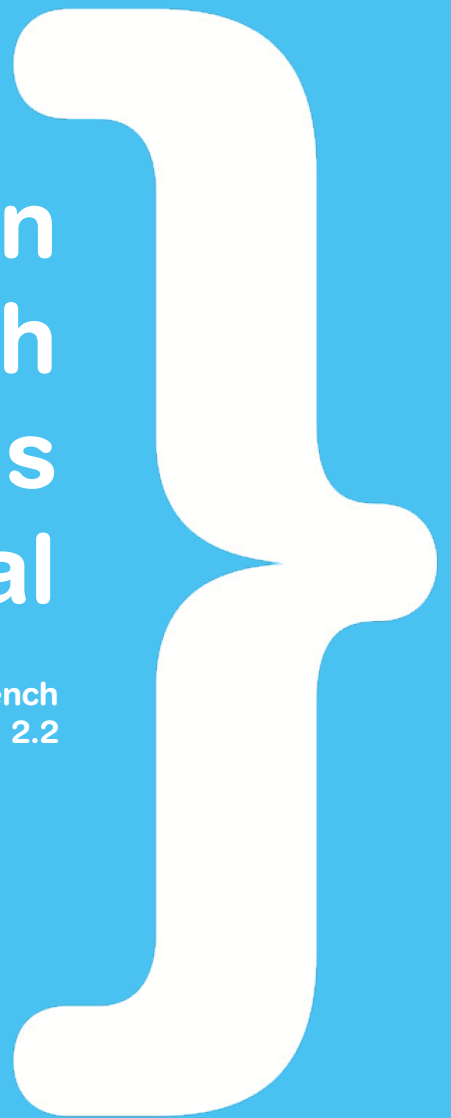# Vaadin TestBench User's Manual

Vaadin TestBench
2.2

Vaadin TestBench is an environment used for automated user interface regression testing of Vaadin applications on multiple platforms and browsers.

Vaadin TestBench allows you to run tests on the UI after a build and catch problems created by changes to the business logic. This helps you catch problems that affect the UI functionality early on, before they become a real problem.

Table of Contents

1. Introduction to Vaadin TestBench

1.1. Overview

Quality assurance is one of the cornerstones of modern software development. Extending throughout the entire development process, quality assurance is what 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.

Vaadin TestBench makes it possible to automate the regression testing of web applications that use Vaadin. You record test cases by interacting with your application. After recording, you can compile the test as JUnit tests and run them for as many times as you want, on multiple platforms and browsers. The test results can be collected for later analysis and quality assurance.

The main features are:

➔ Recording and playing back test cases using a recorder in browser

➔ Validating UI state by assertion points and screen capture comparison

➔ Screen capture comparison with difference highlighting

➔ Execution of tests through **JUnit**

➔ Distributed test grid for running tests

➔ Integration with unit testing

Execution of tests are 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 recording the tests, you can use Vaadin TestBench on just a single computer.

Vaadin TestBench is based on the **Selenium** testing framework and **Selenium Grid** for distributed testing. Selenium is augmented with Vaadin-specific extensions, such as the screen capture feature.

thinking of U and I **vaadin }>**

1.2. TestBench Components

The main components of Vaadin TestBench are:

→ Vaadin TestBench Recorder

→ Vaadin TestBench Library

→ Vaadin TestBench Grid Hub

→ Vaadin TestBench Grid Remote Control

The components and a basic setup are illustrated in Figure   1.
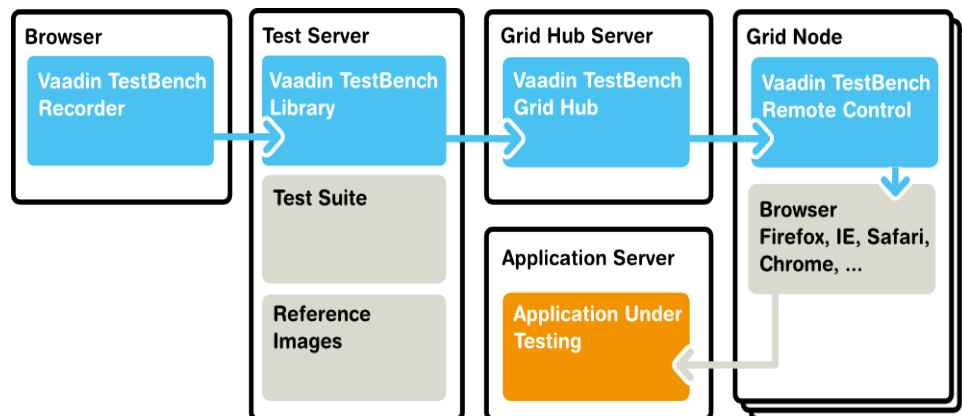
Figure 1: Vaadin TestBench Architecture

Recording test cases requires **Vaadin TestBench Recorder**, which is a Mozilla Firefox extension that you install in your browser. It provides a control panel to record test cases and play them back. You can play test cases right in the recorder and later automatically by the Grid Node Remote Control, which opens a browser and starts the recorder in playback mode.

