parts of the Spanish language in a semantically rich way.



We will not go into much detail of the business model. It suffices to say that it is a rather standard business model of only small size. Accordingly, there are only a few business services that deal with the business objects in the database, and provide for the handling of the Spanish grammar, e.g. by performing the conjugation of verbs.

In the web layer, the SpanishTrainer web application uses Struts (<http://struts.apache.org>). While there are newer web frameworks out there that may be more powerful/elegant/modern/you name it ..., Struts can still be seen as the de-facto standard for Java web applications. Nearly every Java web application developer has worked with or still does work with Struts. This makes it the perfect candidate for a tutorial web application.

That is not to say that the presented approach is Struts-only. As will be apparent later, quite to the contrary, you can use the approach with almost any type of web application, at least to some degree.

To make matters a bit more interesting for the testing, one part of the web application uses DWR (<http://getahead.ltd.uk/dwr>) which is a library that makes it quite easy to integrate AJAX-style functionality into Java web applications. Accordingly, one of the lessons will deal with testing AJAX-enabled web pages.

The database layer is using OJB (<http://db.apache.org/ojb>) and DdlUtils (<http://db.apache.org/ddlutils>) to deal with the database-related tasks, e.g. writing words to the database or querying for words. OJB provides the object-relational mapping (ORM) which automatically maps business objects (words and their translations etc.) to database tables and vice versa. DdlUtils is used to initialize the database in a vendor-independent way, i.e. create the database, create the tables and associated constraints, and insert initial data.

Other solutions, like other ORM tools such as iBATIS or Hibernate, or even plain JDBC could certainly have been used. However, the integration between OJB and DdlUtils will prove useful for our tests as you'll see later.

Both the web and database layer use Spring to simplify common tasks, and to integrate with the other layers. For instance, database transaction handling is completely handled by Spring by the means of (sort-of) transaction aspects that – transparently to the application – start and commit/rollback transactions at the level of business methods. Likewise, the wiring of view components (Struts actions) and business logic (business services like the word management component) is also done via Spring. In one of the lessons, we will have a slightly more detailed look at Spring, but if you're interested in how exactly this works and what Spring can do, please have a look at the Spring documentation at <http://www.springframework.org/documentation> and the various tutorials dealing with Spring.

A great part of the necessary configuration files (e.g. for OJB, Spring, Struts) is auto-generated from the source code files by the help of XDoclet (<http://xdoclet.sourceforge.net/xdoclet>) and Ant (<http://ant.apache.org>). For instance, DdlUtils uses an abstract, vendor-independent database schema in XML form that is auto-generated by the XDoclet module of OJB. This same database schema will be used later on in the tests to initialize the database for specific purposes.

## Overview of the approach

(*webappunit* is a preliminary name until the library has found a new home.)

The testing approach presented in this tutorial mainly builds upon two components: a normal browser that is embedded into the web application and controlled via its public interface, and an embedded servlet container/web server.

One of the problems in testing the user interface side of web applications, especially modern Javascript-based and AJAX-enabled ones, is that web browsers are quite complicated things that differ in small and not-so-small things. While there are standards like (X)HTML, CSS and ECMAScript, their support in current web browsers may be incomplete or at times even erroneous. Web applications have to work around this and deal with these differences.

Approaches like HtmlUnit (<http://htmlunit.sourceforge.net>) and jWebUnit (<http://jwebunit.sourceforge.net>) achieve a certain level of success rebuilding the intricacies of the diverse browsers, but this is obviously only possible to some degree unless one re-implements the browsers. In order to ensure that a web application really works with a given browser thus requires that this browser is actually used for the tests.

This is obviously not a problem when performing the tests manually, however manual performance of tests is cumbersome and time-consuming at best. Thus, an automated testing facility that uses real browsers and gives access to its public interface and the structure of the currently loaded web page is needed.

This can be achieved in two ways. The first way is to execute the tests within the browser in the context of the tested web application. This approach is for instance pursued by the Selenium test environment (<http://selenium.thoughtworks.com/index.html>). Here, the test framework uses JavaScript executed within the same browser to test the web application.

The other possibility is to control the browser via its Public API. Most modern web browsers provide a browser component that is intended to be used by application developers to embed the browser into their application. This component defines an API with which the browser can be controlled from the outside. In addition, they expose the DOM for the current web page, a set of standards defined by the W3C (<http://www.w3.org/DOM>). They make it possible to query and even interact and manipulate the current document from the outside. This is the approach *webappunit* uses.

Depending on which browser shall be embedded and into which environment, it is more or less easy to embed the browser. For instance, in .Net programs it is quite simple to integrate the Internet Explorer into the application and control it via its COM API (at least when using C# or Visual Basic). The same can be achieved (though perhaps requiring more work) for Mozilla/Firefox and Konqueror/Safari.

In the Java world, this gets slightly more complicated by the fact that these browsers are written in C/C++. Fortunately, there exist projects like JREx (<http://jrex.mozdev.org>) and products like JExplorer (<http://www.jniwrapper.com/pages/jexplorer/overview>) which provide access to Mozilla and Internet Explorer to normal Java applications.

Webappunit in its current version uses JReX to provide unit tests access to a real web browser which they can control (e.g. load a web page, click a link), and from where they can gain access to and manipulate the currently loaded web page (via DOM level 2). Thus, the web application can be accessed via a real web browser, and the browser is controlled by the unit test.

The actual browser details are abstracted via a facade interface, so that in the future access to other browsers such as Internet Explorer or Safari, can be provided without requiring changes to the unit tests.

On the server side, it is actually easier in the Java world. Servlet containers can (depending on which one is used) be easily embedded into and controlled by a Java application. Likewise, all relevant information about a running web application, i.e. its state and incoming and outgoing data, can be intercepted and traced without too much work. The same level of control is a lot harder to achieve in non-Java environments.

Webappunit currently uses the Jetty servlet container (<http://jetty.mortbay.org/jetty>) which has been explicitly designed for this kind of embedding operation. With it, webappunit can give unit test access to such useful information as the incoming requests or to the sessions of a web application.

Work on a similar binding for Tomcat (<http://tomcat.apache.org>) has already begun.

In the following lessons, you will learn how to use webappunit to implement automated functional tests that deal with common scenarios such as data entry into forms, handling of the back button and double posting.

## Lesson 0: Setting up the tests

### **Goal:**

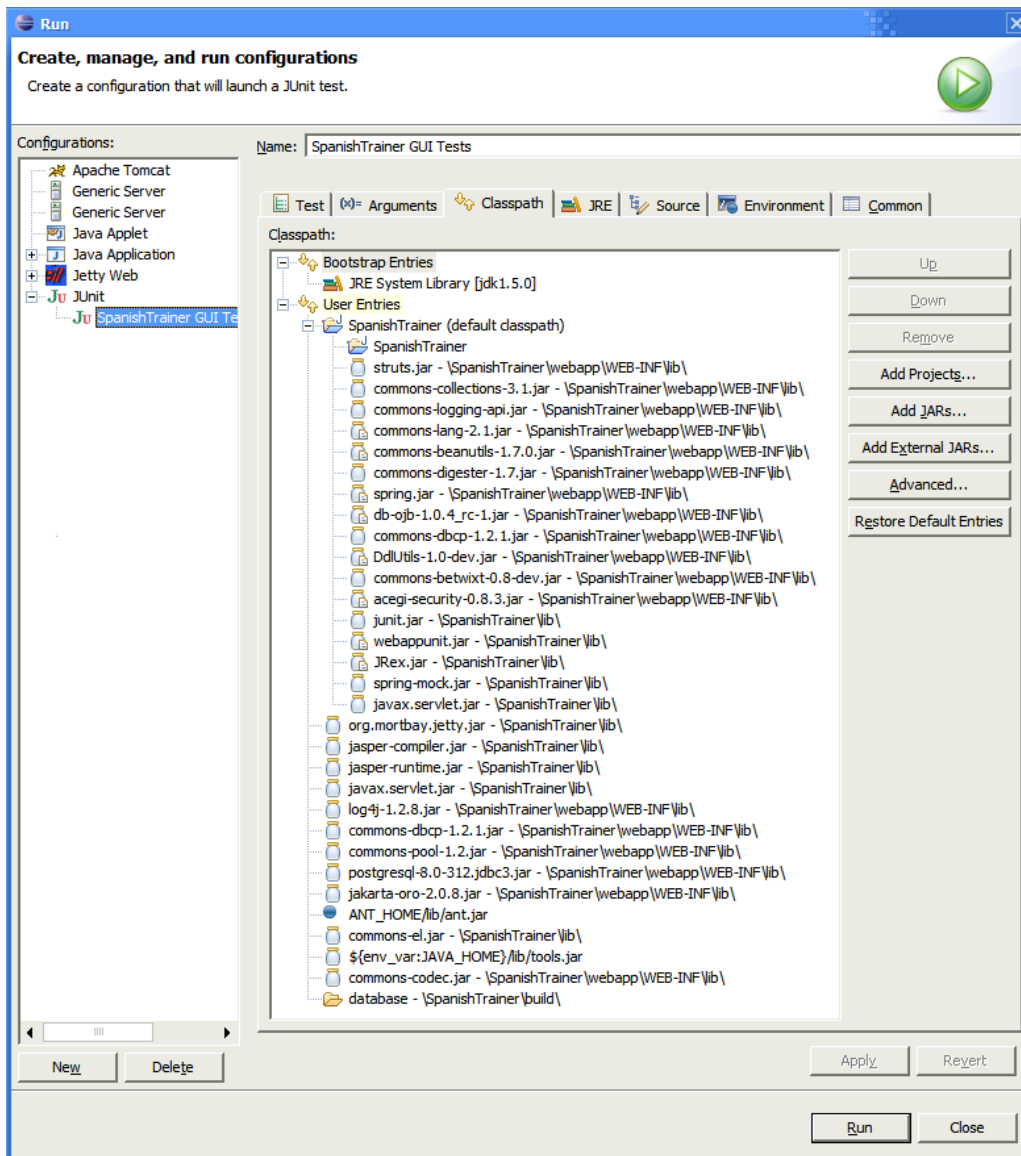
Create and run a first unit test that starts the web server and browser, and then shuts them down.

Before we can actually write tests, we need to setup the environment. Basically, three things are required besides your web application and the actual unit tests:

- Junit, and webappunit
- For our tests, we'll be using JREx and the GRE (gecko runtime environment, basically a stripped-down Mozilla)
- Jetty

Per default, the webappunit distribution contains both JREx/GRE and Jetty, so there is no need to download them. Since we're embedding the webserver, and our web application contains JSP pages, we however need a Java development kit of version J2SE 1.4 or newer. This is because the JSP compiler generates and compiles a Java source file from the JSP file, and for this it needs the classes for the Java compiler, which are contained in the `tools.jar` file that resides in the `lib` sub-folder in your Java environment.

For the sample web application, all libraries that are not required by the web application itself, but only for building and running the tests, are located in the separate folder `lib`. This way, Ant can simply ignore these libraries when creating a WAR file for deploying. Within this folder, the sub-folder `jrex_gre` contains the GRE shared libraries and other configuration files. Its contents are operation system dependent content, for windows for instance a couple of DLLs.



*Eclipse unit test run configuration*

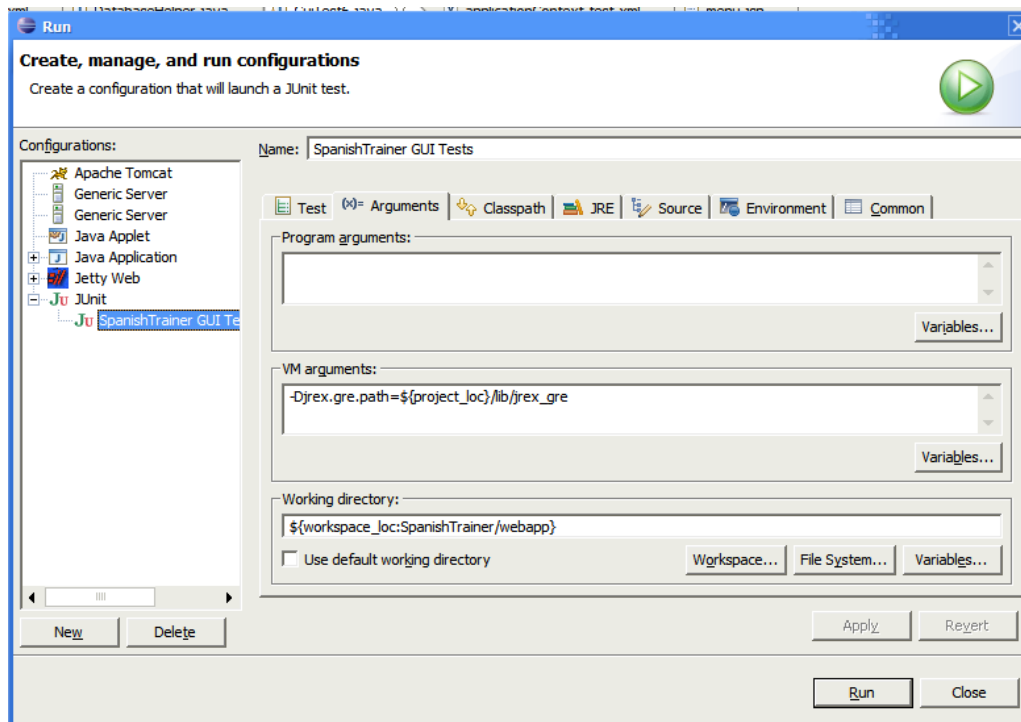
One word about the project's directory layout. When the web application shall be run directly in a web server without a deploy step in between, it is advisable to setup your project so that the web application structure is readily available. A web server – whether embedded or as a plugin of the IDE – can then directly use this structure to run your web application. E.g. for Eclipse a useful directory structure would be:

<b>&lt;root folder&gt;</b>	- contains the build files (build.xml, Eclipse's files)
- <b>src</b>	
- <b>java</b>	- contains the Java source files and some configuration files (log4.properties), which Eclipse automatically copies to the build folder
- <b>test</b>	- contains the Test source files
- <b>build</b>	
- <b>test</b>	- the Test source files will be compiled to this directory
- <b>lib</b>	- the libraries not required for running the web application, but e.g. for the tests
- <b>jrex_gre</b>	- the gecko runtime environment
- <b>webapp</b>	- contains the publicly visible content
- <b>WEB-INF</b>	- contains configuration files like web.xml
- <b>jsp</b>	- contains the jsp files
- <b>classes</b>	- contains the compiled Java classes; this is the build folder for Eclipse
- <b>lib</b>	- contains the web application's libraries

This way, you will automatically have a proper web application, and there is no extra deploy step necessary. However, an external build step is still needed for generating the configuration files via XDoclet, but invoking it is not required often.

