
Chapter 20

Vaadin TestBench

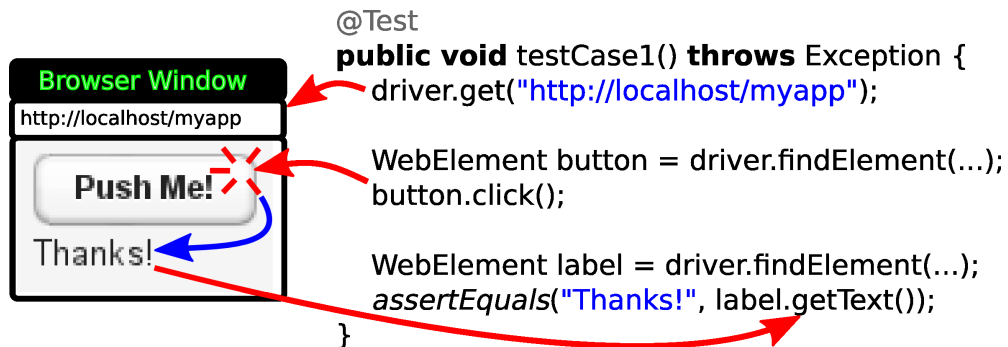
20.1. Overview	453
20.2. Installing Vaadin TestBench	456
20.3. Preparing an Application for Testing	461
20.4. Using Vaadin TestBench Recorder	462
20.5. Developing JUnit Tests	468
20.6. Taking and Comparing Screenshots	479
20.7. Running Tests in an Distributed Environment	483
20.8. Known Issues	485

This chapter describes the installation and use of the Vaadin TestBench.

20.1. Overview

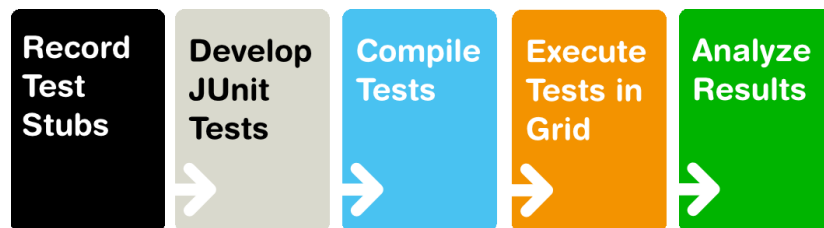
Quality assurance is one of the cornerstones of modern software development. Extending throughout the entire development process, quality assurance is the thread that binds the end product to the requirements. In iterative development processes, with ever shorter release cycles and continuous integration, the role of regression testing is central. The special nature of web applications creates many unique requirements for regression testing.

In a typical situation, you are developing a web application with Vaadin and want to ensure that only intended changes occur in its behaviour after modifying the code, without testing the application manually every time. There are two basic ways of detecting such regressions. Screenshots are the strictest way, but often just checking the displayed values in the HTML is better if you want to allow some flexibility for themeing, for example. You may also want to generate many different kinds of inputs to the application and check that they produce the desired outputs.

Figure 20.1. Controlling the Browser with WebDriver

Vaadin TestBench utilizes the Selenium WebDriver to control the browser from Java code, as illustrated in Figure 20.1, “Controlling the Browser with WebDriver”. It can open a new browser window to start the application, interact with the components for example by clicking them, and then get the HTML element values.

You can develop such WebDriver unit tests along your application code, for example with JUnit, which is a widely used Java unit testing framework. You can also use a recorder that runs in the browser to create JUnit test case stubs, which you can then refine further with Java. You can run the tests as many times as you want in your workstation or in a distributed grid setup.

Figure 20.2. TestBench Workflow

The main features of Vaadin TestBench are:

- Record JUnit test case stubs in browser
- Develop tests in Java with the WebDriver
- Validate UI state by assertions and screen capture comparison
- Screen capture comparison with difference highlighting
- Distributed test grid for running tests
- Integration with unit testing
- Test with browsers on mobile devices

Execution of tests can be distributed over a grid of test nodes, which speeds up testing. The grid nodes can run different operating systems and have different browsers installed. In a minimal setup, such as for developing the tests, you can use Vaadin TestBench on just a single computer.

Based on Selenium

Vaadin TestBench is based on the Selenium web browser automation library. With the Selenium WebDriver API, you can control browsers straight from Java code. The TestBench Recorder is based on the Selenium IDE.

Selenium is augmented with Vaadin-specific extensions, such as:

- Proper handling of Ajax-based communications of Vaadin
- Exporting test case stubs from the Recorder
- Performance testing of Vaadin applications
- Screen capture comparison
- Finding HTML elements using a Vaadin selector

TestBench Components

The main components of Vaadin TestBench are:

- Vaadin TestBench Java Library
- Vaadin TestBench Recorder

The library includes WebDriver, which provides API to control a browser like a user would. This API can be used to build tests, for example, with JUnit. It also includes the grid hub and node servers, which you can use to run tests in a grid configuration.

The Vaadin TestBench Recorder is helpful for creating test case stubs. It is a Firefox extension that you install in your browser. It has a control panel to record test cases and play them back. You can play the test cases right in the recorder. You can then export the tests as JUnit tests, which you can edit further and then execute with the WebDriver.

Vaadin TestBench Library provides the central control logic for:

- Executing tests with the WebDriver
- Additional support for testing Vaadin-based applications
- Comparing screen captures with reference images
- Distributed testing with grid node and hub services

Requirements

Requirements for recording test cases with Vaadin TestBench Recorder:

- Mozilla Firefox

Requirements for running tests:

- Java JDK 1.6 or newer
- Browsers installed on test nodes as supported by Selenium WebDriver

- Google Chrome
- Internet Explorer
- Mozilla Firefox
- Opera
- Mobile browsers: Android, iPhone
- Build system such as Ant or Maven to automate execution of tests during build process (recommended)

Continuous Integration Compatibility

Continuous integration means automatic compilation and testing of applications frequently, typically at least daily, but ideally every time when code changes are committed to the source repository. This practice allows catching integration problems early and finding the changes that first caused them to occur.

You can make unit tests with Vaadin TestBench just like you would do any other Java unit tests, so they work seamlessly with continuous integration systems. Vaadin TestBench is tested to work with at least TeamCity and Hudson/Jenkins build management and continuous integration servers, which all have special support for the JUnit unit testing framework.

Licensing and Trial Period

You can download Vaadin TestBench from Vaadin Directory and try it out for a free 30-day trial period, after which you are required to acquire the needed licenses. You can purchase licenses from the Directory. A license for Vaadin TestBench is also included in the Vaadin Pro Account subscription.

20.2. Installing Vaadin TestBench

Installation of Vaadin TestBench covers the following tasks:

- Download and unpack the Vaadin TestBench installation package
- Install Vaadin TestBench Recorder
- Install Vaadin TestBench Library

Which modules you need to install depends on whether you are developing tests or running existing tests. Two basic installation types are covered in these instructions:

- Test development installation on a workstation
- Distributed grid installation

20.2.1. Test Development Installation

In a typical test development setup, you install Vaadin TestBench on a workstation. You can use the TestBench Recorder to record test cases and export them as JUnit test case stubs. This is especially recommended if you are new to Vaadin TestBench and do not want to code from

scratch. You can install the Recorder in Firefox as described in Section 20.2.6, “Installing the Recorder”.

You may find it convenient to develop and execute tests under an IDE such as Eclipse. The special support for running JUnit test cases in Eclipse is described in Section 20.5.3, “Running JUnit Tests in Eclipse”.

In such a test development setup, you do not need a grid hub or nodes. However, if you develop tests for a grid, you can run the tests, the grid hub, and one node all in your development workstation. A distributed setup is described in the following section.

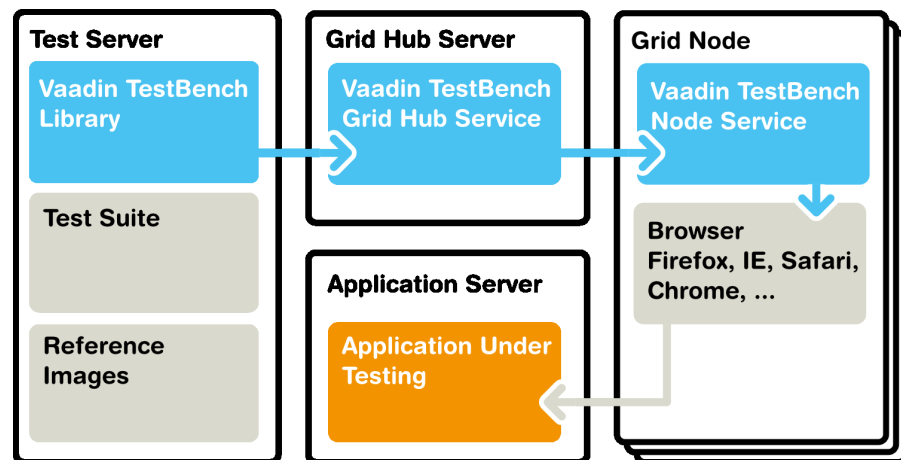
20.2.2. A Distributed Testing Environment

Vaadin TestBench supports distributed execution of tests in a grid. A test grid consists of the following categories of hosts:

- One or more test servers executing the tests
- A grid hub
- Grid nodes

The components of a grid setup are illustrated in Figure 20.3, “Vaadin TestBench Grid Setup”.