The test suite and results from test runs are stored on a test server. A test suite includes recorded test cases and possible reference images for similarity tests.

Vaadin TestBench Library provides the central control logic for:

→ Converting tests from recordings to JUnit tests

→ Comparing screen captures with reference images

→ Controlling the execution of tests on the VaadinTestBench Grid Hub

Vaadin TestBench Grid Hub is a service that distributes testing tasks to nodes on the test grid. It contacts a Grid Node Controller on a node and asks it to open a browser to run the current test on. The Grid Hub runs on a server which can be dedicated, one of the grid nodes, or the test server itself.

Vaadin TestBench Grid Node Controller is a service that runs on each grid node. It is able to open any of the browsers installed in a node and run commands for tests on the browsers. It receives requests to execute test commands from the Grid Hub and reports the results back.

A basic Vaadin TestBench environment consists of:

→ A workstation with Firefox and the TestBench Recorder for recording tests

→ A build/test server used to build, launch, and test the web application

→ A server running the Grid Hub

→ One or more servers running Grid Remote Controls

The workstation and servers can be separate computers or one computer can work in multiple roles.

1.3. Requirements

1.3.1. Requirements for Vaadin TestBench Recorder

For recording and playback with Vaadin TestBench Recorder:

→ Mozilla Firefox 3.x or newer

1.3.2. Requirements for Automated Testing

For running tests:

→ Java JDK 1.5 or newer

→ Browsers installed on test nodes

→ Apache Ant or some other way to run Ant scripts (recommended)

1.3.3. Continuous Integration Compatibility

Vaadin TestBench works with continuous integration systems that support JUnit testing. It is tested to work on the TeamCity build management and continuous integration server.

1.3.4. Known Compatibility Problems

Firebug should be disabled

In some cases Firebug injects a `<div id="_firebugConsole">` element under the `<body>` element (the element is invisible in the Firebug's HTML structure browser). This can disturb recording of test cases, especially when closing notifications. Firebug 1.6 should fix this issue, but we still encourage TestBench users to disable Firebug when recording or playing test cases.

1.4. Limitations

Vaadin TestBench has currently the following limitations:

→ Playback for **Modifier+Key** combination only works in Mozilla Firefox and Internet Explorer

→ Tests do not work with the RichTextField component

thinking of U and I **vaadin }>**

1.4.1.   Common Problems

➜ For the LoginForm component, Recorder will record a `selectFrame` command when the LoginForm gets the focus and `selectWindow` when the focus is changed elsewhere. You should remove these commands to get the test to work correctly.

thinking of U and I **vaadin** }>

2. Installing Vaadin TestBench

2.1. Overview

Installation of Vaadin TestBench covers the following tasks:

→ Download and unpack the Vaadin TestBench installation package

→ Install Vaadin TestBench Recorder

→ Install Vaadin TestBench Library and launch scripts

→ Install Vaadin TestBench Grid Hub

→ Install Vaadin TestBench Grid Remote Controls

At first you only need to install the Recorder. It allows you to record and play back tests in the browser. The rest of the installation tasks are for running automated tests from a test server using a grid.

Two basic installation types are covered in these instructions:

→ Test development installation on a workstation

→ Distributed grid installation

2.1.1. Test Development Installation

In a typical small test development setup, you will install all the components on a single workstation. See Section 2.4 : *Quick Setup for Playback on a Workstation* for a quick test development setup.

The default values in the `example/test.xml` Ant script in the installation package assumes that the hub is installed on the localhost where the script is executed. With the base setup the hub and remote control are launched on the localhost.

2.1.2. A Distributed Test Environment

A Vaadin TestBench grid consists of two categories of components:

→ Vaadin TestBench Grid Hub service

→ Grid nodes running Vaadin TestBench Grid Remote Control

The hub is a service that handles communication between the JUnit test runner and the node controllers. The node controllers are services that can launch a browser and 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.

In a fully distributed setup, you install the Vaadin TestBench components on separate hosts.

thinking of U and I vaadin }>

2.2. Downloading and Unpacking the Installation Package

First, download the installation package `vaadin-testbench-2.2.0.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`.