You'll need to invoke two Ant targets in order to get a working web application. The first one, `gen-config-files`, invokes XDoclet for generating the Struts and Spring configuration files. Invoking it is only needed initially and when something in the configuration has changed. The second one is called `test-data`. It creates the database and fill it with the initial test data using DdlUtils.

For using JREx, two additional things are needed. First, you need to define where the GRE is located by setting the Java property `jrex.gre.path`. For sample web application, this is the path `lib/jrex_gre`. In the Eclipse run configuration this then looks like:



*Eclipse unit test run configuration*

We also have to tell our web server where to find the web application, i.e. its physical location. But we can take a shortcut here and simply execute the unit tests in the web application's directory. This allows us to use the current directory (“.”) in the unit test, and webappunit resolves it for us to the absolute path which it then gives Jetty.

The only other thing missing now is the actual test code. We want to achieve four things in that order:

1. starting the web server
2. starting the browser
3. stopping the browser
4. stopping the web server.

Note that it is important that the browser and web server are shut down cleanly in order to avoid memory issues or other difficult to trace errors, especially for the browser which uses native code.

The best places to do these steps are the `setUp` and `tearDown` methods of the test case class. Since we'll be going to need these steps for every test case, let's create a base class for our tests:

```
import junit.framework.TestCase;
import webappunit.Browser;
import webappunit.JRexBrowser;
import webappunit.JettyServer;
import webappunit.Server;

public abstract class GuiTestBase extends TestCase
{
    protected Server _server;
    protected Browser _browser;

    protected void setUp() throws Exception
    {
        _server = new JettyServer();
        _server.registerWebapp("/SpanishTrainer", ".");
        _server.start();

        _browser = new JRexBrowser();
        _browser.start();
    }

    protected void tearDown() throws Exception
    {
        _browser.close();
        _browser.destroy();
        _server.stop();
        _server.destroy();
    }
}
```

That is really all that is needed.

Of note is that since `webappunit` does not define a sub class of `junit.framework.TestCase`, we're free to choose the test base class. In this simple case, `TestCase` suffices, though.

Also notice that we register the web application under the name "SpanishTrainer" in the web server's root context, and that we tell Jetty that it can be found in the current directory.

Putting the start and stop of the web server and browser into `setUp/tearDown` means that for every test that is run, you'll get a new web server instance and a new browser instance. While this nicely isolates the tests from each other, it also has a severe impact on the performance of the test execution. If you are sure that the individual tests do not interfere with each other, then it is of course possible to use one server or browser instance for all tests.

This can be achieved for instance by moving the the start actions into the test case constructor and the stop actions into the `finalize` method.

Another thing is that you don't need to use an embedded web server. You can just as well test against a running web server. The difference is that you cannot start/stop this web server (at least not via `webappunit`), and that you cannot access server-side information such as the session. On the

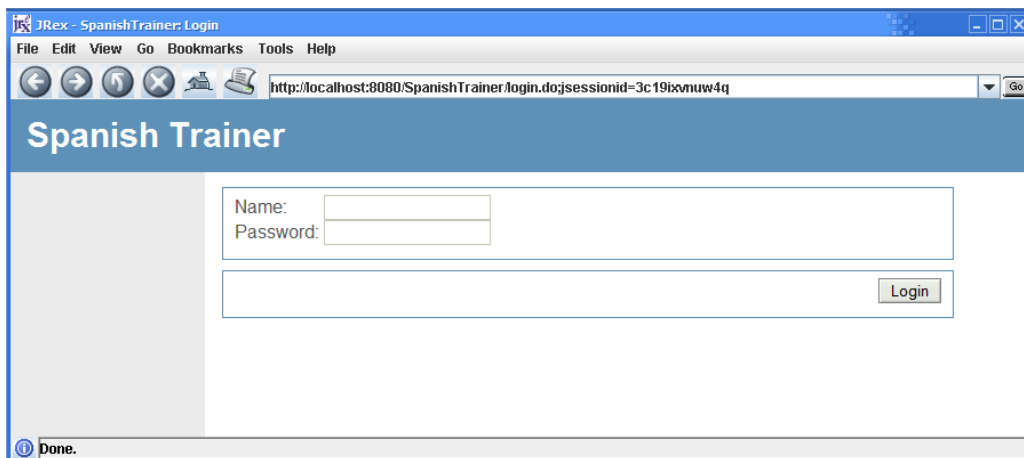
other hand, you can for instance write unit tests (in Java) for web applications written using PHP or .Net.

Now that we have the base class, we need a test case to run:

```
public class GuiTest0 extends GuiTestBase
{
    public void testSimple() throws InterruptedException
    {
        _browser.loadURIAndWaitForCompletion(
            "http://localhost:8080/SpanishTrainer/index.jsp");

        Thread.sleep(10000);
    }
}
```

If you run this, you should see a Swing window with some browser navigation controls, which – after the few seconds that it takes Jetty to start the web application - shows the login page of the SpanishTrainer web application (the `sleep` call makes sure that you have enough time to actually see the web page):



*SpanishTrainer login screen*

You may wonder why the method that we've invoked at the browser, is called `loadURIAndWaitForCompletion`. The reason is simple: loading a web page is asynchronous. The unit test is free to do other things in the meantime, but in this case we specifically asked to the browser to return only when the target page was loaded (though not necessarily every contained object such as images or audio files).

## Lesson 1: The login

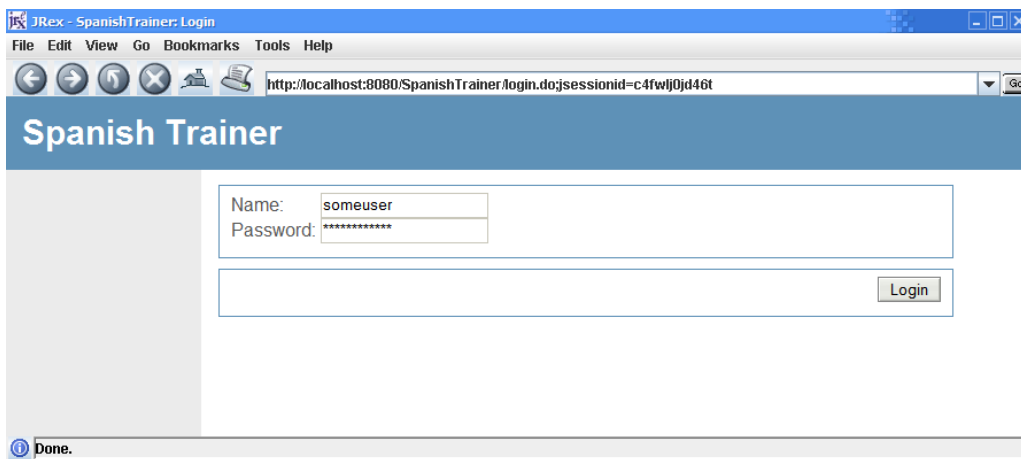
### Goal:

Create and run a unit test that tests the login by:

- starting the web application
- entering invalid data into the fields
- pressing the login button
- checking that the login page is shown again
- entering valid data into the fields
- pressing the login button again
- checking that now the index page is shown

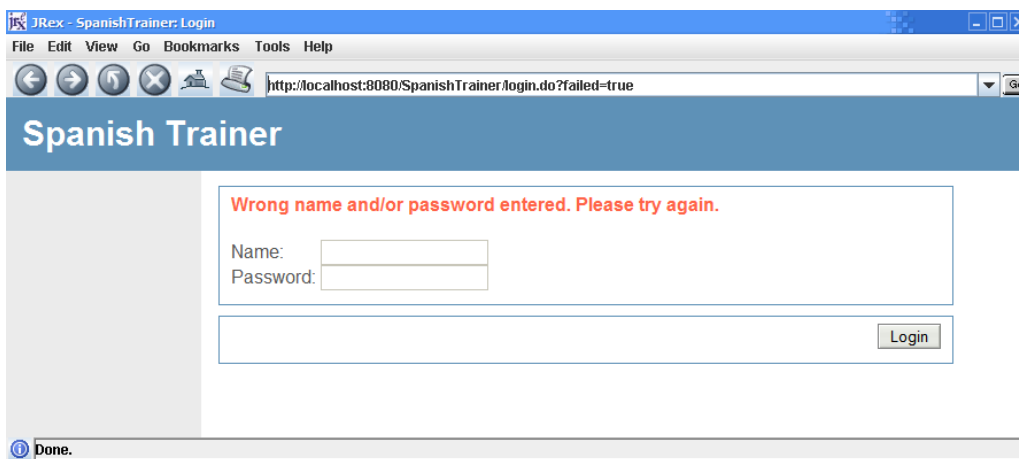
If you look closely at the above list, you'll notice that this is quite similar to a checklist that you'd use to perform a manual functional test of the web application. The reason is obvious: with webappunit we are using the browser just as a human tester would do, and so we perform the same actions in the same order. The only exception is that the unit test can also check the state of the web application on the server, something that a human tester usually is not able to do.

We've already seen the login page in the previous lesson:



*SpanishTrainer login screen*

After we've entered the wrong username or password, we arrive at a slightly changed login page that also includes a notice about the failed login try:



*SpanishTrainer login screen after failed attempt*

The address bar of this second login page now contains an additional parameter “failed=true”.

Ok, let's do that in a unit test. We start with the code from the previous lesson, and add a first check that we are really at the login page:

```
public class GuiTest1 extends GuiTestBase
{
    public void testLogin() throws InterruptedException
    {
        _browser.loadURIAndWaitForCompletion(
            "http://localhost:8080/SpanishTrainer/index.jsp");
        assertTrue(_browser.getCurrentURI().toString().startsWith(
            "http://localhost:8080/SpanishTrainer/login.do"));
    }
}
```

You'll notice that we don't use `assertEquals`s. The reason for this is that we arrive at the login page via a redirect from the `index.jsp` page that we originally asked for. Therefore, the login url contains the session id as an additional parameter (`jsessionid`). In this test we do not actually care about this parameter, though it might be useful as you can use it to retrieve a specific session from the web server.

This illustrates a general problem. For ascertaining that the browser has arrived at a specific page, the URL of the page might not be the best thing to test because it is subject of redirects and may contain additional parameters. A better way is to test for information in the page itself as we'll see soon.

Next the form fields. What we have to do is to get the two input fields – one for the user name and one for the password – from the browser, and then set their values. Conveniently, the web browser stores the web page as a tree of nodes called the Document Object Model, or short DOM. Consider for instance the snippet containing the two text fields. In the JSP page, it looks like this (surrounding elements removed):

```
<form method="POST" action="j_acegi_security_check">
  <table width="100%" border="0" cellspacing="0" cellpadding="2">
    <tr>
      <td valign="top" colspan="2">
        <fieldset>
          <table cellspacing="0" cellpadding="0">
            <tr>
              <td>
                <bean:message key="view.login.name"/>:&nbsp;
              </td>
              <td>
                <input type="text" name="j_username"/>
              </td>
            </tr>
            <tr>
              <td>
                <bean:message key="view.login.password"/>:&nbsp;
              </td>
              <td>
                <input type="password" name="j_password"/>
              </td>
            </tr>
          </table>
        </fieldset>
      </td>
      <td>
        <fieldset>
          <table cellspacing="0" cellpadding="0" width="100%">
            <tr>
              <td align="right">
                <input type="submit"
                  value="<bean:message key='view.login.submit'/>"/>
              </td>
            </tr>
          </table>
        </fieldset>
      </td>
    </tr>
  </table>
</form>
```

The names of the input fields, and the action that is executed when the form is submitted, are defined by Acegi (<http://acegisecurity.sourceforge.net>) which handles authentication and authorization for the sample web application.