Figure 20.3. Vaadin TestBench Grid Setup



The grid hub is a service that handles communication between the JUnit test runner and the nodes. The nodes are services that perform the actual execution of test commands in the browser.

The hub requires very little resources, so you would typically run it either in the test server or on one of the nodes. You can run the tests, the hub, and one node all in one host, but in a fully distributed setup, you install the Vaadin TestBench components on separate hosts.

Controlling browsers over a distributed setup requires using a remote WebDriver. Grid development and use of the hub and nodes is described in Section 20.7, “Running Tests in an Distributed Environment”.

20.2.3. Downloading and Unpacking the Installation Package

First, download the installation package `vaadin-testbench-3.0.0.zip` and extract the installation package where you can find it.

Windows

In Windows, use the default ZIP decompression feature to extract the package into your chosen directory, for example, `C:\dev`.



Windows Zip Decompression Problem

The default decompression program in Windows XP and Vista as well as some versions of WinRAR cannot unpack the installation package properly in certain cases. Decompression can result in an error such as: "The system cannot find the file specified." This can happen because the default decompression program is unable to handle long file paths where the total length exceeds 256 characters. This can occur, for example, if you try to unpack the package under Desktop. You should unpack the package directly into `C:\dev` or some other short path or use another decompression program.

Linux, MacOS X, and other UNIX

In Linux, Mac OS X, and other UNIX-like systems, you can use Info-ZIP or other ZIP software with the command:

```
$ unzip vaadin-testbench-3.0.0.zip
```

The contents of the installation package will be extracted under the current directory.

In Mac OS X, you can also double-click the package to extract it under the current folder in a folder with the same name as the package.

20.2.4. Installation Package Contents

The installation package contains the following:

documentation

The documentation folder contains the TestBench library API documentation, a PDF excerpt of this chapter of Book of Vaadin, and the license.

example

The example folder provides TestBench examples. An example Maven configuration POM is given, as well as the JUnit test Java source files. For a description of the contents, see Section 20.2.5, "Example Contents".

maven

The Maven folder contains version of the Vaadin TestBench libraries that you can install in your local Maven repository. Please follow the instructions in Section 20.5.5, "Executing Tests with Maven".

vaadin-testbench-recorder

This folder contains the Vaadin TestBench Recorder, which you can install in Firefox. Please follow the instructions in Section 20.2.6, "Installing the Recorder".

`vaadin-testbench-standalone-3.0.0.jar`

This is the Vaadin TestBench library. It is a standalone library that includes the Selenium WebDriver and many other required libraries.

`vaadin-testbench-standalone-3.0.0-javadoc.jar`

This is the JavaDoc API documentation for the TestBench library. If you use Eclipse, you can associate the JAR with the TestBench JAR in the project preferences, in the build path library settings.

20.2.5. Example Contents

The `example/maven` folder provides a number of examples for using Vaadin TestBench. The source code for the application to be tested, a desktop calculator application, is given in the `src/main/java` subfolder.

The tests examples given under the `src/test/java` subfolder, in the `com/vaadin/testbenchexample` package subfolder, are as follows:

`SimpleCalculatorITCase.java`

Demonstrates the basic use of WebDriver. Interacts with the buttons in the user interface by clicking them and checks the resulting value. Uses `By.id()` to access the elements.

`LoopingCalculatorITCase.java`

Otherwise as the simple example, but shows how to use looping to produce programmatic repetition to create a complex use case.

`ScreenshotITCase.java`

Shows how to compare screenshots, as described in Section 20.6, “Taking and Comparing Screenshots”. Some of the test cases include random input, so they require masked screenshot comparison to mask the random areas out.

The included reference images were taken with Firefox on Mac OS X, so if you use another platform, they will fail. You will need to copy the error images to the reference screenshot folder and mask out the areas with the alpha channel as described in Section 20.6.3, “Taking Screenshots for Comparison”.

`SelectorExamplesITCase.java`

This example shows how to use different selectors:

- `By.id()` - selecting by identifier
- `By.xpath()` - selecting by an XPath expression

`VerifyExecutionTimeITCase.java`

Shows how to time the execution of a test case and how to report it.

`AdvancedCommandsITCase.java`

Demonstrates how to test tooltips (Section 20.5.10, “Testing Tooltips”) and context menus. Uses debug IDs, XPath expressions, as well as CSS selectors to find the elements to check.

For information about running the examples with Maven, see Section 20.5.5, “Executing Tests with Maven”.

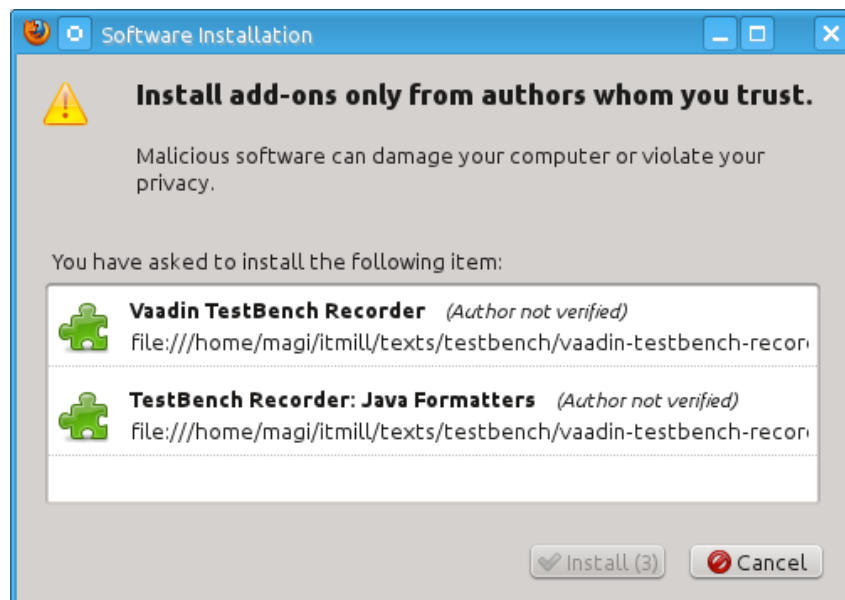
20.2.6. Installing the Recorder

You can use the Vaadin TestBench Recorder in a test development environment to record test cases and to export them as JUnit test case stubs, which you can then develop further. This gives you a quick start when you are learning to use TestBench. Later you can use the Recorder to identify the HTML DOM paths of the user interface elements which you want to test.

After extracting the files from the installation package, do the following:

1. Change to the `vaadin-testbench-recorder` directory under the installation directory.
2. Open Mozilla Firefox
3. Either drag and drop the `vaadin-testbench-recorder-3.0.0.xpi` to an open Firefox window or open it from the menu with **File** **Open File**.
4. Firefox will ask if you want to install the TestBench Recorder extension. Click **Install**.

Figure 20.4. Installing Vaadin TestBench Recorder



5. After the installation of the add-on is finished, Firefox offers to restart. Click **Restart Now**.

Installation of a new version of Vaadin TestBench Recorder will overwrite an existing previous version.

After Firefox has restarted, navigate to a Vaadin application for which you want to record test cases, such as <http://demo.vaadin.com/sampler>.

20.2.7. Test Node Configuration

If you are running the tests in a grid environment, you need to make some configuration to the test nodes to get more stable results.

Operating system settings

Make any operating system settings that might interfere with the browser and how it is opened or closed. Typical problems include crash handler dialogs.

On Windows, disable error reporting in case a browser crashes as follows:

1. Open **control panel** **System**
2. Select **Advanced** tab
3. Select **Error reporting**
4. Check that **Disable error reporting** is selected
5. Check that **But notify me when critical errors occur** is not selected

Settings for Screenshots

The screenshot comparison feature requires that the user interface of the browser stays constant. The exact features that interfere with testing depend on the browser and the operating system.

