Getting Started with Vaadin TestBench 4

Introduction

Vaadin TestBench is a browser automation tool, suited for creating UI level tests for Vaadin applications. It is based on Selenium 2, which means that all features of Selenium 2/WebDriver are also available in TestBench.

This document will give you the necessary information to get started with TestBench. It will show you how to create a runnable test, how to actually control a browser instance and finally verify the results.

At the end of this document, you'll find links to sites around the Internet, where you can learn more about testing with TestBench and Selenium.

Recommended Project Structure

It is recommended to create a separate project or module for your TestBench tests, as running a comprehensive UI test suite is going to take some time. There is also the possible issue of dependency clashes, where TestBench depends on one version of a certain library and your application depends on some other, possibly incompatible, version. By isolating the TestBench tests into a separate project, the risk of dependency clashes is minimized.

Installing TestBench 4

Vaadin TestBench 4 comes packaged in two different flavors: standalone and Maven. The standalone package contains TestBench and all required dependencies, while the Maven package contains only TestBench itself and can be used through e.g. Maven or Ivy.

Note that some drivers, notably `InternetExplorerDriver` and `ChromeDriver` require you to install executables on the system. See the respective subsections below for instructions.

Standalone

Installing the standalone package in your project is as easy as dropping the `vaadin-testbench-standalone-4.0.0-alpha1.jar` in the `lib` directory of your Java™ project, or whichever directory your project loads JAR packages from.

Note: When using the standalone package, you need to be aware that it contains lots of dependencies, which means that there might be dependency clashes with your own project unless you follow the recommended project structure outlined above.

Maven

Add the TestBench dependency to any Java project:

```
<dependency>
      <groupId>com.vaadin</groupId>
      <artifactId>vaadin-testbench</artifactId>
      <version>4.0.0-alpha1</version>
      <scope>test</scope>
</dependency>
```

Ivy

Add the Ivy dependency to any Java project:

```
<dependency org="com.vaadin" name="vaadin-testbench"
    rev="4.0.0-alpha1" conf="test -> default" />
```

IEDriverServer

To run on Internet Explorer, you first need to install `IEDriverServer.exe` on your system path (`%PATH%`). The executable can be downloaded from http://code.google.com/p/selenium/downloads/list.

ChromeDriver

Running on Google Chrome also requires a procedure similar to Internet Explorer. You need to install `chromedriver` / `chromedriver.exe` on your system path (`%PATH%`) or define the

`webdriver.chrome.driver` java system property to point to the driver executable, e.g. `-Dwebdriver.chrome.driver=/path/to/driver/chromedriver`

FirefoxDriver on Mac OS X

Firefox requires focus in order for many input events to be triggered correctly, e.g. no focus events will be fired unless the entire window has focus. Unfortunately, due to various implementation realities, running a TestBench test on Mac OS X causes the java process running the unit test to be promoted to a UI process, moving focus away from any open browser window to the "blind" process. In order to avoid this and keep the browser window focused, the java process needs to be started with the `-Djava.awt.headless=true` JVM parameter. You should be able to specify this JVM parameter in the run configuration view of your favorite IDE.

Your First Test

Follow these steps to make your very first Vaadin TestBench 4 test:

1. Create a new Java file in the package of your choice the tests source directory
2. Make the new class extend `TestBenchTestCase`
3. Add a `@Before` method, which calls `setDriver(new PreferredDriver())` where `PreferredDriver` can be e.g. `FirefoxDriver`, `ChromeDriver`, etc.
4. Add an `@After` method, which quits the driver (`getDriver().quit()`)
5. Now you can add your first `@Test` method. Something like the following, which goes to the Vaadin Quicktickets Dashboard demo and checks that it is deployed and started:

```
@Test
public void testQuickticketsDeployed() throws Exception {
    getDriver()
       .get("http://demo.vaadin.com/dashboard?restartApplication");
    assertTrue(getDriver().getPageSource()
       .contains("Welcome to the Dashboard Demo Application"));
}
```

Your test can now be run like any other JUnit test in your IDE or on your CI server.

Finding Components / Elements

The ElementQuery API

TestBench 4 brings you new ways of finding and selecting Vaadin components in your applications. Using the ElementQuery API, you can easily find the element you are interested in. We will go through a few examples to get you started. Note that the ElementQuery API ***requires*** Vaadin 7.2.alpha1 or later.

```
List<ButtonElement> buttons = $(ButtonElement.class).all();
```

Will find all button components that are visible in your application.

Let's break down the first example a bit. The `ButtonElement` class is a specialized `WebElement` (part of the WebDriver API), which contains API for interacting with Vaadin Button components[1]. There are `XyzElement` classes for all the core Vaadin components, explore the API using your IDE's code completion feature.

The `$(ButtonElement.class)` is actually a method call which instantiates an `ElementQuery` class to search for Vaadin Buttons. In this case we ask for all buttons, but `ElementQuery` provides API for filtering the results.

> [1)] In this case, the ButtonElement implementation is empty as the backing WebElement holds all necessary functionality in this case, but for more complex Vaadin components the *Element class holds API for the component in question.