The corresponding DOM tree follows this structure closely. It contains an element for each tag in

the resulting HTML - the above minus the `bean:message` Struts tags – shown again without the surrounding elements:

```

FORM [action='j_acegi_security_check', method='post']
  TABLE [width='100%', cellspacing='0', cellpadding='2', border='0']
    TBODY
      TR
        TD [valign='top', colspan='2']
          FIELDSET
            TABLE [cellspacing='0', cellpadding='0']
              TBODY
                TR
                  TD
                  TD
                    INPUT [type='text', name='j_username']
                TR
                  TD
                  TD
                    INPUT [type='password', name='j_password']
            
```

For each element type, there exists a corresponding class in the DOM API. Since we're using DOM Level 2 (<http://www.w3.org/TR/DOM-Level-2-HTML/>), the classes are called `org.w3c.dom.html2.HTMLInputElement` for a `INPUT` tag, `org.w3c.dom.html2.HTMLAnchorElement` for an `A` tag and so on. So for the above snippet, we get a `HTMLFormElement` with attributes `action` and `method`, that contains one child of type `HTMLTableElement` with attributes `width`, `cellspacing`, `cellpadding` and `border`, and so on.

In the test we could now traverse the DOM manually, but this is cumbersome and error-prone, and it would also tie the unit test to the layout. For instance, the SpanishTrainer web application uses Sitemesh (<http://www.opensymphony.com/sitemesh/>) for combining the menu with the content page. Greatly simplified, it inserts the HTML for the content page (the login page in this case) into a table cell in the HTML of the menu page. This combined HTML is then returned to the user. If the test now walks the DOM tree manually until reaching the form element, and the designer decides to change the layout/structure of the menu page, then we got a problem. A much better solution is to query the browser directly for interesting elements. The DOM elements have two attributes that we can use, the `id` and the `name`. The major difference between the two is that – at least theoretically in XHTML – the `id` value is unique in the document whereas there may be multiple elements with the same `name` value.

The net effect is that searching for the id is usually faster than for the name because the browser can cache the nodes by their id (if specified). Interestingly, you will find yourself searching for names more often than for ids. For instance, Struts almost always generates a name attribute in the resulting HTML. To make searching by name a bit faster, webappunit therefore adds a nodes-by-name cache that is filled when the page has been loaded.

So how do we now get these input fields ? We simply ask the browser for them:

```
HTMLInputElement usernameInput = (HTMLInputElement)
    _browser.getElementByName("j_username");
HTMLInputElement passwordInput = (HTMLInputElement)
    _browser.getElementByName("j_password");
```

Once we have the input elements, we can simply set their values:

```
usernameInput.setValue("someuser");
passwordInput.setValue("somepassword");
```

Because filling input elements is a common task, webappunit provides a shortcut method for this:

```
_browser.setFormInputByName("j_username", "someuser");
_browser.setFormInputByName("j_password", "somepassword");
```

Now we only need to submit the form and wait for the result. Since the submit button is also an input element, we could retrieve it just like the above input elements. However the submit button has neither id nor name:

```
<input type="submit" value="<bean:message key='view.login.submit' />" />
```

We now could search for the element in the DOM whose value is “Login”. But unfortunately this text is internationalized (it is set with the help of the `bean:message` tag). Luckily, there's only one submit button on the page, and webappunit provides a shortcut for this case (we will see how to deal with multiple buttons, and how to utilize internationalized text in a later lesson):

```
_browser.getSubmitButton().click();
_browser.waitForLoadCompletion();
```

Again, we wait for the loading of the target page. This is important because only if the HTML page has been loaded fully, the DOM tree is guaranteed to be accessible. Before that, you may get a partial DOM tree, or you may also get a null back – which one, will be decided by the browser. Since we entered nonsense data, we should be back at the login page. Lets check this, and then do the same things again, but this time with proper values:

```
assertEquals("http://localhost:8080/SpanishTrainer/login.do?failed=true",
    _browser.getCurrentURI().toString());

_browser.setFormInputByName("j_username", "someuser");
_browser.setFormInputByName("j_password", "abc");
_browser.getSubmitButton().click();
_browser.waitForLoadCompletion();

assertEquals("http://localhost:8080/SpanishTrainer/index.jsp",
    _browser.getCurrentURI().toString());
```

After we've entered the correct values, we expect to end up at the index page. For reference, here's the complete test:

```
public class GuiTest1 extends GuiTestBase
{
    public void testLogin()
    {
        _browser.loadURIAndWaitForCompletion(
            "http://localhost:8080/SpanishTrainer/index.jsp");

        assertTrue(_browser.getCurrentURI().toString().startsWith(
            "http://localhost:8080/SpanishTrainer/login.do"));

        _browser.setFormInputByName("j_username", "someuser");
        _browser.setFormInputByName("j_password", "somepassword");
        _browser.getSubmitButton().click();
        _browser.waitForLoadCompletion();

        assertEquals(
            "http://localhost:8080/SpanishTrainer/login.do?failed=true",
            _browser.getCurrentURI().toString());

        _browser.setFormInputByName("j_username", "someuser");
        _browser.setFormInputByName("j_password", "abc");
        _browser.getSubmitButton().click();
        _browser.waitForLoadCompletion();

        assertEquals("http://localhost:8080/SpanishTrainer/index.jsp",
            _browser.getCurrentURI().toString());
    }
}
```

If you run this, you should see how the login page loads, and then the input fields on this page are magically filled with values. Then the login page re-loads and the fields are filled again with different values. Finally, we get to a different page with a menu at the side.

The nice thing about this test is that you can actually read it and understand it as it is. Of course, you would have to know what the names of the elements mean, but this should be no problem when the web application uses a proper naming scheme.

But wait, we can do more! How about we check at the server, or more precisely the server's session, that the user really has been logged in ?

Accessing the session is easy: we simply ask the server. Because there is only one user of the web server, we do not even need a session id, we can ask for the single/first session:

```
HttpSession session = _server.getSession();
```

Now we happen to know that Acegi puts a `SecureContext` instance into the session using the name `ACEGI_SECURITY_CONTEXT`, and this instance contains the authentication information, such as an object representing the authenticated user (a so-called principal).

When written in code, this would look like:

```
HttpSession      session = _server.getSession();
SecureContext    context = (SecureContext)session.getAttribute(
                    HttpSessionContextIntegrationFilter.
                    ACEGI_SECURITY_CONTEXT_KEY);
Authentication    auth    = context.getAuthentication();
AuthenticatedUser user    = (AuthenticatedUser)auth.getPrincipal();

assertEquals("someuser",
             user.getUsername());
```

But `webappunit` provides another shortcut (again). By integrating OGNL (<http://www ognl.org/>), we can query for values in the session using OGNL expressions. In this case, we can simply write

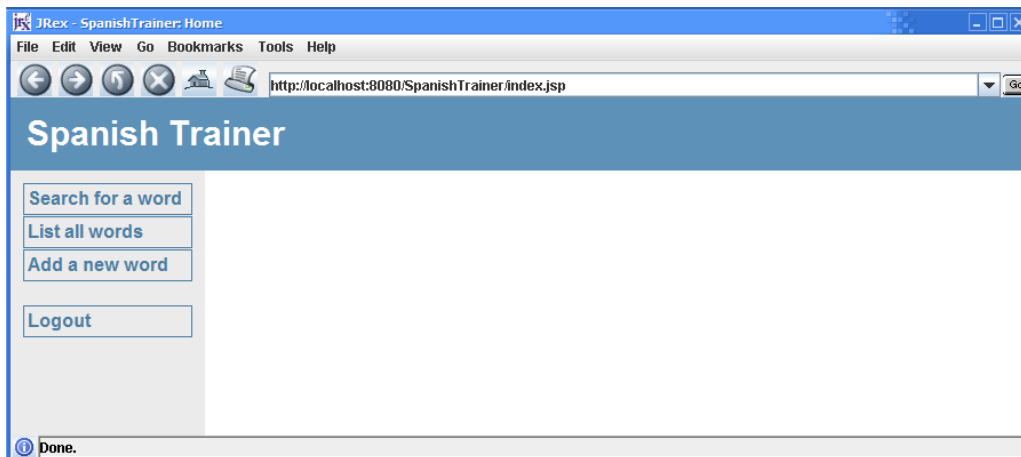
```
assertEquals(
    "someuser",
    _server.extractValueFromSession(
        "ACEGI_SECURITY_CONTEXT.authentication.principal.username"));
```

which looks a lot better and is quite sufficient for most situations. Not to mention that we don't have to use Acegi's API.

While it might or might not be useful to query the session for the name of the authenticated user, this example shows that it is quite easy to check for values in the session within an unit test. Perhaps not so obvious is that due to the fact that we use an embedded web server, we can actually use the same classes as the web application. This of course provides the benefit that we can use these classes to extract session values as shown in the above code snippet. However you should be aware that using application classes can introduce unwanted side-effects and modify the execution of the web application in undesired ways.

## Lesson 2: Clicking a link

Now that we can login into the web application, it is time that we explore it. The individual parts of the web application are reached by the menu on the left side:



*SpanishTrainer menu*

### Goal:

After performing a login, click the link in the menu on the left side for listing all words, and then assert that the corresponding page has been loaded.

In order to access, say, the overview page showing all words in the system, we have to click the second link. Basically, this involves the same steps as clicking a submit button: asking the browser for the link object, then clicking it and waiting for the loading of the target page to finish. Getting the link DOM object would be easy if it would have an id or name, but usually links only have a href attribute and the visible text or image. Now we could search for the element whose href attribute contains a given value:

```
HTMLAnchorElement anchor =  
    (HTMLAnchorElement)_browser.getElementByAttributeValue("href",  
        "http://localhost:8080/SpanishTrainer/listAllWords.do");
```

Webappunit provides a shortcut for this

```
HTMLAnchorElement anchor = _browser.getAnchorByHrefValue(  
    "http://localhost:8080/SpanishTrainer/listAllWords.do");
```

One problem with this is that we hard-code the link target URL. A better way would be to check for

the text within the link instead of the target URL:

```
HTMLAnchorElement anchor = _browser.getAnchorByText("List all words");
```

This method basically checks the inner contents of all anchor elements and returns the first whose text matches the given text.

While this looks better, it still has one problem: it ties the test to the specific target language, English in this case. The web application however uses application properties for internationalization. If we look into the menu JSP we find:

```
<a href="<html:rewrite action='/initSearchForWord' />">
  <bean:message key="view.menu.searchForWord" />
</a>
```

So, in order to test in a language-independent way, we need to access the resource bundle and retrieve the message with the key `view.menu.searchword`.

Since we're testing within the context of the web application, the properties files for the resource bundle (`application.properties`) are already in the class path, so we can simply access the message via Java's `ResourceBundle` class:

```
HTMLAnchorElement anchor = _browser.getAnchorByText(
    ResourceBundle.getBundle("application")
        .getString("view.menu.searchForWord"));
```

As our sample web application accesses the resource bundle itself, it conveniently provides a helper class for this task:

```
HTMLAnchorElement anchor = _browser.getAnchorByText(
    ResourceBundleHelper.getMessage("view.menu.showAllWords"));
```

Clicking it however is not so easy. If you remember from the previous lesson, clicking a submit button is done by calling the `click()` method on it. Unfortunately, there is no such thing for anchors. For whatever reason, the W3C forgot to add such a method to the `HTMLAnchorElement` class which means that we have to resort to other means to perform our click. DOM level 2, with which we work, includes an event model that gives a user of the DOM two abilities:

- listening for events
- creating and dispatching standard events such as mouse events or key events

You can read more about the event model here: <http://www.w3.org/TR/DOM-Level-2-Events/events.html>.

Basically what we need to do is to find the event target, create a mouse event, and dispatch this event to the event target. We already have the event target: our anchor.

Creating such an event unfortunately is not cleanly specified by the DOM specification. While it defines a `createEvent` method for the `Document` interface, the official Java binding does not include it. Likewise, while DOM level 2 defines the `EventTarget` interface which we can use to dispatch the event, it does not require any nodes to implement it. Hence, we have to use the proprietary browser API.