Warning: 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 occurs, 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, use Info-ZIP or other ZIP software with the command:

```
$ unzip vaadin-testbench-2.2.0.zip
```

The contents of the installation package will be extracted under `vaadin-testbench` installation directory in the chosen directory.

2.3. Installing the Recorder

An environment for developing tests require you to use the Vaadin TestBench Recorder for recording test cases and to play them back.

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

1. Enter the `testbench-recorder` directory under the installation directory.

2. Open Mozilla Firefox

3. Either drag and drop the `vaadin-testbench-recorder-2.2.0.xpi` to an open Firefox window or open it from the **File**

4. Firefox will ask if you want to install the TestBench Recorder extension. Click **Install Now**.

thinking of U and I **vaadin** }>
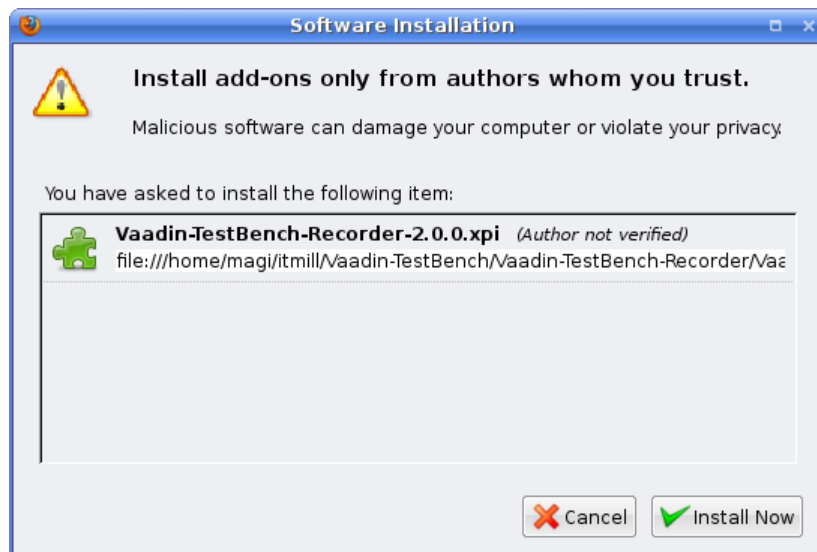
Figure 2: Installing Vaadin TestBench Recorder

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

The installation of a new version of Vaadin TestBench Recorder will overwrite a 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/colorpicker.

2.4. Quick Setup for Playback on a Workstation

You can run the grid hub and a remote control on your workstation when developing tests. The configuration uses the localhost as default, so you do not need to edit the configuration files at this point.

Windows

1. Navigate to `grid/hub/` in the installation directory and run `hub.bat` to start the hub on port 4444.

2. Navigate to `grid/remote-control/` in the installation directory and run rc.bat to start the remote control on the local machine and have it connect to the hub.

Linux, Mac OS X, and other UNIX

1. Open a terminal window and change to `grid/hub/` under the installation directory

2. Edit hub configuration file `grid_configuration.yml` and check that it includes your environment, as described in Section 2.5.2 : *Predefined Target Environments*. Especially in Linux, if using Firefox, edit the path to the `firefox-bin` executable.

3. Run `hub.sh` to start the hub on port 4444.

thinking of U and I **vaadin** }>

```
$ cd vaadin-testbench-2.2.0/grid/hub
$ sh hub.sh
```

4. Open a second terminal window and change to `grid/node-controller/` under the installation directory

5. Edit remote configuration file `rc_configuration.xml`, disable the Windows targets and enable the targets for your environment, as described in Section 2.6.1: *Configuring the Remote Control*.

6. Run `rc.bat` to start the node controller on the local machine and have it connect to the hub.

```
$ cd vaadin-testbench-2.2.0/grid/remote-control
$ sh rc.sh
```

Check that the remote control registers correctly by opening http://localhost:4444/console in a web browser.

Next, change to the `example/` directory.

```
$ cd vaadin-testbench-2.2.0/example
```

Open the `test.xml` file in an editor and edit the list of browsers (target environments names) to match your system and the installed browsers.

```
<property name="browsers"
          value="winxp-ie8,winxp-firefox3" />
```

The value must be a comma-separated list of target names. A complete list of predefined browsers (target environments) is given in Section 2.5.1: *Configuring the Hub*. If your environment is not included in the predefined entries, you need to edit the hub and remote control configurations before starting them. Notice that the target name is merely an alias for a browser identifier, and you could use "`winxp-firefox3`" in Linux or Mac OS X as well, assuming that Firefox is installed on the host.