In general:

- Disable blinking cursor
- Use identical operating system themeing on every host
- Turn off any software that may suddenly pop up a new window
- Turn off screen saver

If using Windows and Internet Explorer, you should give also the following setting:

- Turn on **Allow active content to run in files on My Computer** under **Security settings**

20.3. Preparing an Application for Testing

Vaadin TestBench can usually test Vaadin applications as they are, especially if just taking screenshots. However, assertions on HTML elements require a DOM path to the element and this path is vulnerable to even small changes in the DOM structure. They might change because of your layout or UI logic, or if a new Vaadin version has some small changes. To make such problems less common, you can use *debug IDs* to refer to components.

```
public class ApplicationToBeTested extends Application {
    public void init() {
        final Window main = new Window("Test window");
        setMainWindow(main);

        // Create a button
        Button button = new Button("Push Me!");

        // Optional: give the button a unique debug ID
```

```

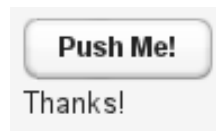
button.setDebugId("main.button");

// Do something when the button is clicked
button.addListener(new ClickListener() {
    @Override
    public void buttonClick(ClickEvent event) {
        // This label will not have a set debug ID
        main.addComponent(new Label("Thanks!"));
    }
});
main.addComponent(button);
}
}

```

The application is shown in Figure 20.5, “A Simple Application To Be Tested”, with the button already clicked.

Figure 20.5. A Simple Application To Be Tested



The button would be rendered as a HTML element: `<div id="main.button" ...>...</div>`. The DOM element would then be accessible from the HTML page with: `driver.findElement(By.id="main.button")`. For the label, which doesn't have a debug ID, the path would be from the page root. A recorded test case stub for the above application is given in Section 20.5.1, “Starting From a Stub”, which is further refined in this chapter.

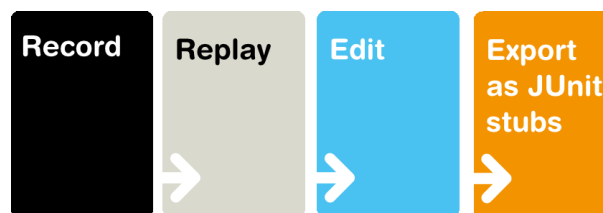
20.4. Using Vaadin TestBench Recorder

The Vaadin TestBench Recorder is used for recording and exporting JUnit test stubs that you can then develop further.

The most important role for using the Recorder is to identify all user interface elements that you want to test - you can do all other test logic by coding. The elements are identified by a *selector*, which usually use an HTML document path that selects the element. By default, the Recorder records the paths using a Vaadin selector, where the root of the path is the application element. The path can also be an XPath expression or a CSS selector. It can use a debug ID that you can set in the application code.

You can play back recoded test cases and use the Recorder to make assertions and take screenshots for screen capture comparison. Then, you export the test stubs as JUnit Java source files which you can then develop further.

Figure 20.6. Recorder Workflow



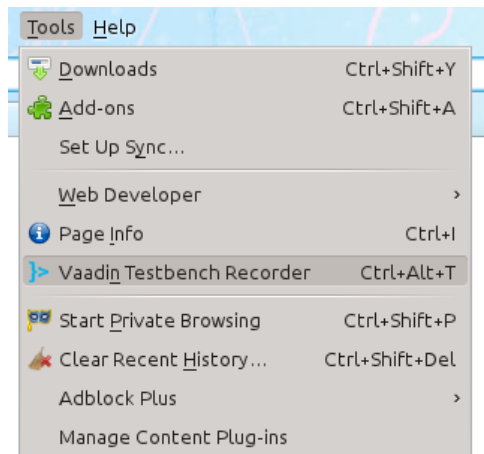
The Recorder is available only for Mozilla Firefox. To run the recorded tests in other browsers, you need to export them as JUnit tests and launch the other browsers with the WebDriver, as described later.

20.4.1. Starting the Recorder

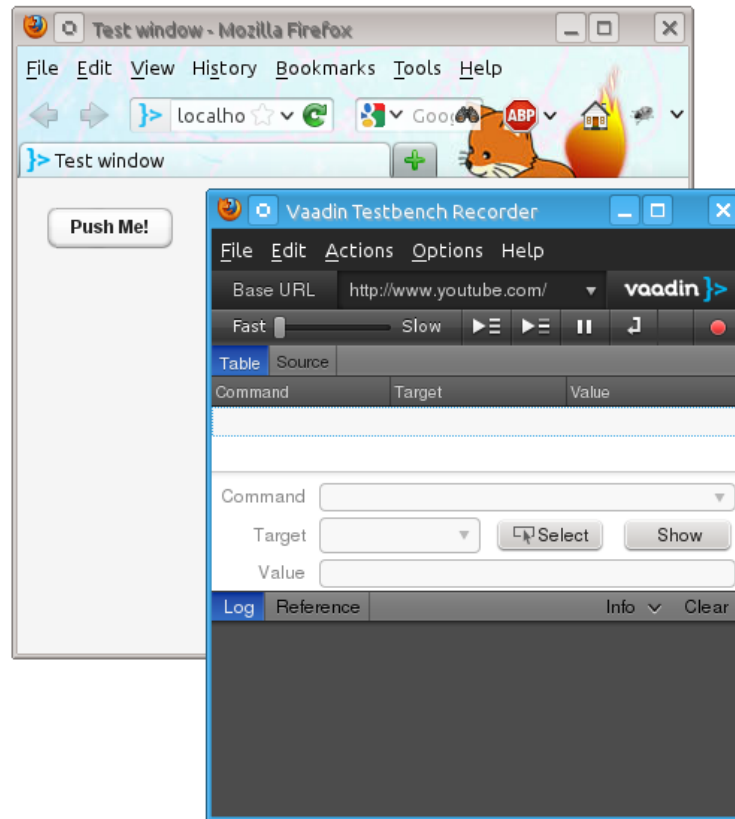
To start the Recorder:


1. Open Mozilla Firefox
2. Open the page with the application that you want to test
3. Select **Tools Vaadin TestBench Recorder** in the Firefox menu

Figure 20.7. Starting Vaadin TestBench Recorder



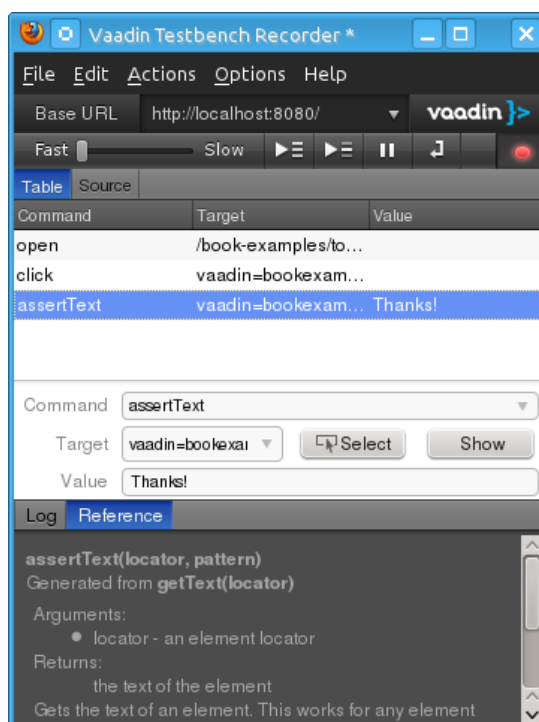
The Vaadin TestBench Recorder window will open, as shown in Figure 20.8, “Vaadin TestBench Recorder Running”.

Figure 20.8. Vaadin TestBench Recorder Running

Recording is automatically enabled when the Recorder starts. This is indicated by the pressed  **Record** button.

20.4.2. Recording

While recording, you can interact with the application in (almost) any way you like. The Recorder records the interaction as commands in a test script, which is shown in tabular format in the Table tab and as HTML source code in the Source tab.

Figure 20.9. User Interaction Recorded as Commands

Please note the following:

- Changing browser tabs or opening a new browser window is not recommended, as any clicks and other actions will be recorded
- Passwords are considered to be normal text input and are stored in plain text

While recording, you can insert various commands such as assertions or take a screenshot by selecting the command from the Command list.

When you are finished, click the  **Record** button to stop recording.

20.4.3. Selectors

The Recorder supports various *selectors* that allow finding the HTML elements that are interacted upon and asserted. By default, Recorder uses the *Vaadin selector*, which finds the elements by an application identifier, a possible debug ID, and a component hierarchy path.

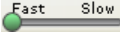
You can find elements by a plain XPath expression from the page root, an element ID, CSS style class, etc. The selectors are exported with the JUnit test cases as corresponding Vaadin or Selenium selector methods, described in Section 20.5.2, “Finding Elements by Selectors”.