In order to avoid tying the unit tests to a particular browser, `webappunit` therefore provides a wrapper method that performs this click. For `JRex` this wrapper looks like this:

```
public void click(HTMLElement element, int buttonNumber)
{
    if (!(element instanceof JRexNodeImpl))
    {
        throw new BrowserException("Cannot click on the given node");
    }

    JRexNodeImpl      realNode      = (JRexNodeImpl)element;
    JRexHTMLDocumentImpl dom        = (JRexHTMLDocumentImpl)getDOM();
    JRexEventTargetImpl clickEvent = (JRexEventTargetImpl)
        realNode.getEventTarget();
    JRexMouseEventImpl mouseEvent = (JRexMouseEventImpl)
        dom.getDocumentEvent()
            .createEvent("MouseEvents");
    JRexDocumentViewImpl docView    = (JRexDocumentViewImpl)
        dom.getDocumentView();

    mouseEvent.initMouseEvent(
        "click",           // type
        true,              // can bubble ?
        false,             // can be canceled ?
        docView.getDefaultView(), // the view
        1,                 // detail (?)
        0, 0,             // screen x, y
        0, 0,             // client x, y
        false,             // control key ?
        false,             // alt key ?
        false,             // shift key ?
        false,             // meta key ?
        (short)buttonNumber, // which button
        clickEvent);

    clickEvent.dispatchEvent(mouseEvent);
}
```

It basically creates a mouse event object, initializes it and then dispatches it to the given element. Our unit test code then looks like this:

```
HTMLAnchorElement anchor = _browser.getAnchorByText(  
    ResourceBundleHelper.getMessage("view.menu.showAllWords"));  
  
_browser.click(anchor, 0);  
_browser.waitForLoadStart();
```

The `waitForLoadStart` call ensures that the tests waits for the start of the loading of the target page. In our case this is needed because the link does not stay within the page but leads to an new one.

This is a common task, so `webapunit` again provides a convenience method:

```
_browser.clickAnchorByText(  
    ResourceBundleHelper.getMessage("view.menu.showAllWords"), true);  
  
_browser.waitForLoadCompletion();
```

The second argument ensures that `waitForLoadStart` is called. Since we're interested in the target page, we also wait for the loading to finish.

We could assert that we arrived at the page showing all words, in the same way as in lesson 1: by comparing the current URI with the expected one. But there is a better way that is independent of the web server environment and any redirects or additional parameters. We check data embedded into the page itself. The element that seems to be best suited for this purpose is the meta tag because it is intended to provide information about the web page that the server or browser then can utilize for some purpose.

The page that we loaded in the above snippet contains a meta tag called `PageName` that we should use for our purpose:

```
<head>  
  <meta name="PageName" content="listAllWords"/>  
  <title>  
    <bean:message key="view.listAllWords.title"/>  
  </title>  
</head>
```

Actually this is slightly more involved because `Sitemesh` – which we're using to compose the menu and the content pages – ignores the header of the content page. So in order to make the meta tags – and the title – available in the resulting page, the menu JSP page contains something like this:

```
<decorator:usePage id="embeddedPage"/>
<head>
  <meta name="PageName"
        content="${embeddedPage.properties['meta.PageName']}" />
  <title>
    <decorator:title default="SpanishTrainer"/>
  </title>
</head>
```

Testing for it simply means to retrieve the meta element by its name, and check its content attribute:

```
assertEquals("listAllWords",
            _browser.getElementByName("PageName").getAttribute("content"));
```

As always, there is a helper method for this:

```
assertEquals("listAllWords",
            _browser.getMetaElement("PageName").getContent());
```

Our complete test now looks like:

```
public class GuiTest2 extends GuiTestBase
{
    public void testLinkAndBack()
    {
        _browser.loadURIAndWaitForCompletion(
            "http://localhost:8080/SpanishTrainer/index.jsp");
        _browser.setFormInputByName("j_username", "someuser");
        _browser.setFormInputByName("j_password", "abc");
        _browser.getSubmitButton().click();
        _browser.waitForLoadCompletion();

        _browser.clickAnchorByText(
            ResourceBundleHelper.getMessage("view.menu.showAllWords"),
            true);
        _browser.waitForLoadCompletion();

        assertEquals("listAllWords",
            _browser.getMetaElement("PageName").getContent());
    }
}
```

## ***Detour: Logging***

When a test case fails (or if it runs successfully even though you're sure that it shouldn't) it is helpful to get additional info from the participating components. For this purpose, most of the libraries utilize commons-logging (<http://jakarta.apache.org/commons/logging>) or log4j directly (<http://logging.apache.org>). In order to enable it, you place a properties file called `log4j.properties` into the classpath where it will be picked up automatically. A typical file would look like:

```
log4j.rootCategory=WARN, CONSOLE, FILE

log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%p - %C{1}.%M(%L) | %m%n

log4j.appender.FILE=org.apache.log4j.RollingFileAppender
log4j.appender.FILE.File=SpanishTrainer.log
log4j.appender.FILE.Append=false
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=%d - %p - %C{1}.%M(%L) | %m%n

#log4j.logger.org.apache.ojb=WARN
#log4j.logger.org.apache.ddlutils=WARN
#log4j.logger.org.springframework=WARN
#log4j.logger.net.sf.acegisecurity=WARN
#log4j.logger.org.mortbay=WARN
log4j.logger.webappunit=INFO
```

Here, two logging targets are defined, `CONSOLE` and `FILE`, which log to `stdout` and a file called `SpanishTrainer.log`, respectively.

Also, the default log level is set to `WARN`, meaning that only warnings and errors will be logged. The last line in the file overrides this for the package `webappunit` and its subpackages, and sets the logging level for these packages to `INFO`.

Running the test with these logging settings produces this output:

```
INFO - JettyServer.<init>(48) | Jetty server created
INFO - JettyServer.registerWebapp(61) | Registering the web application at
      . for context path /SpanishTrainer
INFO - JettyServer.registerWebapp(87) | Web application for context path
      /SpanishTrainer registered
INFO - JettyServer.start(105) | Starting the Jetty server
INFO - JettyServer.start(112) | Jetty server started
INFO - JRExBrowser.<init>(75) | Initializing JREx
***** JRExL inited *****
***** JREX-Logging disabled *****
INFO - JRExBrowser.<init>(92) | JREx initialized
INFO - JRExBrowser.start(108) | Creating JREx browser instance
INFO - JRExBrowser.start(184) | JREx browser instance created
INFO - JRExBrowser.loadURI(312) | Loading
      'http://localhost:8080/SpanishTrainer/index.jsp' into the browser
INFO - JRExBrowser.waitForLoadCompletion(271) | Waiting for the document
      loading to complete
INFO - JRExBrowser.ensureDOM(421) | Getting DOM of document
INFO - JRExBrowser.ensureDOM(429) | DOM of document retrieved
INFO - JRExBrowser.waitForLoadCompletion(286) | Document loading has
      completed
INFO - JRExBrowser.waitForLoadCompletion(271) | Waiting for the document
      loading to complete
INFO - JRExBrowser.ensureDOM(421) | Getting DOM of document
INFO - JRExBrowser.ensureDOM(429) | DOM of document retrieved
INFO - JRExBrowser.waitForLoadCompletion(286) | Document loading has
      completed
INFO - JRExBrowser.waitForLoadStart(245) | Waiting for the document loading
      to start
INFO - JRExBrowser.waitForLoadStart(260) | Document loading has started
INFO - JRExBrowser.waitForLoadCompletion(271) | Waiting for the document
      loading to complete
INFO - JRExBrowser.ensureDOM(421) | Getting DOM of document
INFO - JRExBrowser.ensureDOM(429) | DOM of document retrieved
INFO - JRExBrowser.waitForLoadCompletion(286) | Document loading has
      completed
INFO - JRExBrowser.close(195) | Closing JREx browser instance
INFO - JRExBrowser.close(207) | JREx browser instance closed
INFO - JRExBrowser.destroy(218) | Shutting JREx down
INFO - JRExBrowser.destroy(232) | JREx shut down
INFO - JettyServer.stop(257) | Stopping the Jetty server
INFO - JettyServer.stop(264) | Jetty server stopped
INFO - JettyServer.destroy(282) | Destroying the Jetty server
INFO - JettyServer.destroy(289) | Jetty server destroyed
```

For JREx, you'll have to download the library version that contains logging from their website (<http://jrex.mozdev.org/releases.html>). Be warned though that JREx produces **a lot** of logging output, which is why webappunit rather includes the version without logging.

## Lesson 3: Database-driven testing

Most web applications deal in one way or the other with databases. Usually, the business objects have some persistent representation in the database, and additional data is retrieved by querying the database and combining existing information.

Therefore, one of the more important aspects of testing web applications is the proper database initialization for the test, and also the access to the database during the test.

For instance, web applications often contain workflows that change depending on what information are contained in the database, and on the environment. A typical event ticket booking system obviously depends upon the current date and the information about upcoming events stored in the database.

In order to test such workflows, a functional test must setup the database with data in such a way that the web application exhibits the expected behavior. For instance, for a ticket booking system, a test would determine the current date and insert events for the next weeks, so that the tested web application then find upcoming events and can offer tickets for them.

There are a lot of (technical and organizational) ways to setup the database for a test. For instance:

1. Static setup for all tests using fixed data that is inserted before running the tests. This is the simplest technical way, as it does not affect the unit tests. But defining the data properly requires care because the data sets for the individual tests intermix. The same applies to the tests that change the database through the web application, or otherwise they affect other tests and thus falsify the test results.
2. The test may insert the data via a direct connection to the database during its execution. This allows to limit the effect on the database to the scope of the test, provided that the test cleans this data properly after execution. Problems may arise however if the web application relies on some caching or clustering mechanisms though they can possibly be turned off for the tests (they shouldn't affect the functioning of the web application).
3. A test can even utilize the database layer of the actual web application. As we've seen, tests can use classes and objects of the running web application. So it is technically not a problem to access the database services of the web application and manipulate the database via them. However, this naturally leads to white box tests as the test developer needs to know the internals of the database layer of the web application in order to be able to ensure the isolation of the tests from each other. This makes this method somewhat difficult to use.

Functional tests typically utilize methods 1 or 2, or a combination of both. In fact, in the previous test we already used method 1. If you remember, in lesson 0 we discussed the Ant targets that needed to be executed before a test can be run. One of these targets was the `test-data` target whose purpose is to setup the database structure and to insert some basic data into this new database.

**Goal:**

Insert a verb into the database. Then login into the web application, search for the verb and assert that its information are correctly presented in the detail page.

We will now combine this with method 2. This new test will insert data directly into the database while the web application is running, and then check that the web application picks it up.

To insert data into the database you can use whatever does the job, such as plain JDBC or an ORM library. For this lesson we will use DdlUtils which was already utilized in the Ant tasks for setting up the database and inserting the basic data. This allows us to re-use the database model that was auto-created by XDoclet during the building of the web application.

Of course, you could also use something like DbUnit (<http://dbunit.sourceforge.net>). The DdlUtils model is richer though, and allows for further customization. For instance, you can define converters that convert between the textual representation in the data file (XML) into the SQL datatype. This is especially useful for things like dates and numeric values. Also, as you will see, the DdlUtils API is quite simple and easy to use.

The first thing in setting up the database is of course gaining access to it, e.g. by setting up a JDBC connection via a data source. Now, we could simply hardcode the connection information (driver name, JDBC url etc.) into the test. But this is not a good practice as it unnecessarily ties the tests to the specific test setup. Rather, we will access the same database setup that the web application uses.

This next section briefly explains how we can use Spring in our tests in order and to use the web application's configuration data. Our basic goal is to let Spring create and configure an instance of the `DatabaseHelper` class which contains the functionality that the test require for manipulating the database. This instance is then wired by Spring into the test case so that we can use it there.

You can simply skip this section if you're not interested in the details of this process. Important is that the unit test instance has access to a `DatabaseHelper` instance when the tests are run, and that this instance is connected to the same database that the web application uses.

***Detour: Using Spring in the test case***

As mentioned before, the web application uses Spring for maintaining the connection to the database and for the transaction handling. The basic information about the connection is contained in the file `jdbc.properties`, which looks something like this:

```
jdbc.driverClassName=org.postgresql.Driver
jdbc.username=root
jdbc.password=somepassword
jdbc.level=3.0
jdbc.url=jdbc:postgresql://localhost/spanishtrainer
jdbc.dbtype=postgresql
```

The other relevant configuration file is called `applicationContext.xml`. It contains the declarations of the objects that Spring shall create and maintain, among them the Struts actions, the business and DAO objects, and the data source. Within it, the settings from the `jdbc.properties` file are imported and used to create a data source:

```
<bean id="propertyConfigurer"
      class="org.springframework.beans.factory.config
            .PropertyPlaceholderConfigurer">
  <property name="location">
    <value>classpath:jdbc.properties</value>
  </property>
</bean>

<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName">
    <value>${jdbc.driverClassName}</value>
  </property>
  <property name="url">
    <value>${jdbc.runtime.url}</value>
  </property>
  <property name="username">
    <value>${jdbc.username}</value>
  </property>
  <property name="password">
    <value>${jdbc.password}</value>
  </property>
</bean>
```

The data source is then globally available under the name “`dataSource`”. The DAO objects for instance derive from a Spring class which automatically uses a bean of that name.