When you feel that the configuration is OK, run the script:

```
$ ant -f test.xml
```

The script will convert, compile, run the recorded test and give a brief output on the success or failure of the test.

2.5. Setting Up a Grid Hub

The grid hub can be installed in the same server where the tests are run, in one of the grid nodes, or on a dedicated server. In a test development installation, you can install it on the same workstation or server with all the other components.

You can find the grid hub from the full installation package and from the `vaadin-testbench-grid` package.

2.5.1. Configuring the Hub

The hub is configured in the file `grid_configuration.yml`. You will need to edit this file if the predefined list of targets does not cover all the wanted targets, that is, operating system + browser combinations that you wish to test.
The first line contains tag "`hub:`", after which should come the port definition. A hub uses port 4444 by default. After the "`environments:`" tag comes a list of name-browser pairs that each define a target.

Parameter	Description
`name`	Corresponds to the Target value in a remote control in a node. The name can be defined as anything, but may not contain commas, which act as separator characters in the environment variable of the remote control configuration.
`browser`	Identifies a browser run by this target. The browser identifiers are prefixed with an asterisk and can be appended with an exact path to the browser executable. A list of allowed browser identifiers and a description of the executable paths is given in Section 2.5.3: *Browser Identifiers*.

For example:

```
hub:
    port: 4444
    remoteControlPollingIntervalInSeconds: 120
    sessionMaxIdleTimeInSeconds: 90
    newSessionMaxWaitTimeInSeconds: 5
    environments:
      - name:      "winxp-ie7"
        browser:   "*iexplore"
      - name:      "winxp-firefox3"
        browser:   "*firefox"
      - name:      "winxp-googlechrome4"
        browser:   "*googlechrome"

      - name:      "linux-firefox3"
        browser:   "*firefox /usr/lib/firefox/firefox-bin"

      - name:      "osx-firefox35"
        browser:   "*firefox"
      - name:      "osx-safari4"
        browser:   "*safari"
```

A list of the predefined target environments is given in Section 2.5.2: *Predefined Target Environments*.

thinking of U and I vaadin }>

2.5.2. Predefined Target Environments

The predefined targets are system-browser combinations mapped to a browser identifier. They are as follows:

Target Name	Browser
`winxp-ie6`	`*iexplore`
`winxp-ie7`	`*iexplore`
`winxp-ie8`	`*iexplore`
`winxp-firefox3`	`*firefox`
`winxp-firefox36`	`*firefox`
`winxp-safari4`	`*safari`
`winxp-safari5`	`*safariproxy`
`winxp-opera10`	`*opera`
`winxp-googlechrome6`	`*googlechrome`
`linux-firefox3`	`*firefox`
`linux-firefox36`	`*firefox`
`linux-opera10`	`*opera`
`osx-firefox35`	`*firefox`
`osx-safari4`	`*safari`
`osx-opera10`	`*opera`

2.5.3. Browser Identifiers

Vaadin TestBench supports the following browser identifiers:

Browser Identifier	Browser
`*firefox`	Mozilla Firefox 2 or 3 (prefers 2)
`*firefox2`	Mozilla Firefox 2.x
`*firefox3`	Mozilla Firefox 3.x
`*firefoxchrome`	Mozilla Firefox
`*chrome`	Mozilla Firefox (*not* Google Chrome!)
`*iexplore`	Internet Explorer
`*safari`	Apple Safari and other WebKit based browsers
`*safariproxy`	Apple Safari 5 requires this identifier
`*opera`	Opera
`*googlechrome`	Google Chrome

The above list does not include incompatible or otherwise irrelevant browsers.
Notice that the target `*firefox3` is defined so that, if both Firefox 2 and 3 are installed on the system, preference is given to the version 2. This may cause an unexpected mix-up, as one would expect Firefox 3 to be launched.

thinking of U and I **vaadin** }>

Browser Path

In cases where

➡ multiple browsers with the same name are installed, or

➡ the remote controller can not find the browser because it is not installed to the default location,

the browser identifier can be appended with the absolute path to the browser executable, separated by a space from the identifier. For example, `"*firefox /opt/firefox3.6/firefox-bin"`.

If you specify the absolute path to the browser, it must be same on all grid nodes (remote controls) that support the target environment. If such a browser is installed in a different location on different nodes, you need to have a separate target environment specified for each.

Firefox in Linux

Using the absolute path for Firefox is necessary in Linux, because the "`firefox`" program is just a script that launches the actual executable. Stopping the launch script does not close the browser, so the Firefox window will not close after the tests is done. If your Firefox installation directory is `/opt/firefox`, for example, you need to use `"*firefox /opt/firefox/firefox-bin"` to start Firefox.