Some selectors are not applicable to all elements, for example if an element does not have an ID or it is outside the Vaadin application. In such case, another selector is used according to a preference order. You can change the order of the preferred selectors by selecting **Options** **Options** **Locator Builders** and dragging the selectors (or locators) to a preferred order. Normally, the Vaadin selector should be at top.

20.4.4. Playing Back Tests

After you have stopped recording, reset the application to the initial state and press **▶ Play current test** to run the test again. You can use the `?restartApplication` parameter for an application in the URL to restart it.

You can also play back tests saved in the HTML format by first opening a test in the Recorder with **File Open**.

You can use the  slider to control the playback speed, click **Pause** to interrupt the execution and **Resume** to continue. While paused, you can click **Step** to execute the script step-by-step.

Check that the test works as intended and no unintended or invalid commands are found; a test should run without errors.

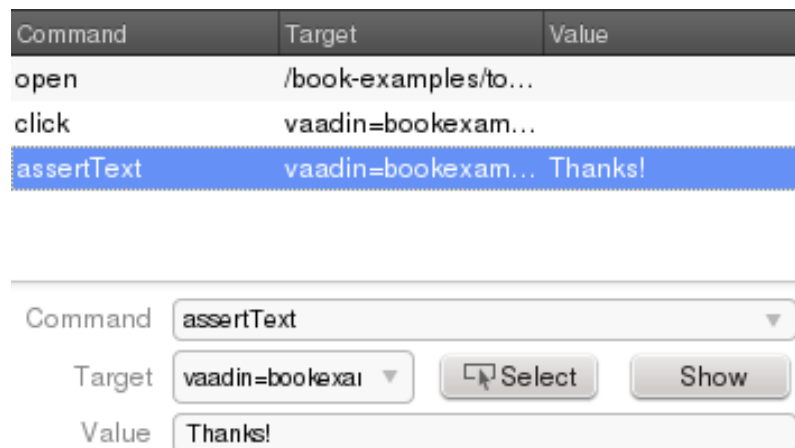
20.4.5. Editing Tests

While the primary purpose of using the Recorder is to identify all user interface elements to be tested, you can also edit the tests at this point. You can insert various commands, such as assertions or taking a screenshot, in the test script during or after recording.

You insert a command by selecting an insertion point in the test script and right-clicking an element in the browser. A context menu opens and shows a selection of Recorder commands at the bottom. Selecting **Show All Available Commands** shows more commands. Commands inserted from the sub-menu are automatically added to the top-level context menu.

Figure 20.10, “Inserting commands in a test script” shows adding an assertion after clicking the button in the example application.

Figure 20.10. Inserting commands in a test script



Inserting a command from the context menu automatically selects the command in the **Command** field and fills in the target and value parameters.

You can also select the command manually from the **Command** list. The new command or comment will be added at the selected location, moving the selected location down. If the command requires a target element, click **Select** and then click an element in your application. A reference

to the element is shown in the **Target** field and you can highlight the element by clicking **Show**. If the command expects some value, such as for comparing the element value, give it in the **Value** field.

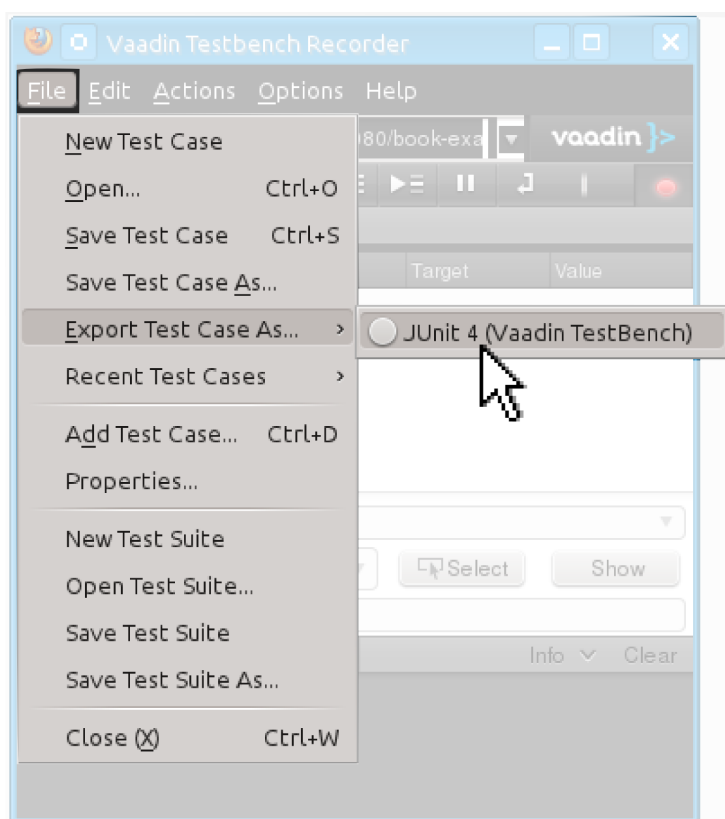
Commands in a test script can be changed by selecting a command and changing the command, target, or value.

20.4.6. Exporting Tests

Once you have are satisfied with a test case, you need to export it as a JUnit test case stub.

You can save a test by selecting **File** **Export** **JUnit Test**.

Figure 20.11. Exporting Test Case as JUnit Test



In the dialog that opens, enter a file name for the Java source file. The file contains a Java class with name **Testcase**, so you might want to name the file as `Testcase.java`. You can rename the class later.

20.4.7. Saving Tests

While exporting tests as JUnit tests is the normal case, the Recorder also allows saving test cases and test suites in a HTML format that can be loaded back in the Recorder. Vaadin TestBench does not support other use for these saved tests, but you may still find the feature useful if you like to develop test cases more with the Recorder.

20.5. Developing JUnit Tests

Tests are developed using the Selenium WebDriver, which is augmented with Vaadin TestBench API features useful for testing Vaadin applications.

Perhaps the easiest way to start developing tests is to use the Recorder to create a JUnit test stub, which is described in the next section. The main purpose of the recorder is to help identify the HTML DOM paths of the user interface elements that you want to interact with and use for assertions. Once you get the hang of coding tests, you should be able to do it without using the Recorder. Working with debug IDs and using a browser debugger, such as Firebug, is usually the easiest way to find out the DOM paths. You can also use the Recorder just to find the paths, and copy and paste them directly to your source code without going through the export hassle.

While this section describes the development of JUnit tests, Vaadin TestBench and the WebDriver are in no way specific to JUnit and you can use any test execution framework, or just regular Java applications, to develop TestBench tests.

20.5.1. Starting From a Stub

Let us assume that you recorded a simple application, as described earlier, and exported it as a JUnit stub. You can add it to a project in a suitable package. You may want to keep your test classes in a separate source tree in your application project, or in an altogether separate project, so that you do not have to include them in the web application WAR. Having them in the same project may be nicer for version control purposes.

You need to perform at least the following routine tasks:

- Rename the package
- Rename the class
- Check the base URL
- Clean up unnecessary code

A JUnit stub will look somewhat as follows:

```
package com.example.tests;

import java.util.regex.Pattern;
import java.util.concurrent.TimeUnit;
import org.junit.*;
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.openqa.selenium.*;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.Select;
import com.vaadin.testbench.By;
import com.vaadin.testbench.TestBench;
import com.vaadin.testbench.TestBenchTestCase;

public class Testcase1 extends TestBenchTestCase {
    private WebDriver driver;
    private String baseUrl;
    private StringBuffer verificationErrors = new StringBuffer();
```


The `verificationErrors` is used to collect some errors in some recorded commands, but can be removed if such commands are not used. You can also use it to collect non-fatal errors, for example screenshot comparison errors, and only fail on logic errors.

Test Setup

The set-up method, annotated with `@Before`, makes the basic configuration for the test. Most importantly, it creates the **WebDriver** instance, which is for Firefox by default. Drivers for different browsers extend the **RemoteWebDriver** class - see the API type hierarchy for the complete list.

```
@Before
public void setUp() throws Exception {
    driver = TestBench.createDriver(new FirefoxDriver());
    baseUrl = "http://localhost:8080/myapp";
}
```

Check that the `baseUrl` is the correct URL for the application. It might not be.

Test Case Stub

The test case methods are marked with `@Test` annotation. They normally start by calling the `get()` method in the driver. This loads the URL in the browser.