In order to use this setup in our tests, we have to do two things:

- Make Spring aware of the test case.
- Create and initialize a `DatabaseHelper` object via Spring, and wire it into our test case.

The first one is achieved by inheriting our test case from the Spring-provided class `AbstractDependencyInjectionSpringContextTests`. Since we might need the setup in more than one test class, we derive a class similar to the `GuiTestBase` class that we defined earlier:

```
import
    org.springframework.test.AbstractDependencyInjectionSpringContextTests;
import webappunit.Browser;
import webappunit.JRexBrowser;
import webappunit.JettyServer;
import webappunit.Server;

public abstract class SpringifiedGuiTestBase extends
    AbstractDependencyInjectionSpringContextTests
{
    protected Server _server;
    protected Browser _browser;

    protected String[] getConfigLocations()
    {
        return new String[] { "applicationContext-test.xml" };
    }

    protected void setUp() throws Exception
    {
        _server = new JettyServer();
        _server.registerWebapp("/SpanishTrainer", ".");
        _server.start();

        _browser = new JRexBrowser();
        _browser.start();
    }

    protected void tearDown() throws Exception
    {
        _browser.close();
        _browser.destroy();
        _server.stop();
        _server.destroy();
    }
}
```

There are only two differences to the `GuiTestBase` class that we defined earlier:

- We have to tell Spring where to find the configuration file. This is done by defining the `getConfigLocations` method which returns the classpath locations of the configuration files that Spring shall use for setting up the test class.
- The `AbstractDependencyInjectionSpringContextTests` class redefines the methods `setUp` and `tearDown` and prevents sub classes from overriding them. Therefore, we have to move our start and stop actions for the web server and browser to the Spring-provided `setUp` and `tearDown` methods.

A concrete test class then inherits from this base class and defines setters for the objects that it is interested in. In our case, we're interested in a `DatabaseHelper` instance, so the test class would look like this:

```
public class GuiTest3 extends SpringifiedGuiTestBase
{
    private DatabaseHelper _databaseHelper;

    public void setDatabaseHelper(DatabaseHelper dbHelper)
    {
        _databaseHelper = dbHelper;
    }

    public void testDefinedData()
    {
        ...
    }
}
```

Spring then auto-wires the test class, i.e. it calls this method with the single `DatabaseHelper` instance defined in the configuration file. This `applicationContext-test.xml` file looks like this:

```
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans default-autowire="no"
    default-lazy-init="false"
    default-dependency-check="none">

    <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config
            .PropertyPlaceholderConfigurer">
        <property name="location">
            <value>classpath:jdbc.properties</value>
        </property>
    </bean>

    <bean name="databaseHelper"
        class="spanishtrainer.dao.DatabaseHelper">
        <property name="dataSource">
            <bean class=
                "org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName">
                    <value>${jdbc.driverClassName}</value>
                </property>
                <property name="url">
                    <value>${jdbc.url}</value>
                </property>
                <property name="username">
                    <value>${jdbc.username}</value>
                </property>
                <property name="password">
                    <value>${jdbc.password}</value>
                </property>
            </bean>
        </property>
        <property name="modelFiles">
            <list>
                <value>classpath:ojbcore-schema.xml</value>
                <value>classpath:project-schema.xml</value>
            </list>
        </property>
        <property name="databaseType">
            <value>${jdbc.dbtype}</value>
        </property>
    </bean>

</beans>
```

As you can see, we use the same database setup as the web application. However, we're not reusing its configuration file because it contains lots of stuff that we don't need in our test, or that might even fail to work. For instance, the test is not running inside the web application, thus it is not possible to access some objects managed by Spring such as the Struts actions, in this way. Other objects however could be accessed, such as the DAO objects and probably the business services (if not dependent upon being run in a web application).

The best way to make such objects available to the test via Spring, is to split the Spring configuration file into multiple files, e.g. one for the DAO objects, one for the business objects and so forth. As we've already seen, you can tell Spring to load multiple configuration files in the test. The same is possible for the web application (in fact the web application already does this: for technical reasons the struts actions reside in their own configuration file, `applicationContext-struts.xml`).

Spring will initialize the `DatabaseHelper` instance with the data source, the database type value (which we'll use for `DdlUtils`), and the database schemas, `ojbcore-schema.xml` (containing OJB internal tables) and `project-schema.xml` (containing the tables for the web application). The `DatabaseHelper` instance defines setter methods for these properties:

```
public class DatabaseHelper
{
    private String      _databaseType;
    private DataSource  _dataSource;
    private File[]      _modelFiles;
    private Database    _model;

    public void setDatabaseType(String databaseType)
    {
        _databaseType = databaseType;
    }

    public void setDataSource(DataSource dataSource)
    {
        _dataSource = dataSource;
    }

    public void setModelFile(Resource modelFile) throws IOException
    {
        _modelFiles = new File[] { modelFile.getFile() };
    }

    public void setModelFiles(Resource[] modelFiles) throws IOException
    {
        _modelFiles = new File[modelFiles.length];
        for (int idx = 0; idx < modelFiles.length; idx++)
        {
            _modelFiles[idx] = modelFiles[idx].getFile();
        }
    }

    ...
}
```

The interesting stuff happens in the rest of this class:

```
public class DatabaseHelper
{
    ...

    private Platform getPlatform()
    {
        Platform platform =
            PlatformFactory.createNewPlatformInstance(getDatabaseType());

        platform.setDataSource(_dataSource);
        platform.getPlatformInfo().setUseDelimitedIdentifiers(false);
        return platform;
    }

    private Database getModel()
    {
        if (_model == null)
        {
            DatabaseIO dbIO = new DatabaseIO();

            _model = new Database();
            for (int idx = 0; idx < _modelFiles.length; idx++)
            {
                _model.mergeWith(dbIO.read(_modelFiles[idx]));
            }
        }
        return _model;
    }

    public void insertData(String dataAsXml) throws IOException,
        SAXException
    {
        DataReader reader = new DataReader();
        Database model = getModel();

        reader.setModel(model);
        reader.setSink(new DataToDatabaseSink(getPlatform(), model));
        reader.parse(new StringReader(dataAsXml));
    }

    public void deleteObject(String tableName, Object pk)
    {
        Table table = getModel().findTable(tableName);
        DynaBean bean = getModel().createDynaBeanFor(table);
        String pkColumn = table.getPrimaryKeyColumns()[0].getName();

        bean.set(pkColumn, pk);
        getPlatform().delete(getModel(), bean);
    }

    public void deleteObjects(String tableName, Object[] pks)
    {
        Table table = getModel().findTable(tableName);
        DynaBean bean = getModel().createDynaBeanFor(table);
        String pkColumn = table.getPrimaryKeyColumns()[0].getName();

        for (int idx = 0; idx < pks.length; idx++)
        {
            bean.set(pkColumn, pks[idx]);
            getPlatform().delete(getModel(), bean);
        }
    }
}
```

We're using DdlUtils here to insert data in XML form into the database, and to delete objects by their primary key values. There are two relevant classes for this:

- The Platform class is the facade provided by DdlUtils to perform database-related actions, such as creating the database, or deleting objects from it (used in the deleteObject and deleteObjects methods in the DatabaseHelper class).
- The Database class and the related classes (Table, Column, ...) represent the database schema in a vendor-independent way. An instance of it is read from the XML file that was auto-generated before by XDoclet. This model is then used by DdlUtils for performing the actions.

For instance, a snippet of the database schema for our sample web application looks like:

```
<database name="SpanishTrainer">
  ...
  <table name="verb">
    <column name="id"
      type="INTEGER"
      primaryKey="true"
      required="true" />
    <column name="text"
      type="VARCHAR"
      size="254" />
    ...
  </table>
  ...
  <table name="conjform">
    <column name="id"
      type="INTEGER"
      primaryKey="true"
      required="true" />
    <column name="verbid"
      type="INTEGER" />
    <column name="type"
      type="INTEGER" />
    <column name="person"
      type="INTEGER" />
    <column name="text"
      type="VARCHAR"
      size="254" />
    <foreign-key foreignTable="verb">
      <reference local="verbid" foreign="id"/>
    </foreign-key>
  </table>
```

The `data.xml` file that we filled our database with earlier on, contains the initial data defined with corresponding entries:

```
<data>
  ...
  <verb id="17"
    origin="Avance, Nivel Elemental, Unidad preliminar"
    text="decir"
    orthochange="1"
    gerund="diciendo"
    participle="dicho"/>
  <translations id="33"
    termid="17"
    language="1"
    text="sagen"/>
  <translations id="34"
    termid="17"
    language="2"
    text="to say"/>
  <conjform id="3"
    verbid="17"
    type="6"
    person="1"
    text="digo"/>
  <conjform id="4"
    verbid="17"
    type="6"
    person="2"
    text="dices"/>
  ...
</data>
```

This is the same format that we will be using now for our test. There is also an Ant task provided by DdlUtils that generates a DTD for the data files.

### ***Back on track: Writing the test***

After this somewhat longish detour, we can finally write our test. If you remember, our test case now has access to a `DatabaseHelper` instance with which it can insert data into the database, or delete it from the database. Lets put that to a use by adding a new verb to the database:

```

private static final String TEST_DATA =

"<data>" +
" <verb id='100' origin='test' text='dar' />" +
" <translations id='100' termid='100' language='1' text='geben' />" +
" <translations id='101' termid='100' language='2' text='to give' />" +
" <conjform id='100' verbid='100' type='6' person='1' text='doy' />" +
" <conjform id='101' verbid='100' type='20' person='1' text='diere' />" +
" <conjform id='102' verbid='100' type='20' person='2' text='dieres' />" +
" <conjform id='103' verbid='100' type='20' person='3' text='diere' />" +
" <conjform id='104' verbid='100' type='20' person='4' text='diéremos' />" +
" <conjform id='105' verbid='100' type='20' person='5' text='diereis' />" +
" <conjform id='106' verbid='100' type='20' person='6' text='dieren' />" +
"</data>";

public void testDefinedData() throws IOException, SAXException
{
    _databaseHelper.insertData(TEST_DATA);
    ...
}

```

This effectively inserts a new Verb, *dar*, into the database along with two translations and its seven irregular conjugation forms. We simply define the data XML in string form within the test case, but it could just as well have come from an external file or some other source.

DdlUtils will automatically ensure that the foreignkey order is maintained (*conjform* has a foreign key constraint referencing *verb*), though it is not required in this case because we defined them in the correct order. Similarly, if the database creates the primary key values, e.g. because of identity/auto-increment columns or sequences, DdlUtils will make sure that the foreign key in the database uses the real primary key values, not the ones specified in the XML file. This allows to specify relations between the entries in the data XML file independent of how the real primary key values in the database are created.

A test case that manipulates the database should be well behaved and clean up after the test is finished, regardless of whether the test was successful or not. Since in our case the application defines the primary key values, they will be equal to the values that we specified in the original data XML. This makes it easy to delete the rows from the database, we only have to make sure that we remove them in the right order (otherwise we would violate the foreign key constraints):

```
public void testDefinedData() throws IOException, SAXException
{
    _databaseHelper.insertData(TEST_DATA);

    try
    {
        ...
    }
    finally
    {
        _databaseHelper.deleteObjects("conjform",
                                      new String[] { "100", "101", "102",
                                                    "103", "104", "105",
                                                    "106" });
        _databaseHelper.deleteObjects("translations",
                                      new String[] { "100", "101" });
        _databaseHelper.deleteObject("verb", "100");
    }
}
```

Now we can add the actual testing calls. We are going to perform these steps:

1. Login
2. Go to the search page
3. Enter our new word into the search field and click the submit button
4. Assert that we have arrived at the detail page
5. Check the contents of the detail page