The exact installation directory of Firefox depends on the system and used installation method (package management or manual installation). Typical locations include the following paths:

➡ `/opt/firefox/firefox-bin`

➡ `/usr/lib/firefox/firefox-bin`

➡ `/usr/local/firefox/firefox-bin`

➡ `/usr/local/lib/firefox/firefox-bin`

The binary itself can also be named something else that `firefox-bin`.

2.5.4. Starting the Hub

The grid hub is a service bound to a port, 4444 by default.

Windows

Navigate to `grid/hub/` in the installation directory and run `hub.bat` to start the hub on port 4444.

Closing the console window will stop the service.

Linux, Mac OS X, and other UNIX

1. Open a terminal window and change to `grid/hub/`

2. Run `hub.sh` to start the hub on port 4444.

```
$ cd vaadin-testbench-2.2.0/grid/hub
$ sh hub.sh
```

The hub service starts attached to the terminal and writes its log to standard output. Press **Ctrl+C** to stop the service. You can daemonize the service.

2.6. Setting Up a Grid Node

A grid node is a server, a desktop computer, or a virtual machine running a windowed operating system. It needs to have:

➜ Java 1.5 JRE (or newer) installed

➜ One or more web browsers installed

➜ **Vaadin TestBench Remote Control** installed and configured

A remote control is a service running on a grid node. It acts as a "remote control" for controlling the web browsers installed on the node: it can launch and stop browser applications and execute test commands in them. A remote control is itself "remote controlled" by the grid hub, which delegates the tests to the available remote controls.

The installation of web browsers is not covered in this manual.

2.6.1. Configuring the Remote Control

A remote control should be configured before starting it. The configuration is done by editing the `rc_configuration.xml` file. Values given in the XML configuration file override the default values defined in the run script.

Parameter	Description
`port`	The port this remote control listens to for commands from the hub. Default port is 5555.
`hubURL`	The address of the hub as a URL. Local host by default.
`environment`	An environment target (system-browser combination) that this remote control supports. The target must match one of the target names defined in the hub configuration. You can define multiple environments. See 2.5.1: *Configuring the Hub* for more information about the targets and a list of predefined targets.
`host`	The host name or IP address of this node controller. If the host name is not defined, the hub tries to determine it from the registration request.

If the hub is running on the same host as the remote control, only the environment targets need to be defined.

2.6.2. Configuring the Run Script

A remote control can also be configured in the run script `rc.sh` (or `rc.bat`) by defining:

Environment Variable	Description
`ENVIRONMENT`	Environment targets (system-browser combinations) supported by this remote control. The targets in the comma-separated list must match one of the target names defined in the hub configuration.
`USEREXTENSIONS`	Where `user-extensions.js` is located.

In this case, you will also need to add the following parameters after `SelfRegisteringRemoteControlLauncher`.

Parameter	Description
`hubUrl`	Address of the hub.
`port`	The port that this node controller should listen to.

On a windows machine -ensureCleanSession can be in the rc.bat to have the remote-control close all iexplore.exe running processes.

2.6.3. Starting the Remote Control

A remote control is a service bound to a port, 5555 by default.

Windows

Navigate to `grid/remote-control/` in the installation directory and run `rc.bat` to start the remote control.

Closing the console window will stop the service.

Linux, Mac OS X, and other UNIX

1. Open a terminal window and change to `grid/remote-control/`

2. Run `rc.sh` to start the remote control.

```
$ cd vaadin-testbench-2.2.0/grid/remote-control
$ sh rc.sh
```

The remote control service starts attached to the terminal and writes its log to standard output. Press **Ctrl+C** to stop the service. You can daemonize the service.

2.6.4. Running Tests Without a Grid Hub

You can run tests on a single testing node also without a grid hub, with just a remote control installed. The used remote-control needs to be a clean selenium remote-control that is started with the command:

```
java -jar selenium-server.jar -userExtensions user-
extensions.js
```

The user-extensions.js file can be found in the grid/remote-control/ folder.

When running the tests through JUnit, as described in Chapter 4: *Compiling and Executing JUnit Tests*, you need to:

1. Set the `com.vaadin.testbench.tester.host` property to point to the server on which the remote control runs instead of a hub host.
2. List the browsers installed in the RC host in the `browsers` property, either in the Ant script or in command-line. See the list of allowed browser identifiers in Section 2.5.1: *Configuring the Hub*.

```
<property name="browsers" value="*firefox"/>
```

You can also specify the browser executable path here, as described in the Section 2.5.1.

```
<property name="browsers"
         value="*firefox /opt/firefox/firefox-bin"/>
```

2.6.5. Browser settings

Turn off pop-up blockers for all browsers.