Actual test commands usually call the `findElement()` method in the driver to get hold of an HTML element to work with. The button has the `main.button` ID, as we set that ID for the **Button** object with the `setDebugId()` method in the application. The HTML element is represented as a **WebElement** object.

```
@Test
public void testCase1() throws Exception {
    driver.get(concatUrl(baseUrl, "/myapp"));
    assertEquals("Push Me!", driver.findElement(By.vaadin(
        "bookexamplestobetested::PID_Smain.button")).getText());
    driver.findElement(By.vaadin(
        "bookexamplestobetested::PID_Smain.button")).click();
    assertEquals("Thanks!", driver.findElement(By.vaadin(
        "bookexamplestobetested::VVerticalLayout[0]/"+
        "ChildComponentContainer[1]/VLabel[0]")).getText());
}
```

The `get()` call appends the application path to the base URL. If it is already included in the base URL, you can remove it.

After Testing

Finally after running all the test cases, the method annotated with `@After` is called. Calling `quit()` for the driver closes the browser window.

The stub includes code for collecting verification errors. If you do not collect those, as is often the case, you can remove the code.

```
@After
public void tearDown() throws Exception {
    driver.quit();

    String verificationErrorString =
        verificationErrors.toString();
    if (!"".equals(verificationErrorString)) {
        fail(verificationErrorString);
    }
}
```

20.5.2. Finding Elements by Selectors

The Selenium WebDriver API provides a number of different *selectors* for finding HTML DOM elements. The available selectors are defined as static methods in the **org.openqa.selenium.By** class. They create and return a **By** instance, which you can use for the `findElement()` method in **WebDriver**.

The ID, CSS class, and Vaadin selectors are described below. For others, we refer to the Selenium WebDriver API documentation [http://seleniumhq.org/docs/03_webdriver.html].

Finding by ID

Selecting elements by their HTML element `id` attribute is usually the easiest way to select elements. It requires that you use debug IDs, as described in Section 20.3, “Preparing an Application for Testing”. The debug ID is used as is for the `id` attribute of the top element of the component. Selecting is done by the `By.id()` selector.

For example, in the `SimpleCalculatorITCase.java` example we use the debug ID as follows to click on the calculator buttons:

```
@Test
public void testOnePlusTwo() throws Exception {
    openCalculator();

    // Click the buttons in the user interface
    getDriver().findElement(By.id("button_1")).click();
    getDriver().findElement(By.id("button_+")).click();
    getDriver().findElement(By.id("button_2")).click();
    getDriver().findElement(By.id("button_=")).click();

    // Get the result label value
    assertEquals("3.0", getDriver().findElement(
        By.id("display")).getText());
}
```

The ID selectors are used extensively in the TestBench examples.

Finding by Vaadin Selector

In addition to the Selenium selectors, Vaadin TestBench provides a *Vaadin selector*, which allows pointing to a Vaadin component by its layout path. The JUnit test cases saved from the Recorder use Vaadin selectors by default.

You can create a Vaadin selector with the `By.vaadin()` method. You need to use the Vaadin **By**, defined in the `com.vaadin.testbench` package, which extends the Selenium **By**.

The other way is to use the `findElementByVaadinSelector()` method in the `TestBenchCommands` interface. It returns the **WebElement** object.

A Vaadin selector begins with an application identifier. It is the path to application without any slashes or other special characters. For example, `/book-examples/tobetested` would be `bookexamplestobetested`. After the identifier, comes two colons `::`, followed by a slash-delimited component path to the component to be selected. The elements in the component path are client-side classes of the Vaadin user interface components. For example, the server-side **VerticalLayout** component has **VVerticalLayout** client-side counterpart. All path elements except the leaves are component containers, usually layouts. The exact contained component is identified by its index in brackets.

A reference to a debug ID is given with a `PID_S` suffix to the debug ID.

For example:

```
// Get the button's element.
// Use the debug ID given with setDebugId().
WebElement button = driver.findElement(By.vaadin(
    "bookexamplestobetested::PID_Smain.button"));

// Get the caption text
assertEquals("Push Me!", button.getText());

// And click it
button.click();

// Get the Label's element by full path
WebElement label = driver.findElement(By.vaadin(
    "bookexamplestobetested::VVerticalLayout[0]/"+
    "ChildComponentContainer[1]/VLabel[0]"));

// Make the assertion
assertEquals("Thanks!", label.getText());
```

Finding by CSS Class

An element with a particular CSS style class name can be selected with the `By.className()` method. CSS selectors are useful for elements which have no ID, nor can be found easily from the component hierarchy, but do have a particular unique CSS style. Tooltips are one example, as they are floating `div` elements under the root element of the application. Their `v-tooltip` style makes it possible to select them as follows:

```
// Verify that the tooltip contains the expected text
String tooltipText = driver.findElement(
    By.className("v-tooltip")).getText();
```

For a complete example, see the `AdvancedCommandsITCase.java` file in the examples.

20.5.3. Running JUnit Tests in Eclipse

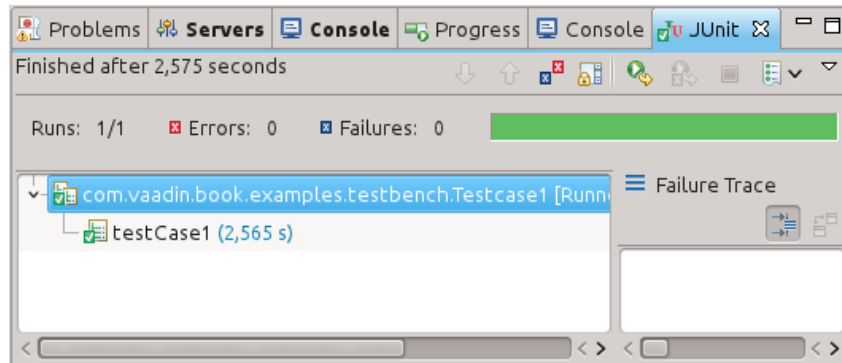
The Eclipse IDE integrates JUnit with nice control features. To run TestBench JUnit test cases in Eclipse, you need to do the following:

1. Add the TestBench JAR to a library folder in the project, such as `lib`. You should not put the library in `WEB-INF/lib` as it is not used by the Vaadin web application. Refresh the project by selecting it and pressing **F5**.
2. Right-click the project in Project Explorer and select **Properties**, and open the **Java Build Path** and the **Libraries** tab. Click **Add JARs**, navigate to the library folder, select the library, and click **OK**.
3. Switch to the **Order and Export** tab in the project properties. Make sure that the TestBench JAR is above the `gwt-dev.jar` (it may contain an old `httpClient` package), by selecting it and moving it with the **Up** and **Down** buttons.
4. Click **OK** to exit the project properties.
5. Right-click a test source file and select **Run As JUnit Test**.

A JUnit view should appear, and it should open the Firefox browser, launch the application, run the test, and then close the browser window. If all goes well, you have a passed test case, which

is reported in the JUnit view area in Eclipse, as illustrated in Figure 20.12, “Running JUnit Tests in Eclipse”.

Figure 20.12. Running JUnit Tests in Eclipse



If you are using some other IDE, it might support JUnit tests as well. If not, you can run the tests using Ant or Maven.

20.5.4. Executing Tests with Ant

Apache Ant has built-in support for executing JUnit tests. To enable the support, you need to have the JUnit library `junit.jar` and its Ant integration library `ant-junit.jar` in the Ant classpath, as described in the Ant documentation.

Once enabled, you can use the `<junit>` task in an Ant script. The following example assumes that the source files are located under a `src` directory under the current directory and compiles them to the `classes` directory. The the class path is defined with the `classpath` reference ID and should include the TestBench JAR and all relevant dependencies.

```
<project default="run-tests">
  <path id="classpath">
    <fileset dir="lib"
      includes="vaadin-testbench-standalone-*.jar" />
  </path>

  <!-- This target compiles the JUnit tests. -->
  <target name="compile-tests">
    <mkdir dir="classes" />
    <javac srcdir="src" destdir="classes"
      debug="on" encoding="utf-8">
      <classpath>
        <path refid="classpath" />
      </classpath>
    </javac>
  </target>

  <!-- This target calls JUnit -->
  <target name="run-tests" depends="compile-tests">
    <junit fork="yes">
      <classpath>
        <path refid="classpath" />
        <pathelement path="classes" />
      </classpath>

      <formatter type="brief" usefile="false" />
    </junit>
  </target>
</project>
```

```
        <batchtest>
          <fileset dir="src">
            <include name="**/*.java" />
          </fileset>
        </batchtest>
      </junit>
    </target>
  </project>
```

You also need to deploy the application to test, and possibly launch a dedicated server for it.

20.5.5. Executing Tests with Maven

Executing JUnit tests with Vaadin TestBench under Maven requires installing the TestBench library in the local Maven repository and defining it as a dependency in any POM that needs to execute TestBench tests.

A complete example of a Maven test setup is given in the `example/maven` folder in the installation package. Please see the `README` file in the folder for further instructions.