We have already talked about the first four steps in the previous lessons, so there is nothing new so far:

```

_browser.loadURIAndWaitForCompletion(
    "http://localhost:8080/SpanishTrainer/index.jsp");
_browser.setFormInputByName("j_username", "someuser");
_browser.setFormInputByName("j_password", "abc");
_browser.getSubmitButton().click();
_browser.waitForLoadCompletion();

_browser.clickAnchorByText(
    ResourceBundleHelper.getMessage("view.menu.searchForWord"),
    true);
_browser.waitForLoadCompletion();

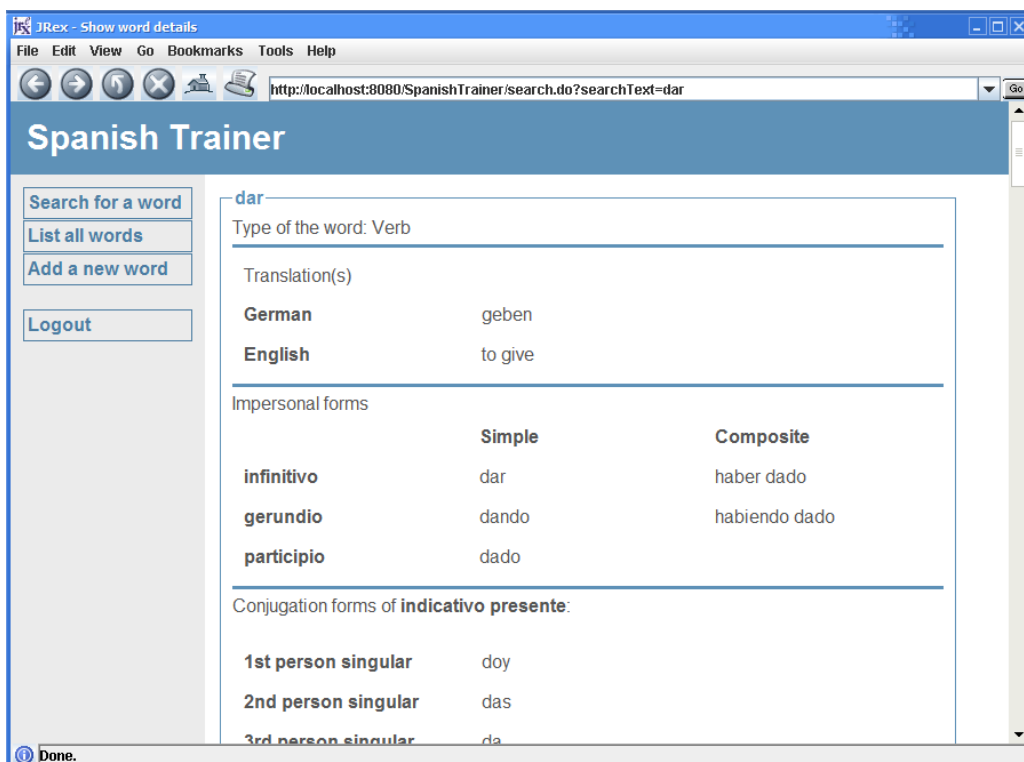
assertEquals("search",
    _browser.getMetaElement("PageName").getContent());

_browser.getFormInputByName("searchText").setValue("dar");
_browser.getSubmitButton().click();
_browser.waitForLoadCompletion();

assertEquals("showWord",
    _browser.getMetaElement("PageName").getContent());

```

The detail pane presents all information about the verb that the system contains, including the translations and the conjugation forms:



*SpanishTrainer detail page for the verb dar*

For checking the content we, as always, have the option of manually walking the DOM for the page, or using `id` or `name` attributes to access specific elements in the page. For instance, we can check the contents of the `legend` element which should contain the word that we searched for. Since there is only one legend on the page, we can use a helper method in `webappunit`'s class `DOMHelper` that retrieves the first element of a given type:

```
HTMLLegendElement legend =
    (HTMLLegendElement)DOMHelper.getFirstChildOfType(
        _browser.getDOM(), "legend", true);

assertEquals("dar",
    legend.getTextContent().trim());
```

The `getTextContent` method is a DOM level 3 method that returns the complete textual content of the node and its children. However since its defined in DOM level 3, it is usually only available when using Java 5. Therefore, `webappunit` provides a method `DOMHelper.getTextContent(Node)` that performs the same function for DOM level 2 nodes.

We can also use the ids that the JSP defines for the elements (`TD` elements) containing the individual conjugation forms. The JSP uses the textual representation of the conjugation forms for these ids (generated by the `toString` method from the values defined in the resource bundle), so we should do the same in the tests:

```
Element conjForm1 =
    _browser.getElementById(
        ConjugationFormTypeEnum.PARTICIPIO.toString());
Element conjForm2 =
    _browser.getElementById(
        ConjugationFormTypeEnum.SUBJUNTIVO_FUTURO_IMPERFECTO.toString() +
        " " +
        ConjugationPersonEnum.PLURAL_PRIMERO.toString());

assertEquals("dado",
    conjForm1.getTextContent().trim());
assertEquals("diéremos",
    conjForm2.getTextContent().trim());
```

That's about it. The complete test method now looks like this:

```

public void testDefinedData() throws IOException, SAXException
{
    _databaseHelper.insertData(TEST_DATA);

    try
    {
        _browser.loadURIAndWaitForCompletion(
            "http://localhost:8080/SpanishTrainer/index.jsp");
        _browser.setFormInputByName("j_username", "someuser");
        _browser.setFormInputByName("j_password", "abc");
        _browser.getSubmitButton().click();
        _browser.waitForLoadCompletion();

        _browser.clickAnchorByText(
            ResourceBundleHelper.getMessage("view.menu.searchForWord"),
            true);
        _browser.waitForLoadCompletion();

        assertEquals("search",
            _browser.getMetaElement("PageName").getContent());

        _browser.getFormInputByName("searchText").setValue("dar");
        _browser.getSubmitButton().click();
        _browser.waitForLoadCompletion();

        assertEquals("showword",
            _browser.getMetaElement("PageName").getContent());

        HTMLLegendElement legend =
            (HTMLLegendElement)DOMHelper.getFirstChildOfType(
                _browser.getDOM(), "legend", true);

        assertEquals("dar",
            legend.getTextContent().trim());

        Element conjForm1 =
            _browser.getElementById(
                ConjugationFormTypeEnum.PARTICIPIO.toString());
        Element conjForm2 =
            _browser.getElementById(
                ConjugationFormTypeEnum
                    .SUBJUNTIVO_FUTURO_IMPERFECTO.toString() +
                " " +
                ConjugationPersonEnum.PLURAL_PRIMERO.toString());

        assertEquals("dado",
            conjForm1.getTextContent().trim());
        assertEquals("diéremos",
            conjForm2.getTextContent().trim());
    }
    finally
    {
        _databaseHelper.deleteObjects("conjform",
            new String[] { "100", "101", "102",
                "103", "104", "105",
                "106" });
        _databaseHelper.deleteObjects("translations",
            new String[] { "100", "101" });
        _databaseHelper.deleteObject("verb", "100");
    }
}

```

## Lesson 4: Testing AJAX-enabled web pages

In the last months, the AJAX technology has been the talk of the day amongst web application developers. This is understandable as it allows for a feature richness and responsiveness that hasn't been achieved before in web applications. This is at least partly due to the visibility of Google's new AJAX-enhanced web applications such as the Google Mail service, Google Suggest and Google Maps.

In short, AJAX – which stands for Asynchronous JavaScript & XML – allows to interact with a web server in the background. No page refresh/reloading is performed, the communication is done asynchronously via JavaScript without requiring any interaction of the user. Now combine this with the document object model, and you can create web applications that are nearly as feature rich and interactive as their desktop counterparts.

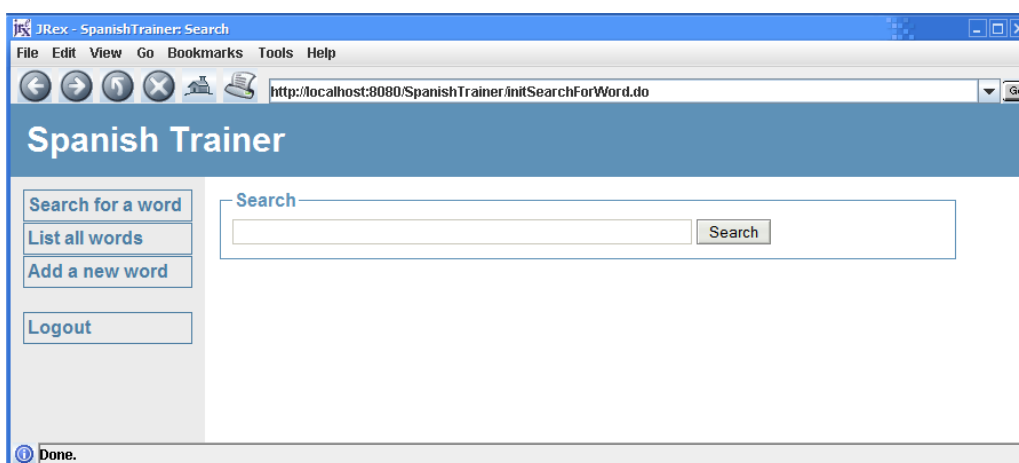
This trend is gaining even more momentum with the appearance of high-quality AJAX libraries that deal with the technical side of this and make it quite easy for web developers and even web designers to incorporate AJAX into their web applications.

This leaves two major obstacles in using AJAX. Web designers/developers have to design the web application differently because there is no longer the simple request-response cycle.

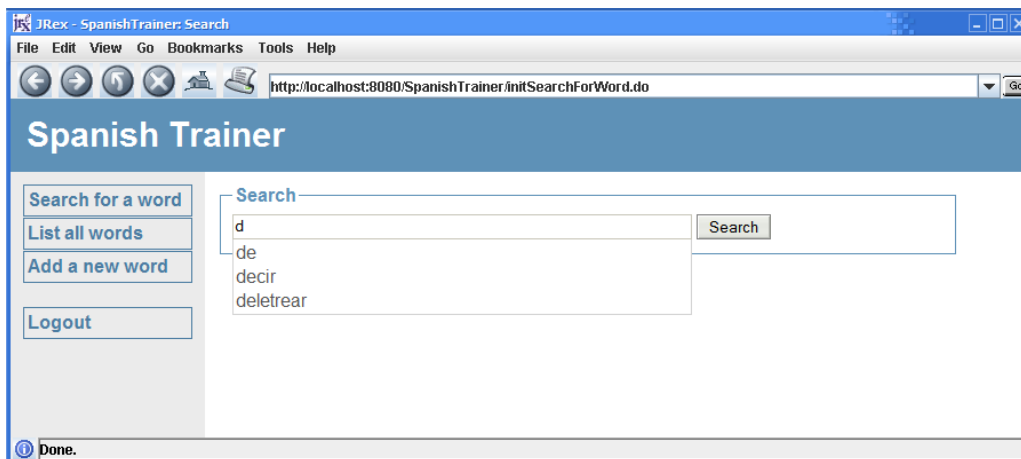
Also, this new technology shifts functionality to the client side, where JavaScript is now used to perform tasks that until recently would have been done by the server. JavaScript however is notoriously difficult to develop for, mostly because of the differences between the web browsers and the lack of development environments (IDEs).

Just like for normal web applications or for normal desktop applications, functional unit tests can help to improve this second point because they can be used to assert that the user interface works as expected, or even to define how it should behave.

The search page in the sample web application includes an AJAX-style functionality that works in the spirit of Google Suggest. While you enter the term to search for, the web page loads possible results in the background and presents them in a list below the search field:



*SpanishTrainer search page*



*SpanishTrainer search page with suggestions*

**Goal:**

Test the search hint functionality by

- logging into the web application
- going to the search page
- 'pressing' the key 'd' in the search box
- asserting that an element is now visible that contains three search hints
- select the second search hint via the mouse
- assert that we arrive at the detail page for the second search hint

We are now going to test this suggest function. As always, we first log into the web application and navigate to the search page:

```
_browser.loadURIAndWaitForCompletion(  
    "http://localhost:8080/SpanishTrainer/index.jsp");  
_browser.setFormInputByName("j_username", "someuser");  
_browser.setFormInputByName("j_password", "abc");  
_browser.getSubmitButton().click();  
_browser.waitForLoadCompletion();  
  
_browser.clickAnchorByText(  
    ResourceBundleHelper.getMessage("view.menu.searchForWord"), true);  
_browser.waitForLoadCompletion();
```

(If by now you think that we could extract the login and the initial navigation into a separate method that then can be used by the unit tests, then you are right, that would be a sensible thing to do.)

## ***Detour: AJAX in the sample web application***

For testing the AJAX part it would be useful to know how it is implemented in the sample web application. The SpanishTrainer web application uses DWR (<http://getahead.ltd.uk/dwr>) for this. AJAX with DWR is actually quite easy, only three steps were needed:

- DWR automatically creates JavaScript wrappers for Java classes, so in order to perform the search necessary to fill the suggest area, we need a Java class and tell DWR about it.
- In the search HTML (JSP) page we need to incorporate the DWR-created JavaScript.
- And finally, we have to use this JavaScript wrapper.

The search class has to perform only one simple task: search for all words that start with a given string. Since this is essentially a task that the database can perform with a LIKE query, the search class simply delegates this to the appropriate business service. We're using Spring, so gaining access to the service instance is quite easy:

```
/** @spring.bean name="searchBean" */
public class SearchBean
{
    private wordManagement _wordManagement;

    /** @spring.property ref="wordManagement" */
    public void setwordManagement(wordManagement management)
    {
        _wordManagement = management;
    }

    public SearchResult[] search(String startOfWord, int maxNumResults)
    {
        ArrayList result = new ArrayList();

        if ((startOfWord != null) && (startOfWord.length() > 0))
        {
            collection words = _wordManagement.getWordsStartingwith(
                startOfWord, maxNumResults)

            for (Iterator it = words.iterator(); it.hasNext();)
            {
                result.add(new SearchResult((Word)it.next()));
            }
        }
        return (SearchResult[])result.toArray(
            new SearchResult[result.size()]);
    }
}
```

The XDoclet tags ensure that the search bean instance is maintained by Spring and properly wired with the rest of the web application. We only have to tell DWR about it now:

```
<!DOCTYPE dwr PUBLIC
  "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
  "http://www.getahead.ltd.uk/dwr/dwr10.dtd">
<dwr>
  <allow>
    <create creator="spring"
      javascript="SearchBean">
      <param name="beanName"
        value="searchBean"/>
      <exclude method="setWordManagement"/>
    </create>
    <create creator="new"
      javascript="SearchResult">
      <param name="class"
        value="spanishtrainer.web.actions.SearchResult"/>
    </create>
    <convert converter="bean"
      match="spanishtrainer.web.actions.SearchResult"/>
  </allow>
</dwr>
```

As you'll notice we return an array of objects of the special `SearchResult` class instead of the collection returned by the DAO object. The reason is that this collection contains instances of the word model class and its subclasses, e.g. `Verb`, `Substantive` etc. Since we want to access information of the search result in the HTML page, we would have to tell DWR about all these classes. But we are only interested in some of the data in these words (namely the id, the word's text, the type of the word, and the translations), so we simply create a data transfer class for this which we then tell DWR about.