Internet Explorer

Make the settings in **Tools → Pop-up Blocker → Turn Off Pop-up Blocker**

Safari

Make the settings in **Edit → Block Pop-up Windows**

Also turn off default browser checks for all browsers.

2.6.6. 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.

Windows

Disable error reporting in case a browser crashes.

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

thinking of U and I **vaadin }>**

2.6.7. 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 the auto-hide function for the toolbar

→ Check that the toolbar is either locked or unlocked on all test hosts

→ Disable blinking cursor

→ Use the same screen resolution on all test machines and check that the maximized window is always the same size

→ Configure browsers in the same manner on all machines (same toolbars visible, same themes, etc)

→ Use identical operating system themeing on every host

→ Turn off any software that may suddenly pop up a new window

→ Turn off screen saver

Platform-Specific Settings

Windows / Internet Explorer:

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

3. Using Vaadin TestBench Recorder

3.1. Overview

Tests are recorded using the Vaadin TestBench Recorder. You can play back recoded test cases and use the Recorder to make assertions and take screenshots for screen capture comparison.
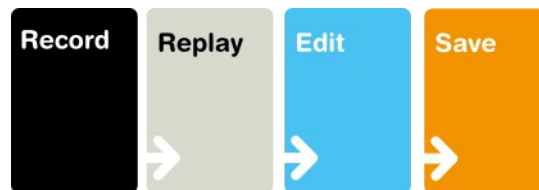
The Recorder is available only for Mozilla Firefox. To run the recorded tests in other browsers, you need to compile them as JUnit tests and run them with JUnit, as described in Chapter 4: *Compiling and Executing JUnit Tests*. It also allows automating the testing.

3.2. 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
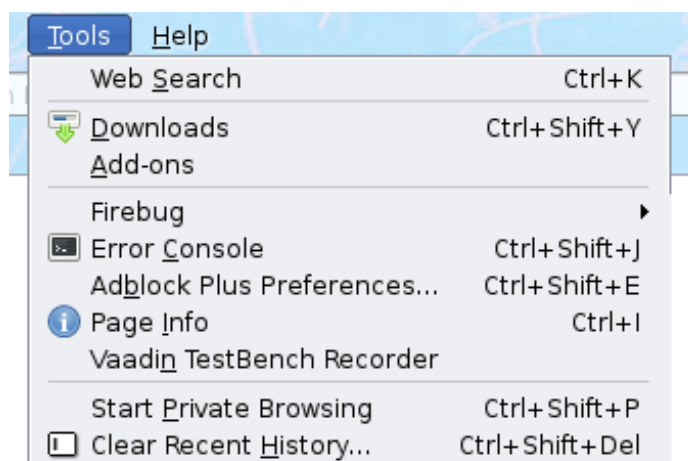
Figure 3: Starting Vaadin TestBench Recorder

The Vaadin TestBench Recorder window will open, as shown in Figure 4.

thinking of U and I **vaadin }>**
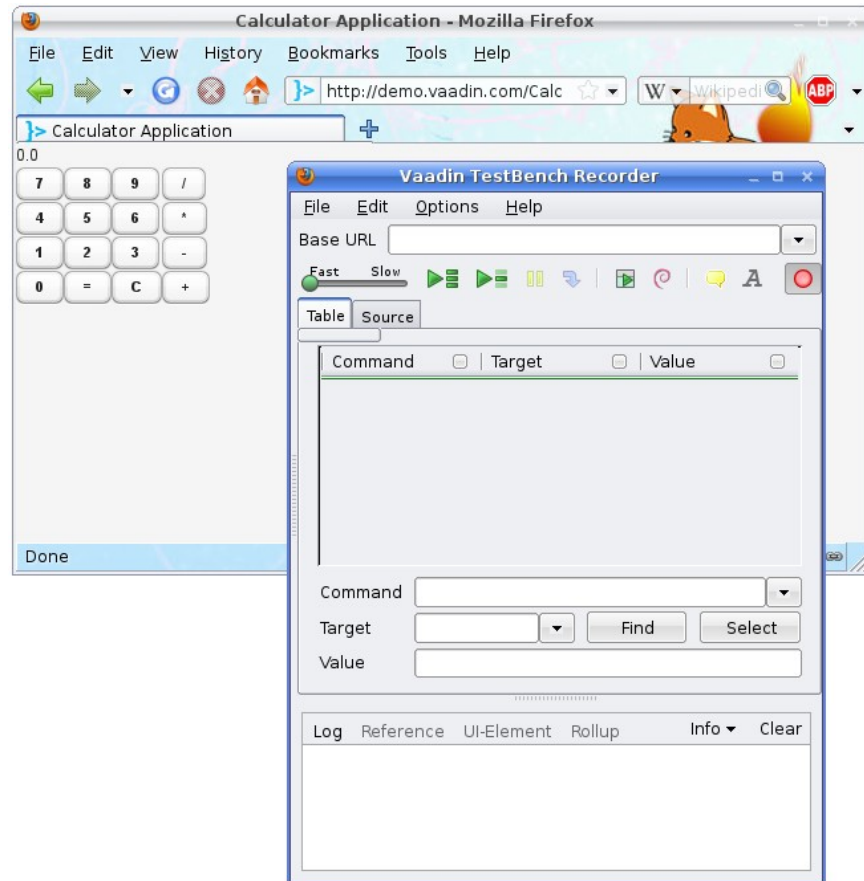
Figure 4: Vaadin TestBench Recorder running with the Calc demo