Installing TestBench in Local Repository

You can install TestBench in the local Maven repository with the following commands:

```
$ cd maven
$ mvn install:install-file \
  -Dfile=vaadin-testbench-3.0.0-SNAPSHOT.jar \
  -Djavadoc=vaadin-testbench-3.0.0-SNAPSHOT-javadoc.jar \
  -DpomFile=pom.xml
```

The `maven` folder also includes an `INSTALL` file, which contains instructions for installing TestBench in Maven.

Defining TestBench as a Dependency

Once TestBench is installed in the local repository as instructed in the previous section, you can define it as a dependency in the Maven POM of your project as follows:

```
<dependency>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin-testbench</artifactId>
  <version>3.0.0-SNAPSHOT</version>
</dependency>
```

For instructions on how to create a new Vaadin project with Maven, please see Section 2.5, “Creating a Project with Maven”.

Running the Tests

To compile and run the tests, simply execute the `test` lifecycle phase with Maven as follows:

```
$ mvn test
...
-----
T E S T S
-----
Running TestBenchExample
Tests run: 6, Failures: 2, Errors: 0, Skipped: 1, Time elapsed: 36.736 sec <<< FAILURE!

Results :
```

```
Failed tests:
  testDemo(TestBenchExample): expected:<[5/17/]12> but was:<[17.6.20]12>
  testScreenshot(TestBenchExample): Screenshots differ
```

```
Tests run: 6, Failures: 2, Errors: 0, Skipped: 1
...
```

The example configuration starts Jetty to run the application that is tested. Error screenshots from screenshot comparison are written to the `target/testbench/errors` folder. To enable comparing them to "expected" screenshots, you need to copy the screenshots to the `src/test/resources/screenshots/reference/` folder. See Section 20.6, "Taking and Comparing Screenshots" for more information regarding screenshots.

20.5.6. Test Setup

Test configuration is done in a method annotated with `@Before`. The method is executed before each test case. In a JUnit stub exported from Recorder, this is done in the `setUp()` method.

The basic configuration tasks are:

- Set TestBench parameters
- Create the web driver
- Do any other initialization

TestBench Parameters

TestBench parameters are defined with static methods in the **com.vaadin.testbench.Parameters** class. The parameters are mainly for screenshots and documented in Section 20.6, "Taking and Comparing Screenshots".

20.5.7. Creating and Closing a Web Driver

Vaadin TestBench uses Selenium WebDriver to execute tests in a browser. The **WebDriver** instance is created with the static `createDriver()` method in the **TestBench** class. It takes the driver as the parameter and returns it after registering it. The test cases must extend the **TestBenchTestCase** class, which manages the TestBench-specific features.

The basic way is to create the driver in a method annotated with the JUnit `@Before` annotation and close it in a method annotated with `@After`.

```
public class AdvancedTest extends TestBenchTestCase {
    private WebDriver driver;

    @Before
    public void setUp() throws Exception {
        ...
        driver = TestBench.createDriver(new FirefoxDriver());
    }
    ...
    @After
    public void tearDown() throws Exception {
        driver.quit();
    }
}
```

This creates the driver for each test you have in the test class, causing a new browser instance to be opened and closed. If you want to keep the browser open between the test, you can use

@BeforeClass and @AfterClass methods to create and quit the driver. In that case, the methods as well as the driver instance have to be static.

```
public class AdvancedTest extends TestBenchTestCase {
    static private WebDriver driver;

    @BeforeClass
    static public void createDriver() throws Exception {
        driver = TestBench.createDriver(new FirefoxDriver());
    }
    ...
    @AfterClass
    static public void tearDown() throws Exception {
        driver.quit();
    }
}
```

20.5.8. Basic Test Case Structure

A typical test case does the following:

1. Open the URL
2. Navigate to desired state
 - a. Find a HTML element (**WebElement**) for navigation
 - b. Use `click()` and other commands to interact with the element
 - c. Repeat with different elements until desired state is reached
3. Find a HTML element (**WebElement**) to check
4. Get and assert the value of the HTML element
5. Get a screenshot

The **WebDriver** allows finding HTML elements in a page in various ways, for example, with XPath expressions. The access methods are defined statically in the **By** class.

These tasks are realized in the following test code:

```
@Test
public void testCase1() throws Exception {
    driver.get(baseUrl + "/book-examples/tobetested");

    // Get the button's element.
    // (Actually the caption element inside the button.)
    // Use the debug ID given with setDebugId().
    WebElement button = driver.findElement(By.xpath(
        "//div[@id='main.button']/span/span"));

    // Get the caption text
    assertEquals("Push Me!", button.getText());

    // And click it. It's OK to click the caption element.
    button.click();

    // Get the Label's element.
    // Use the automatically generated ID.
    WebElement label = driver.findElement(By.xpath(
        "//div[@id='myapp-949693921']" +
        "/div/div[2]/div/div[2]/div/div"));
}
```

```
// Make the assertion
assertEquals("Thanks!", label.getText());
}
```

You can also use URI fragments in the URL to open the application at a specific state. For information about URI fragments, see Section 12.9, “URI Fragment and History Management with **UriFragmentUtility**”.

You should use the JUnit assertion commands. They are static methods defined in the `org.junit.Assert` class, which you can import (for example) with:

```
import static org.junit.Assert.assertEquals;
```

Please see the Selenium API documentation [http://seleniumhq.org/docs/03_webdriver.html#selenium-webdriver-api-commands-and-operations] for a complete reference of the element search methods in the **WebDriver** and **By** classes and for the interaction commands in the **WebElement** class.

TestBench has a collection of its own commands, defined in the `TestBenchCommands` interface. You can get a command object that you can use by calling `testBench(driver)` in a test case.

20.5.9. Waiting for Vaadin

Selenium is intended for regular web applications that load a page that is immediately rendered by the browser. Vaadin, on the other hand, is an Ajax framework where page is loaded just once and rendering is done in JavaScript. This takes more time so that the rendering might not be finished when the WebDriver continues executing the test. Vaadin TestBench allows waiting until the rendering is finished.

The waiting is automatically enabled. You can disable waiting by calling `disableWaitForVaadin()` in the `TestBenchCommands` interface. You can call it in a test case as follows:

```
testBench(driver).disableWaitForVaadin();
```

When disabled, you can wait for the rendering to finish by calling `waitForVaadin()` explicitly.

```
testBench(driver).waitForVaadin();
```

You can re-enable the waiting with `enableWaitForVaadin()` in the same interface.

20.5.10. Testing Tooltips

Component tooltips show when you hover the mouse over a component. Events caused by hovering are not recorded by Recorder, so this interaction requires special handling when testing.

Let us assume that you have set the tooltip as follows:

```
// Create a button with a debug ID
Button button = new Button("Push Me!");
button.setDebugId("main.button");

// Set the tooltip
button.setDescription("This is a tip");
```

The tooltip of a component is displayed with the `showTooltip()` method in the **TestBenchElementCommands** interface. You should wait a little to make sure it comes up. The floating tooltip

element is not under the element of the component, but you can find it by `//div[@class='v-tooltip']` XPath expression.

```
@Test
public void testTooltip() throws Exception {
    driver.get(appUrl);

    // Get the button's element.
    // Use the debug ID given with setDebugId().
    WebElement button = driver.findElement(By.xpath(
        "//div[@id='main.button']/span/span"));

    // Show the tooltip
    testBenchElement(button).showTooltip();

    // Wait a little to make sure it's up
    Thread.sleep(1000);

    // Check that the tooltip text matches
    assertEquals("This is a tip", driver.findElement(
        By.xpath("//div[@class='v-tooltip']")).getText());

    // Compare a screenshot just to be sure
    assertTrue(testBench(driver).compareScreen("tooltip"));
}
```

20.5.11. Scrolling

Some Vaadin components, such as **Table** and **Panel** have a scrollbar. To get hold of the scrollbar, you must first find the component element. Then, you need to get hold of the `TestBenchElementCommands` interface from the **WebElement** with `testBenchElement(WebElement)`. The `scroll()` method in the interface scrolls a vertical scrollbar down the number of pixels given as the parameter. The `scrollLeft()` scrolls a horizontal scrollbar by the given number of pixels.

20.5.12. Testing Notifications

When testing notifications, you will need to close the notification box. You need to get hold of the `TestBenchElementCommands` interface from the **WebElement** of the notification element with `testBenchElement(WebElement)`. The `closeNotification()` method in the interface closes the notification.

20.5.13. Testing Context Menus

Opening context menus require special handling. You need to create a Selenium **Actions** object to perform a context click on a **WebElement**.