DWR is Spring-aware, it will pick up automatically the Spring-created singleton instance of our search bean because we told it that the search bean instance shall be retrieved from Spring via the `creator="spring"` attribute in the `dwr.xml` file.

For this all to work, we also have to add the DWR servlet to our web application's `web.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  ...

  <servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <display-name>DWR Servlet</display-name>
    <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>true</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>

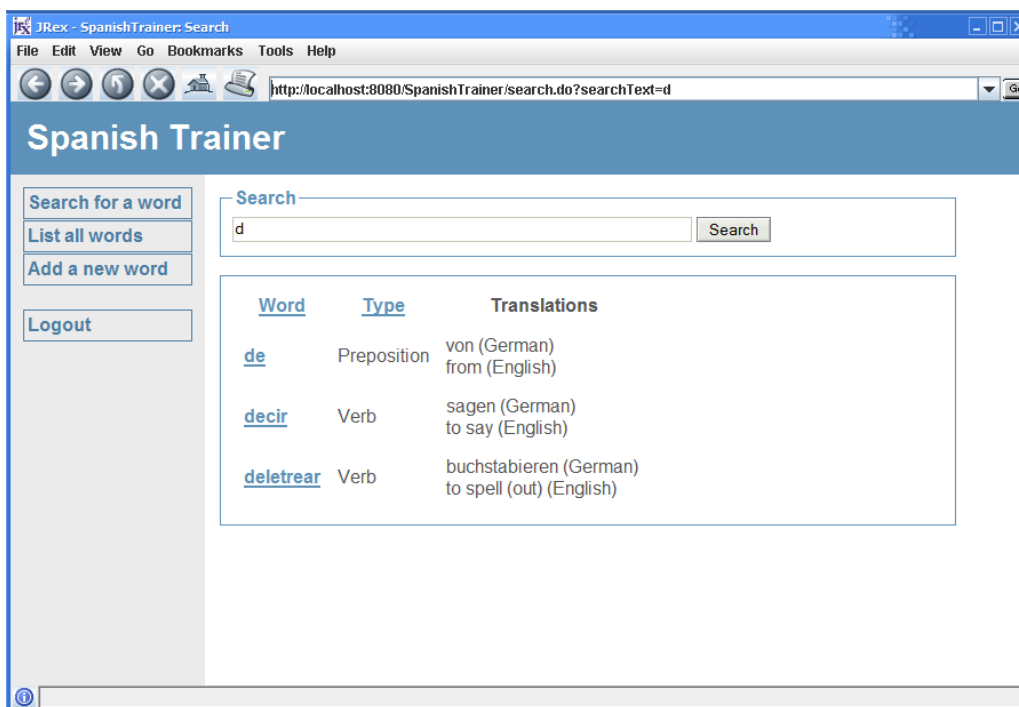
  ...
</web-app>
```

What's left is using the JavaScript wrapper in the JSP page:

```
<%@ page contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1" %>
<%@ include file="taglibs.jsp"%>
<html:html>
<head>
  <meta name="PageName" content="search"/>
  <title>
    <bean:message key="view.search.title"/>
  </title>
</head>
<body>
  <script language="JavaScript"
          src="/dwr/engine.js"
          type="text/javascript"></script>
  <script language="JavaScript"
          src="/dwr/util.js"
          type="text/javascript"></script>
  <script language="JavaScript"
          src="/dwr/interface/SearchBean.js"
          type="text/javascript"></script>
  ...
</body>
</html:html>
```

The first two JavaScript files provide the core DWR engine and some utility functions. The third one contains the JavaScript wrapper that is auto-generated by DWR for our search bean. As you can see, providing AJAX functionality is the easy part. What is hard is the manipulation of the web page to achieve the desired effect. Explaining this in detail would be a bit too much here, but in short it does the following:

- When the user presses a key in the input field, the DWR-generated JavaScript wrapper is invoked which in turn fetches the suggestions from the server. After that, the `div` element is shown and filled with a table that contains the suggestions.
- If the user presses the escape key in the input field, the `div` element is hidden.
- If the `div` element is shown and the user clicks on one of the suggestions, the browser loads the detail page for this word.
- If the user presses the enter key in the input field, the same search is started but via the normal request-response mechanism. The invoked action then either forwards to the detail page if exactly one word is found, or back to the search page, which now also shows all found words.



*SpanishTrainer search page with multiple results*

## **Back on track: Writing the test**

Lets go back to our unit test. In order to invoke the AJAX functionality, we need to press a key in the search input box. But we cannot use our earlier way of setting the value of the input field because that would not trigger the JavaScript handler that performs the AJAX stuff. So, instead we have to manually generate a key press event:

```
_browser.focus(_browser.getFormInputByName("searchText"));  
_browser.pressKey(KeyEvent.VK_D);  
  
Thread.sleep(5000);
```

We set the focus to the input field (otherwise the key press would end up somewhere else), and direct the browser to simulate a key press.

```
public void pressKey(int keyCode) throws BrowserException
{
    pressKeys(new int[] { keyCode });
}
public void pressKeys(int[] keyCodes) throws BrowserException
{
    try
    {
        GraphicsEnvironment environment =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        GraphicsDevice screen =
            environment.getDefaultScreenDevice();
        Robot robot = new Robot(screen);

        for (int idx = 0; idx < keyCodes.length; idx++)
        {
            robot.keyPress(keyCodes[idx]);
        }
        robot.delay(100);
        for (int idx = 0; idx < keyCodes.length; idx++)
        {
            robot.keyRelease(keyCodes[idx]);
        }
        robot.delay(100);
    }
    catch (AWTException ex)
    {
        throw new BrowserException(ex);
    }
}
```

The `Robot` class was created specifically for the purpose of automating the interaction (mouse, keyboard) with a Java AWT or Swing user interface.

The browser in our test case should now contact the web server and then the `DIV` element should be filled with the results. We give it a couple of seconds time for this and then access the `DIV` element as usual:

```
Thread.sleep(5000);

HTMLDivElement divElement = (HTMLDivElement)
    _browser.getElementById("searchHintList");
```

For checking whether the `DIV` element is visible, we have to access its `style` attribute and check whether the `display` CSS attribute is set to `none` (for invisible) or some other value such as `block`.

The `style` attribute however likely contains more than one CSS attribute, so `webappunit` provides a helper method `getStyleAttributeItem` that extracts a specific CSS attribute:

```
assertEquals("block",
            DOMHelper.getStyleAttributeItem(divElement, "display"));
```

The JavaScript function that is filling the `DIV` with the suggestions, is creating an id of the form “`suggestion2`” for the `TD` elements that contain the text of the suggested word. Thus, a check of the suggestions can be done by fetching the elements and checking their textual content:

```
assertEquals("de",
            _browser.getElementById("suggestion0").getTextContent().trim());
assertEquals("decir",
            _browser.getElementById("suggestion1").getTextContent().trim());
assertEquals("deletrear",
            _browser.getElementById("suggestion2").getTextContent().trim());

assertNull(_browser.getElementById("suggestion3"));
```

For good measure, we've also asserted that there are only three suggestions.

With the above assertions we are relying on the fact that we performed the database setup with initial data before running the test. Of course, we also could have applied the techniques from the previous lesson to explicitly initialize the database in the unit test itself.

Due to the usage of an embedded web server, we can also check the request(s) that are generated by DWR. For this, we enable the request logging at the server, and after the key press we check that an access to the DWR servlet was performed:

```
_server.setLoggingRequests(true);

_browser.focus(_browser.getFormInputByName("searchText"));
_browser.pressKey(KeyEvent.VK_D);
Thread.sleep(5000);

// the other checks ...

RequestInfo[] requests = _server.getRequests();

assertEquals(1,
            requests.length);
assertTrue(requests[0].getPath().endsWith("dwr/exec/SearchBean.search"));

_server.setLoggingRequests(false);
```

The `RequestInfo` class currently gives access to the request attributes and parameters, as well as to the original query string. One thing that `webappunit` is not yet able to provide, is the requests' content, which is for instance used by the DWR JavaScript wrapper to send XML to the DWR servlet. If we would have access to the content, we could then parse the XML that DWR sent to the server, and assert its contents, namely the string that is searched for.

It might be even more useful to be able to gather the requests that the browser sends (as opposed to gathering the requests that the server receives) because that would work even without using an embedded web server. However, intercepting the requests at the browser is a lot more difficult, so it is not sure whether that might be possible some time in the future.

To finish our test, we now click on the second suggestion and then assert that we arrive at the detail page for this word, the verb *decir*:

```
_browser.click((HTMLInputElement)_browser.getElementById("suggestion1"), 0);

Thread.sleep(100);
_browser.waitForLoadCompletion();

assertEquals("showWord",
    _browser.getMetaElement("PageName").getContent());
assertEquals("decir",
    _browser.getElementById("showWordLegend").getTextContent().trim());
```

This uses the same mechanism that we used before to click a link, only this time we click on a `TD` element. Since this element has an `onClick` JavaScript handler, this will start the loading of the detail page (for which we give it a bit of time).

The remaining two asserts in the above snippet check the detail page similar to how we did before in the previous lesson.

Here is the complete test:

```
public void testDWR() throws InterruptedException
{
    _browser.loadURIAndWaitForCompletion(
        "http://localhost:8080/SpanishTrainer/index.jsp");
    _browser.setFormInputByName("j_username", "someuser");
    _browser.setFormInputByName("j_password", "abc");
    _browser.getSubmitButton().click();
    _browser.waitForLoadCompletion();

    _browser.clickAnchorByText(
        ResourceBundleHelper.getMessage("view.menu.searchForWord"), true);
    _browser.waitForLoadCompletion();

    _server.setLoggingRequests(true);

    _browser.focus(_browser.getFormInputByName("searchText"));
    _browser.pressKey(KeyEvent.VK_D);
    Thread.sleep(5000);

    HTMLDivElement divElement = (HTMLDivElement)
        _browser.getElementById("searchHintList");

    assertEquals("block",
        DOMHelper.getStyleAttributeItem(divElement, "display"));

    assertEquals("de",
        _browser.getElementById("suggestion0").getTextContent().trim());
    assertEquals("decir",
        _browser.getElementById("suggestion1").getTextContent().trim());
    assertEquals("deletrear",
        _browser.getElementById("suggestion2").getTextContent().trim());

    assertNull(_browser.getElementById("suggestion3"));

    RequestInfo[] requests = _server.getRequests();

    assertEquals(1,
        requests.length);
    assertTrue(requests[0].getPath().endsWith(
        "dwr/exec/SearchBean.search"));