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

3.3. 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.
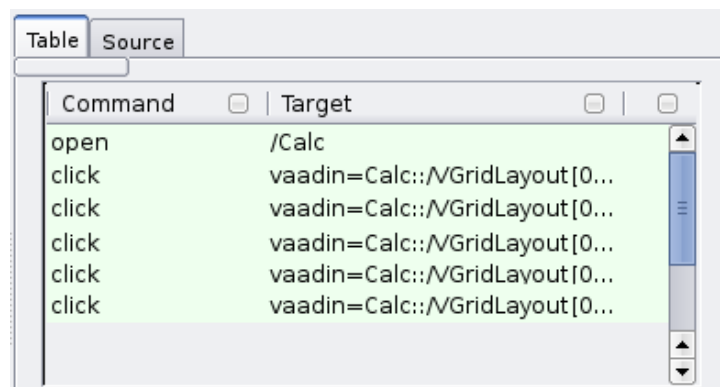
Figure 5: User interaction recorded as commands.

thinking of U and I vaadin }>

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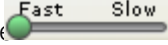
While recording, you can insert various commands such as assertions or take a screenshot by selecting the command from the **Command** list.

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

3.4. 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 `&restartApplication` parameter for an application in the URL to restart it.
You can also play back saved tests by opening a target 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.

3.5. Editing Tests

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 6 shows adding an assertion after calculating "6*7=" with the Calc demo.

thinking of U and I **vaadin }>**
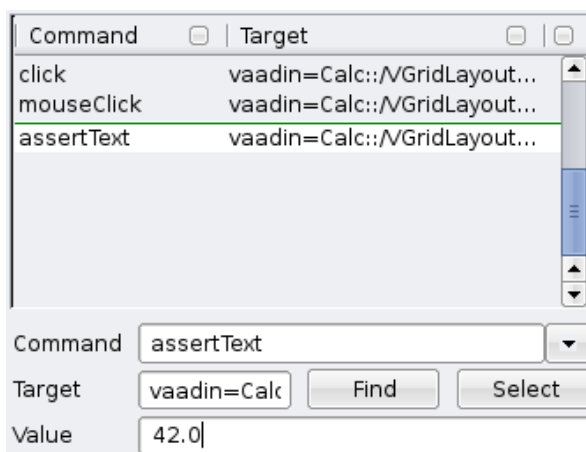
Figure 6: Inserting commands in a test script.

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 **Find**. 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.

3.6. Test Script Commands

Vaadin TestBench Recorder is based on the Selenium IDE, so the Selenium documentation provides a complete reference of all the commands available in the Selenium IDE and therefore in the Recorder.

Vaadin TestBench Recorder has the following special commands:

➔   `screenCapture`

➔   `showTooltip`

➔   `assertText`

➔   `assertCSSClass`

➔   `includeTest`

➔   `expectDialog`

➔   `uploadFile`

These special commands are described next.

thinking of U and I   **vaadin** }>

3.6.1. Comparing Screen Capture Images: `screenCapture`

The `screenCapture` command orders the Remote Control to take a screen capture of the browser view and compare the result against a reference image, if one is available. If a reference image is not available, the command will save the screen capture and fail the test. You can then later copy the captured image as the reference image.

Observe that the screenshots are *not* taken by the Recorder, for example when you are playing back a test case in the Recorder. They are taken by the Remote Control only afterward, when you run the tests from the test server that controls the Remote Control (through the hub).