In the following example, we open a context menu in a **Table** component, find an item by its caption text, and click it.

```
// Select the table body element
WebElement e = getDriver().findElement(
    By.className("v-table-body"));

// Perform context click action to open the context menu
new Actions(getDriver()).moveToElement(e)
    .contextClick(e).perform();

// Select "Add Comment" from the opened menu
getDriver().findElement(
    By.xpath("//*[text() = 'Add Comment']")).click();
```

The complete example is given in the `AdvancedCommandsITCase.java` example source file.

20.5.14. Profiling Test Execution Time

It is not just that it works, but also how long it takes. Profiling test execution times consistently is not trivial, as a test environment can have different kinds of latency and interference. For example in a distributed setup, timings taken on the test server would include the latencies between the test server, the grid hub, a grid node running the browser, and the web server running the application. In such a setup, you could also expect interference between multiple test nodes, which all might make requests to a shared application server and possibly also share virtual machine resources.

Furthermore, in Vaadin applications, there are two sides which need to be profiled: the server-side, on which the application logic is executed, and the client-side, where it is rendered in the browser. Vaadin TestBench includes methods for measuring execution time both on the server-side and the client-side.

The `TestBenchCommands` interface offers the following methods for profiling test execution time:

`totalTimeSpentServicingRequests()`

Returns the total time (in milliseconds) spent servicing requests in the application on the server-side. The timer starts when you first navigate to the application and hence start a new session. The time passes only when servicing requests for the particular session. The timer is shared in the servlet session, so if you have, for example, multiple portlets in the same application (session), their execution times will be included in the same total.

Notice that if you are also interested in the client-side performance for the last request, you must call the `timeSpentRenderingLastRequest()` before calling this method. This is due to the fact that this method makes an extra server request, which will cause an empty response to be rendered.

`timeSpentServicingLastRequest()`

Returns the time (in milliseconds) spent servicing the last request in the application on the server-side. Notice that not all user interaction through the `WebDriver` cause server requests.

As with the total above, if you are also interested in the client-side performance for the last request, you must call the `timeSpentRenderingLastRequest()` before calling this method.

`totalTimeSpentRendering()`

Returns the total time (in milliseconds) spent rendering the user interface of the application on the client-side, that is, in the browser. This time only passes when the browser is rendering after interacting with it through the `WebDriver`. The timer is shared in the servlet session, so if you have, for example, multiple portlets in the same application (session), their execution times will be included in the same total.

`timeSpentRenderingLastRequest()`

Returns the time (in milliseconds) spent rendering user interface of the application after the last server request. Notice that not all user interaction through the `WebDriver` cause server requests.

If you also call the `timeSpentServicingLastRequest()` or `totalTimeSpentServicingRequests()`, you should do so before calling this method. The methods cause a server request, which will zero the rendering time measured by this method.

Generally, only interaction with fields in the *immediate* mode cause server requests. This includes button clicks. Some components, such as **Table**, also cause requests otherwise, such as when loading data while scrolling. Some interaction could cause multiple requests, such as when images are loaded from the server as the result of user interaction.

The following example is given in the `VerifyExecutionTimeITCase.java` file under the TestBench examples.

```
@Test
public void verifyServerExecutionTime() throws Exception {
    openCalculator();

    // Get start time on the server-side
    long currentSessionTime = testBench(getDriver())
        .totalTimeSpentServicingRequests();

    // Interact with the application
    calculateOnePlusTwo();

    // Calculate the passed processing time on the serve-side
    long timeSpentByServerForSimpleCalculation = testBench()
        .totalTimeSpentServicingRequests() - currentSessionTime;

    // Report the timing
    System.out.println("Calculating 1+2 took about "
        + timeSpentByServerForSimpleCalculation
        + "ms in servlets service method.");

    // Fail if the processing time was critically long
    if (timeSpentByServerForSimpleCalculation > 30) {
        fail("Simple calculation shouldn't take "
            + timeSpentByServerForSimpleCalculation + "ms!");
    }

    // Do the same with rendering time
    long totalTimeSpentRendering =
        testBench().totalTimeSpentRendering();
    System.out.println("Rendering UI took " +
        totalTimeSpentRendering + "ms");
    if (timeSpentByServerForSimpleCalculation > 400) {
        fail("Rendering UI shouldn't take "
            + timeSpentByServerForSimpleCalculation + "ms!");
    }

    // A regular assertion on the UI state
    assertEquals("3.0", getDriver().findElement(
        By.id("display")).getText());
}
```

20.6. Taking and Comparing Screenshots

You can take and compare screenshots with reference screenshots taken earlier. If there are differences, you can fail the test case.

20.6.1. Screenshot Parameters

The screenshot configuration parameters are defined with static methods in the **com.vaadin.testbench.Parameters** class.

screenshotErrorDirectory (default: null)

Defines the directory where screenshots for failed tests or comparisons are stored.

screenshotReferenceDirectory (default: null)

Defines the directory where the reference images for screenshot comparison are stored.

captureScreenshotOnFailure (default: true)

Defines whether screenshots are taken whenever an assertion fails.

screenshotComparisonTolerance (default: 0.01)

Screen comparison is usually not done with exact pixel values, because rendering in browser often has some tiny inconsistencies. Also image compression may cause small artifacts.

screenshotComparisonCursorDetection (default: false)

Some field component get a blinking cursor when they have the focus. The cursor can cause unnecessary failures depending on whether the blink happens to make the cursor visible or invisible when taking a screenshot. This parameter enables cursor detection that tries to minimize these failures.

maxScreenshotRetries (default: 2)

Sometimes a screenshot comparison may fail because the screen rendering has not yet finished, or there is a blinking cursor that is different from the reference screenshot. For these reasons, Vaadin TestBench retries the screenshot comparison for a number of times defined with this parameter.

screenshotRetryDelay (default: 500)

Delay in milliseconds for making a screenshot retry when a comparison fails.

For example:

```
@Before
public void setUp() throws Exception {
    Parameters.setScreenshotErrorDirectory(
        "screenshots/errors");
    Parameters.setScreenshotReferenceDirectory(
        "screenshots/reference");
    Parameters.setMaxScreenshotRetries(2);
    Parameters.setScreenshotComparisonTolerance(1.0);
    Parameters.setScreenshotRetryDelay(10);
    Parameters.setScreenshotComparisonCursorDetection(true);
    Parameters.setCaptureScreenshotOnFailure(true);
}
```

20.6.2. Taking Screenshots on Failure

Vaadin TestBench takes screenshots automatically when a test fails, if the *captureScreenShotOnFailure* is enabled in TestBench parameters. The screenshots are written to the error directory defined with the *screenshotErrorDirectory* parameter.

You need to have the following in the setup method:

```
@Before
public void setUp() throws Exception {
    Parameters.setScreenshotErrorDirectory("screenshots/errors");
    Parameters.setCaptureScreenshotOnFailure(true);
    ...
}
```

20.6.3. Taking Screenshots for Comparison

Vaadin TestBench allows taking screenshots of the web browser window with the `compareScreen()` command in the **TestBenchCommands** interface. The method has a number of variants.

The `compareScreen(File)` takes a **File** object pointing to the reference image. In this case, a possible error image is written to the error directory with the same file name. You can get a file object to a reference image with the static `ImageFileUtil.getReferenceScreenshotFile()` helper method.

```
assertTrue("Screenshots differ",
    testBench(driver).compareScreen(
        ImageFileUtil.getReferenceScreenshotFile(
            "myshot.png")));
```

The `compareScreen(String)` takes a base name of the screenshot. It is appended with browser identifier and the file extension.

```
assertTrue(testBench(driver).compareScreen("tooltip"));
```

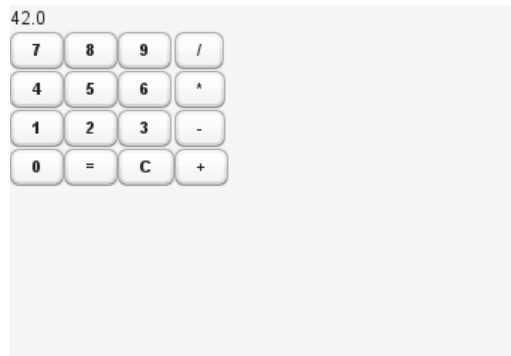
The `compareScreen(BufferedImage, String)` allows keeping the reference image in memory. An error image is written to a file with a name determined from the base name given as the second parameter.

Screenshots taken with the `compareScreen()` method are compared to a reference image stored in the reference image folder. If differences are found (or the reference image is missing), the comparison method returns `false` and stores the screenshot in the error folder. It also generates an HTML file that highlights the differing regions.

Screenshot Comparison Error Images

Screenshots with errors are written to the error folder, which is defined with the *screenshotErrorDirectory* parameter described in Section 20.6.1, “Screenshot Parameters”.