The ElementQuery API contains the following filtering methods.

- `caption(String)` – selects the component by it's caption. This is very useful for finding uniquely captioned components, which don't have an ID.
- `index(int)` – choose a specific component from the many that matches the filter.
- `id(String)` – selects the component by it's ID. IDs can be assigned by calling *component*`.setId(myId)` in your Vaadin application. Using `id()` automatically triggers the search and returns the matching element.

After filtering, these methods can be used for triggering the search and getting the final result:

- `first()` – returns the first matching element/component
- `all()` – returns a list of all matching elements/components
- `get(int)` – returns the nth matching element/component. Indexes start from zero.

Element queries can be chained to utilize the hierarchy to narrow down matches, so you can find all Buttons that are somewhere inside of a VerticalLayout, i.e. each button that has a vertical layout somewhere in its parent hierarchy, by issuing

```
$(VerticalLayoutElement.class).$(ButtonElement.class).all();
```

Or you can find only the buttons that are *direct descendants* of a VerticalLayout by issuing

```
$(VerticalLayoutElement.class).$$(ButtonElement.class).all();
```

Note the double dollar-sign notation. See the examples below for more details.

A Few Examples

```
$(ButtonElement.class).id("cancel");
```

Finds the button with the ID "cancel".

```
$(ButtonElement.class).caption("Submit").first();
```

Finds the (first) button with the caption "Submit".

```
$(GridLayoutElement.class).$(ButtonElement.class).caption("Submit").first();
```

Finds a the “Submit” button that is inside a GridLayout, i.e. there is a GridLayout ancestor somewhere in the component hierarchy.

```
$(PanelElement.class).caption("Details")
      .$(ButtonElement.class).caption("Submit").first();
```

Finds a the “Submit” button that is inside the Panel with the caption “Details”.

```
$(HorizontalLayoutElement.class)
      .$$(ButtonElement.class).caption("Submit").first();
```

Finds a the “Submit” button that is the direct descendant of a HorizontalLayout.

```
$(PanelElement.class).index(1).$(ButtonElement.class).index(12).first();
$(PanelElement.class).index(1).$(ButtonElement.class).get(12);
```

Both find the 13th button inside the second Panel.

```
$(UIElement.class).$$(VerticalLayoutElement.class)
      .$$(ButtonElement.class).first();
```

Finds the first button in the root layout of the application.

The Debug Window

Vaadin 7.2 and later also contain an extension to the debug window, where you can easily point and click on components to generate an ElementQuery string for them. Just start your application in debug mode and add the ?debug URI parameter, after which you can open the TestBench tab in the debug window. There you will be able to click the “target” icon, which will enable picking of components.

The code lines generated in the debug window all contain the complete variable declaration, so you can copy and paste the lines directly into your JUnit code and start referencing them. The code can be pasted as local variables in a test method.

Classic Methods

You can also use the classic Selenium/WebDriver ways of finding elements: XPath, CSS selector, etc. If you are using a Vaadin version older than 7.2.0.alpha1, you are required to use these as the method above relies on functionality introduced in Vaadin 7.2.0.alpha1.

The classic methods are all exposed through the `driver.findElement(By)` method. Most often you’ll use `By.id()` for selecting by ID, `By.className()` for selecting by CSS class name, or `By.xpath()` or `By.cssSelector()` for more complex rules.

You can develop and debug CSS or XPath locator strings against your running application in e.g. Google Chrome by opening the JavaScript console and issuing the following function calls:

`$x("//my/xpath/expression")`
: Shows you the element(s) matching the XPath expression.

`$$("my CSS.selector")`
: Shows you the element(s) matching the CSS selector.

Here's an example that selects the currently selected row in a Table. You can try it out in Vaadin Sampler at http://demo.vaadin.com/sampler/#ui/data-presentation/table

```
$x("//table//tr[contains(@class,'v-selected')]");
$$(".v-table tr.v-selected");
```

There's also a "Copy XPath" option in the context menu for elements in Chrome's elements inspector. Use it to easily get a working, albeit unnecessarily long and fragile, XPath for any element. This XPath expression can then be reduced and refined using the technique above.

Asserting Values

Asserting values works just like in any JUnit (or other testing framework) test. Values are read using the API provided by the different element classes (e.g. `TextFieldElement.getValue()`) or the WebDriver API (e.g. `WebElement.getAttribute("value")`) and then asserted using the methods in the `Assert` class.

More Information / References

http://vaadin.com/addon/vaadin-testbench	*Vaadin Directory*. Download the latest version of Vaadin TestBench from here.
https://vaadin.com/book/-/page/testbench.html	*The Vaadin TestBench manual*. Contains in-depth information on Vaadin locators, setting up a distributed grid, CI server integration and more.
http://docs.seleniumhq.org/docs/ http://code.google.com/p/selenium/w/list	The Selenium documentation and wiki. In-depth information on how Selenium/WebDriver works, the API documentation and much more.
http://dev.vaadin.com	The Vaadin bug tracker