    _browser.click((HTMLElement)_browser.getElementById("suggestion1"), 0);
    Thread.sleep(100);
    _browser.waitForLoadCompletion();

    assertEquals("showword",
        _browser.getMetaElement("PageName").getContent());
    assertEquals("decir",
        _browser.getElementById("showwordLegend").getTextContent().trim());
}
```

## Lesson 5: Testing back button & double posting

Up to now, all parts that we've tested either did not contain forms, or the forms were in the request scope. This means, that all information about entered values is stored within the page and will be transmitted by the requests and responses – no form-related state is stored on the server. For simple and small forms such as the search or login forms, this works perfectly. However for larger forms, especially for multi-page forms, this leads to a proliferation of hidden inputs in the form for the data that is currently not shown. In these cases, web applications usually store already entered data in the session, for instance by defining the form at session scope.

However, usage of the session brings its own set of problems. The web application has to ensure that the form is removed from the session as soon as the data input workflow has been finished, regardless of whether by submitting the form or clicking in the menu to go to some other page.

This leads to some other problems typically encountered by web application developers: back button handling and double posting.

Every web browser has the back button, and if it is one thing that a user knows about navigating in the web, it is that the back button brings him back to an earlier visited page. And you can be sure that he is going to use it. The problem now is that when the back button is not properly handled by the web application, you get into trouble. Imagine for instance that you're using a web mail application in an Internet café. Now you log out, and the next user can use the back button to see your mail or even use your account – obviously not a good thing.

You might use the back button, say, to go back to a form that you entered data earlier on and submitted it. If you now submit again, what is supposed to happen? If you put something into your shopping cart, you actually might want to post it again. But if you entered your credit card data in order to pay for the goods, then you obviously only want it to happen once. This is known as the double post problem. The web application has to deal graciously with the re-submit of a form.

The two other common causes for double posting are:

- The user acts according to the elevator principle: “press the button more often, and it will go faster”
- Pressing the reload button on the page that the user were forwarded to after the form submit, brings the postdata warning dialog which asks the user whether he really wants to submit the form again – but often enough the user does not understand what this dialog asks, and simply clicks it away

The back button and double posting problems can be dealt with properly in a variety of ways, including:

- By controlling how the browser caches the web sites, or more precisely, directing the browser not to cache certain pages, the browser will perform a request to the server if the back button is hit instead of using its cache. This gives the web application control over what is shown.
- By putting special tokens into the request/form and also into the session, the web application has the ability to determine whether a certain form hasn't been handled yet (there would still

be its token in the session) or whether it is a double-post.

- The web application can use a workflow engine. These extend upon the token idea: a flow representation is stored in the session (or perhaps into the request) along with the information where the user currently is. Also the pages that are part of the flow are marked with the flow id. In combination this allows the web application to handle back button and double post easily. When the back button is pressed (in combination with disabling the caching) it simply means a step back in the workflow, and the web application can decide whether it allows this, and if, then can manipulate any data associated with the flow, e.g. reset a form.  
Double posting would be caught automatically by the flow engine because the workflow is already in a different step (usually one further)

You may wonder how to test that, but that is actually quite easy: we use `webappunit` to simulate typical usage scenarios where the problems would be exhibited.

For the purposes of this test, the sample web application contains a variant of the add-word page that uses a session-scoped form rather than the request-scoped one that we saw earlier. In addition, the sample web application contains some pretty basic handling of the back button: it disables the caching of the web pages, and upon leaving the add-word workflow, the form is automatically removed from the session. When the user presses the back button, he will be presented with an empty form.

For the sake of the test, there is also an error in the handling of the reload button: there is no redirect after the submit of the session-scoped add-word form. This will allow us to exhibit the double-post problem when reloading the page after adding a word.

For brevity, we will combine what usually would span multiple separate tests, into two tests.

### **Goal:**

Test the handling of a session-scoped form, of the back button and of double posting by

- logging into the web application
- navigating to the add word page that uses the session-scoped form
- entering some word data
- going to the search page without submitting the form
- pressing the back button
- asserting that the form does not contain any data

and

- logging into the web application
- navigating to the add word page that uses the session-scoped form
- entering some data
- pressing submit twice
- asserting that there is one new object in the database
- asserting that we are on a new page
- pressing the reload button

- asserting that we are on the same page as before
- asserting that there is still only one new object in the database

We start off as usual:

```
_browser.loadURIAndWaitForCompletion(
    "http://localhost:8080/SpanishTrainer/index.jsp");
_browser.setInputByName("j_username", "someuser");
_browser.setInputByName("j_password", "abc");
_browser.getSubmitButton().click();
_browser.waitForLoadCompletion();

_browser.clickAnchorByText(
    ResourceBundleHelper.getMessage("view.menu.addword"), true);
_browser.waitForLoadCompletion();

assertEquals("addOrEditword",
    _browser.getMetaElement("PageName").getContent());
```

Next, let's check that there actually exists a form in the session

```
assertNotNull(_server.getSession().getAttribute("addOrEditwordForm"));
```

We know the name of the form because we explicitly specified in the Struts action mapping of the two actions in the workflow:

```
<action path="/initAddOrEditwordSession"
        type="spanishtrainer.web.actions.InitAddOrEditwordSessionAction"
        name="addOrEditwordForm"
        attribute="addOrEditwordForm"
        scope="session"
        unknown="false"
        validate="false">
    ...
</action>
```

Here, the attribute aptly named `attribute` specifies under which name the form shall be stored in the session.

Next, we will start to add the verb *comprobar* (to check, to test). First we have to select the radio button for verbs:

```
List      radioButtons = _browser.getElementsByName("wordType");
HTMLInputElement radioButton = (HTMLInputElement)
        radioButtons.get(WordTypeEnum.VALUE_VERB - 1);

assertEquals(String.valueOf(WordTypeEnum.VALUE_VERB),
        radioButton.getValue());
_browser.click(radioButton, 0);
```

Radio buttons in a group have the same name, so we use the `getElementsByName` method. In it there are eight radio buttons, one for each defined word type (as specified by the enumeration `wordTypeEnum`), of which we retrieve the one for verbs using the corresponding enum value. Even though a radio button is a form input element and thus has a `click` method, we don't use it here. This is because the web page uses a JavaScript event handler for the `onclick` event, which would not be triggered when we'd use the DOM `click` method. Rather, we click the radio button just like we did before with the links and table cells: we simulate a mouse click on the radio button. In effect, the `DIV` element for verbs should now be visible which we of course check:

```
HTMLDivElement divElement = (HTMLDivElement)
        _browser.getElementById("wordTypeForm." +
        WordTypeEnum.VALUE_VERB);

assertEquals("block",
        DOMHelper.getStyleAttributeItem(divElement, "display"));
```

Now we add the text and origin. And then we go somewhere else:

```
_browser.setFormInputByName("text", "comprobar");
_browser.setFormInputByName("origin", "test");

_browser.clickAnchorByText(
        ResourceBundleHelper.getMessage("view.menu.searchForWord"), true);
_browser.waitForLoadCompletion();
```

Lets check that there is no longer a form object in the session, and that the database is unmodified:

```
assertNull(_server.getSession().getAttribute("addOrEditwordForm"));

assertTrue(
        _databaseHelper.getObjects("word", "text", "comprobar").isEmpty());
```

The `getObjects` method is a little helper method that retrieves the rows from the database in which a specified column has a given value:

```
public Collection getObjects(String tableName,
                             String columnName,
                             Object columnValue)
{
    Table table = getModel().findTable(tableName);
    ArrayList params = new ArrayList();

    params.add(columnValue);
    return getPlatform().fetch(
        getModel(),
        "select * from "+table.getName()+" where "+columnName+"=?",
        params,
        new Table[] { table });
}
```

But wait, we have reconsidered and now want to add the verb after all. So we press the back button and expect to be back at the add word page. However, the form should be empty again, because the data previously entered has been cleaned from the session:

```
_browser.goBack();
_browser.waitForLoadCompletion();

assertEquals("addOrEditWord",
    _browser.getMetaElement("PageName").getContent());

assertTrue(
    StringUtils.isEmpty(_browser.getFormInputByName("text").getValue()));
assertTrue(
    StringUtils.isEmpty(_browser.getFormInputByName("origin").getValue()));
```

(We're using the `StringUtils` class from `commons-lang` to test whether the strings are empty which could mean `null` or a string of length 0.)

We have left the actual creation of a new word for the second test in this lesson. This time, we will add the adverb *normalmente*:

```
_browser.loadURIAndWaitForCompletion(
    "http://localhost:8080/SpanishTrainer/index.jsp");
_browser.setFormInputByName("j_username", "someuser");
_browser.setFormInputByName("j_password", "abc");
_browser.getSubmitButton().click();
_browser.waitForLoadCompletion();

_browser.clickAnchorByText(
    ResourceBundleHelper.getMessage("view.menu.addword"), true);
_browser.waitForLoadCompletion();

List      radioButtons = _browser.getElementsByName("wordType");
HTMLInputElement radioButton = (HTMLInputElement)
    radioButtons.get(WordTypeEnum.VALUE_ADVERB - 1);

_browser.click(radioButton, 0);

_browser.setFormInputByName("text", "normalmente");
_browser.setFormInputByName("origin", "test");
```

We could also add translations for both German and English. The names of the input elements for the translations map to indexed properties in the form, for which Struts generates indexed names in the resulting HTML:

```
_browser.clickButtonByName("buttons.addTranslation");
_browser.waitForLoadCompletion();

_browser.getOptionInSelectByName(
    "translations[0].language",
    LanguageEnum.VALUE_DE - 1).setSelected(true);
_browser.setFormInputByName("translations[0].text", "normal");

_browser.clickButtonByName("buttons.addTranslation");
_browser.waitForLoadCompletion();

_browser.getOptionInSelectByName(
    "translations[1].language",
    LanguageEnum.VALUE_EN - 1).setSelected(true);
_browser.setFormInputByName("translations[1].text", "normally");

_browser.clickButtonByValue(
    ResourceBundleHelper.getMessage("view.addOrEditword.adverb.submit"));
_browser.clickButtonByValue(
    ResourceBundleHelper.getMessage("view.addOrEditword.adverb.submit"));
_browser.waitForLoadCompletion();
```

After submitting the form twice, we should check the database:

```
Collection objs = _databaseHelper.getObjects("adverb",
                                             "text",
                                             "normalmente");

assertEquals(1, objs.size());

DynaBean bean = (DynaBean)objs.iterator().next();

assertEquals("normalmente",
             bean.get("text"));
assertEquals("test",
             bean.get("origin"));
assertEquals("index",
             _browser.getMetaElement("PageName").getContent());
```

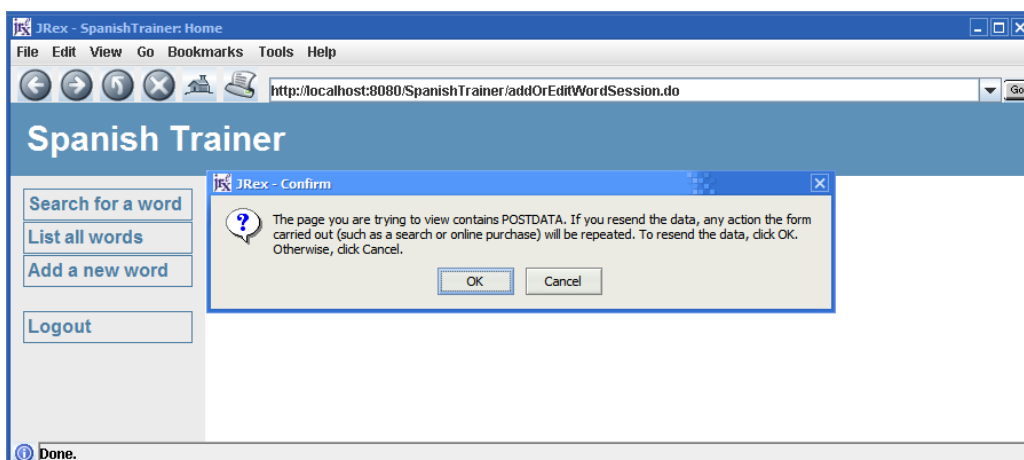
Now we press the reload button, and then check the database again:

```
_browser.reloadPage();
_browser.waitForLoadCompletion();

objs = _databaseHelper.getObjects("adverb", "text", "normalmente");

assertEquals(1,
            objs.size());
```

Running this against the sample web application will fail because the Struts action that handles the submit will not redirect to the index page, only forward. The effect is that the browser will show the familiar “do you want to submit again” dialog:



*POSTDATA warning dialog*

## Summary and outlook

As you've seen, webappunit is already in a quite usable state. It allows to easily define automated tests for the user interface of web applications in a way that is very near to what a user would perform.

The appeal of webappunit when compared to other similar testing frameworks comes from the fact, that it uses a real web browser with all its differences, quirks and errors. This means full support of JavaScript as provided by the browser, and of new technologies like AJAX.

Webappunit also allows to test against a remote web server or against an embedded one. When using an embedded web server, the tests even have comprehensive access to the communication between browser and server, and to the state of the web application and the web server in general. Testing against a remote server on the other hand means that the tests are not limited to Java web applications.

Webappunit is already used for testing commercial web applications, and it is under active development, though it still contains a few limitations:

- It cannot run headless which would allow for automated test runs on a dedicated server, for instance using CruiseControl (<http://cruisecontrol.sourceforge.net/>). However, it might not even be technical possible to run a native browser in headless mode.  
A related restriction of this approach is that the individual browsers only run on certain systems. For instance, it is not possible to test against Internet Explorer on a Linux system, or against Safari on a Windows computer.
- Currently only Jetty is supported as the embedded web server, though a Tomcat variant is under development. Adding support for other Java web servers is not difficult however, as long as they can be used in embedded mode.
- Currently only supports JREx (Mozilla).  
In contrast to the server side, adding support for other browsers is somewhat more involved because they are not written in Java. Hence, a Java binding needs to be defined.  
For Internet Explorer it is possible to use JExplorer (<http://www.jniwrapper.com/pages/jexplorer/overview>), which is a commercial library that makes Internet Explorer available to Java programs. Another option is the browser component of Eclipse SWT which uses the native browser of the system that it is executed in.
- The biggest functional limitation is that it is currently not possible to check for open windows. The sample web application for instance opens a confirmation dialog when the user chooses to delete a word. Likewise, the browser opens windows for instance to warn of double posting.