For example, the error caused by a missing reference image could be written to `screenshot/errors/tooltip_firefox_12.0.png`. The image is shown in Figure 20.13, “A screenshot taken by a test run”.

Figure 20.13. A screenshot taken by a test run

Screenshots cover the visible page area in the browser. The size of the browser is therefore relevant for screenshot comparison. The browser is normally sized with a predefined default size. You can set the size of the browser window with, for example, `driver.manage().window().setSize(new Dimension(1024, 768));` in the `@Before` method. The size includes any browser chrome, so the actual screenshot size will be smaller.

Reference Images

Reference images are expected to be found in the reference image folder, as defined with the `screenshotReferenceDirectory` parameter described in Section 20.6.1, “Screenshot Parameters”. To create a reference image, just copy a screenshot from the `errors/` directory to the `reference/` directory.

For example:

```
$ cp screenshot/errors/tooltip_firefox_12.0.png screenshot/reference/
```

Now, when the proper reference image exists, rerunning the test outputs success:

```
$ java ...
JUnit version 4.5
.
Time: 18.222

OK (1 test)
```

You can also supply multiple versions of the reference images by appending an underscore and an index to the filenames. For example:

```
tooltip_firefox_12.0.png
tooltip_firefox_12.0_1.png
tooltip_firefox_12.0_2.png
```

This can be useful in certain situations when there actually are more than one “correct” reference.

Masking Screenshots

You can make masked screenshot comparison with reference images that have non-opaque regions. Non-opaque pixels in the reference image, that is, ones with less than 1.0 value, are ignored in the screenshot comparison.

Visualization of Differences in Screenshots with Highlighting

Vaadin TestBench supports advanced difference visualization between a captured screenshot and the reference image. A difference report is written to a HTML file that has the same name as the failed screenshot, but with `.html` suffix. The reports are written to the same `errors/` folder as the screenshots from the failed tests.

The differences in the images are highlighted with blue rectangles. Moving the mouse pointer over a square shows the difference area as it appears in the reference image. Clicking the image switches the entire view to the reference image and back. Text **"Image for this run"** is displayed in the top-left corner to identify the currently displayed screenshot.

Figure 20.14, "The reference image and a highlighted error image" shows a difference report with three differences. Date fields are a typical cause of differences in screenshots.

Figure 20.14. The reference image and a highlighted error image



20.6.4. Practices for Handling Screenshots

Access to the screenshot reference image directory should be arranged so that a developer who can view the results can copy the valid images to the reference directory. One possibility is to store the reference images in a version control system and check-out them to the `reference/` directory.

A build system or a continuous integration system can be configured to automatically collect and store the screenshots as build artifacts.

20.6.5. Known Compatibility Problems

Screenshots when running Internet Explorer 9 in Compatibility Mode

Internet Explorer prior to version 9 adds a two-pixel border around the content area.

Version 9 no longer does this and as a result screenshots taken using Internet Explorer 9 running in compatibility mode (IE7/IE8) will include the two pixel border, contrary to what the older versions of Internet Explorer do.

20.7. Running Tests in an Distributed Environment

A distributed test environment consists of a grid hub and a number of test nodes. The hub listens to calls from test runners and delegates them to the grid nodes. Different nodes can run on different operating system platforms and have different browsers installed.

A basic distributed installation was covered in Section 20.2.2, “A Distributed Testing Environment”.

20.7.1. Running Tests Remotely

Remote tests are just like locally executed JUnit tests, except instead of using a browser driver, you use a **RemoteWebDriver** that can connect to the hub. The hub delegates the connection to a grid node with the desired capabilities, that is, which browsers are installed in a suitable node. The capabilities are described with a **DesiredCapabilities** object.

For example, in the example tests given in the `example` folder, we create and use a remote driver as follows:

```
@Test
public void testRemoteWebDriver() throws MalformedURLException {
    // Require Firefox in the test node
    DesiredCapabilities capability =
        DesiredCapabilities.firefox();

    // Create a remote web driver that connects to a hub
    // running in the local host
    WebDriver driver = TestBench.createDriver(
        new RemoteWebDriver(new URL(
            "http://localhost:4444/wd/hub"), capability));

    // Then use it to run a test as you would use any web driver
    try {
        driver.navigate().to(
            "http://demo.vaadin.com/sampler#TreeActions");
        WebElement e = driver.findElement(By.xpath(
            "//div[@class='v-tree-node-caption']"+
            "/div[span='Desktops']"));
        new Actions(driver).moveToElement(e).contextClick(e)
            .perform();
    } finally {
        driver.quit();
    }
}
```

Running the example requires that the hub service and the nodes are running. Starting them is described in the subsequent sections. Please refer to Selenium documentation [http://seleniumhq.org/docs/07_selenium_grid.html] for more detailed information.

20.7.2. Starting the Hub

The TestBench grid hub listens to calls from test runners and delegates them to the grid nodes. The grid hub service is included in the Vaadin TestBench JAR and you can start it with the following command:

```
$ java -jar \
    vaadin-testbench-standalone-3.0.0.jar \
    -role hub
```

You can open the control interface of the hub also with a web browser. Using the default port, just open URL `http://localhost:4444/`. Once you have started one or more grid nodes, as instructed in the next section, the "console" page displays a list of the grid nodes with their browser capabilities.

20.7.3. Starting a Grid Node

A TestBench grid node listens to calls from the hub and is capable of opening a browser. The grid node service is included in the Vaadin TestBench JAR and you can start it with the following command:

```
$ java -jar \
    vaadin-testbench-standalone-3.0.0.jar \
    -role node \
    -hub http://localhost:4444/grid/register
```

The node registers itself in the grid hub and you need to give the address of the hub with the `-hub` parameter.

You can run one grid node in the same host as the hub, as is done in the example above with the localhost address. In such case notice that, at least in OS X, you may need to duplicate the JAR to a separate copy to use it to run a grid node service.

20.7.4. Mobile Testing

Vaadin TestBench includes an iPhone and an Android driver, with which you can test on mobile devices. The tests can be run either in a device or in an emulator/simulator.

The actual testing is just like with any WebDriver, using either the **IPhoneDriver** or the **AndroidDriver**. The Android driver assumes that the hub (`android-server`) is installed in the emulator and forwarded to port 8080 in localhost, while the iPhone driver assumes port 3001. You can also use the **RemoteWebDriver** with either the `iphone()` or the `android()` capability, and specify the hub URI explicitly.

The mobile testing setup is covered in detail in the Selenium documentation for both the `IPhoneDriver` [<http://code.google.com/p/selenium/wiki/IPhoneDriver>] and the `AndroidDriver` [<http://code.google.com/p/selenium/wiki/AndroidDriver>].

20.8. Known Issues

This section provides information and instructions on a few features that are known to be difficult to use or need modification to work.

20.8.1. Testing the LoginForm

Replaying interactions in the **LoginForm** component is somewhat tricky due to the fact that the identifier of the `iframe` element that is used changes each time the application is restarted. The use of an `iframe` element means that recordings have to select the target frame before fields can be correctly identified.

Selecting the correct frame in the *recorder* can be done by modifying the *target* of the recorded **selectFrame** command. One way to identify the correct frame is by specifying its 0 based index:

```
index=2
```

Another way is to specify an XPath selector for the `iframe`. In this case it helps if you can specify a debug ID for the LoginForm component. In the example below, the **LoginForm**'s debug ID is "login".

```
xpath=//id("login")//iframe
```

Selecting the correct frame in *JUnit code* is done like in the following example.

```
getDriver().switchTo().frame(2);
```

Or

```
WebElement frame = getDriver().findElement(  
    By.xpath("//div[@id='login']/iframe"));  
getDriver().switchTo().frame(frame);
```

20.8.2. Using `assertTextPresent` and `assertTextNotPresent`

The `assertTextNotPresent` and `assertTextNotPresent` methods in TestBench Recorder are problematic in that they do not respect CSS rules that hide elements (e.g. `display: none;`). This means that you might have trouble confirming that text is or is not present.

This will be fixed in a future release, but until then a better, albeit more complex, strategy for testing whether a string is present on screen is to use the `assertElementPresent` method with an XPath selector as follows:

```
xpath=//div[contains(@style, "display: none")]//div[contains(text(),"HIDDEN TEXT")]
```

20.8.3. Exporting Recordings of the Upload Component

Exporting recordings of the **Upload** component exports one piece of unnecessary code that makes replay fail. Whenever you record a file upload, remember to remove the call to `clear()` as upload fields are special fields that cannot be cleared. Also make sure that the replay window is wide enough for the upload button to be visible (see `resizeViewPortTo()`), otherwise you will get an exception when running the test.