The **Value** field can be used to define an identifier string for the screenshot. If the value is empty, a running number starting from 1 will be used as the identifier. Using a given identifier is more reliable than the numbers if the test is changed and screenshots are added or removed.

The naming convention for screenshot file names is automated and has the following format:

`NameOfTest_OperatingSystem_BrowserName_BrowserMajorNumber_Identifier.png`

Comparison and storage of screenshots is described in Section 4.6: *Comparing Screenshots*.

3.6.2. Recording ToolTips: `showTooltip`

Clicking switches Recorder to a tooltip mode that allows making a tooltip appear. Recording is done by hovering over the target element until a tooltip appears and then moving the mouse away from the element. This will insert a `showTooltip` command and disable the tooltip button.

After this command causes a tooltip to appear, its content can be asserted with other commands.

3.6.3. Recording AssertText: `assertText`

Clicking allows recording an `assertText` command on elements where the context menu (which appears by clicking right mouse button) does not work for some reason. When the `assertText` button is active, the Recorder will record an `assertText` command for the next mouse click, with the clicked element as the target.

3.6.4. Asserting element css class: `assertCSSClass`

Asserting a css class can be useful to determine if a row in a table is selected or other cases where the use of a normal assertion would be inconvenient. The `assertCSSClass` needs to be manually added and target selected.

thinking of U and I **vaadin }>**

Note that when selecting the target with the Select button, the selected
target may be a child to the wanted element. In this case the target
needs to be edited or re-selected so that it points to the correct
element.

3.6.5. Asserting element css class: `assertNotCSSClass`

The target `assertNotCSSClass` has the reverse function to
`assertCSSClass`. This one is useful if a class should have been removed
from an element.

3.6.6. Connecting tests together: `includeTest`

Vaadin TestBench allows connecting tests together. This is done by
inserting the `includeTest` command, where *Value* is the path to the
test to be inserted. The path can be relative to the directory from which
TestConverter is executed or an absolute path. Target test will be added
in full at the position with `includeTest`. Inclusion only works when
converting to JUnit tests with TestConverter.

3.6.7. Handling a confirmation box: `expectDialog`

The commands expectDialog + assertCommand are needed when a click event
opens up a Confirmation dialog.

3.6.8. Uploading files: `uploadFile`

Vaadin TestBench supports uploading of files for the Firefox browser. File uploads
are recorded automatically in the recorder and can be replayed as such. The
`uploadFile` command differs from default behavior in Selenium in the way that
there is no need to deploy the file to a URL that is accessible from all remote-
controls, but the file is passed along to the remote-controls, which then use the
local copy when running the test. The file can be placed in the same location as
your test scripts (see 3.7 Saving Tests) as long as the name is not changed. This
file is sent to the remote-controls if found, otherwise the file is read from the
absolute path given in the *Value* property of the command.

Vaadin TestBench is not aware of when the upload has been completed
so you need to add a command such as `waitForElementVisible` if
there is a component that is added after the upload, e.g. a Label
showing "Upload ok". Otherwise Vaadin TestBench will continue with the
next command before the upload has finished, which may cause
problems if it is not taken into account.

NOTE! Playback of the `uploadFile` command in the TestBench Recorder will
currently not succeed if using Vaadin locators in Firefox 3.x due to security
restrictions in the browser. This is a restriction in Recorder only and uploadFile in
TestBench RemoteControl works correctly also when using a Vaadin locator. In
order for replay to work in Recorder in Firefox 3.x, the *Target* should be
changed to a standard XPath selector string by choosing one from the
Target drop down menu. This problem has been fixed in the soon to be
released Firefox 4.

thinking of U and I **vaadin** }>

3.6.9. Shortcut keys: `keyPressSpecial`

For testing shortcut keys (Ctrl, Alt, Shift + key) the command keyPressSpecial can be used. Combinations using arrowkeys will be automatically recorded.

The value for keyPressSpecial should be of the form modifier keys split with space and ending in target key eg. 'ctrl a' 'alt shift a' or equivalent.'

Note that the alphabetical keys are dependent on capitalisation and that Vaadin `ShortcutAction.KeyCode` responds to uppercase letters.

3.7. Saving Tests

3.7.1. Saving Individual Tests

You can save a test by selecting **File → Save Test**. If you are just learning to use the Recorder, give `example/testscripts/` directory below the Vaadin TestBench installation directory as the target directory.

Vaadin TestBench stores the tests and test suites as HTML files. This makes it easy to review saved test scripts with a web browser and edit them manually.

3.7.2. Saving Test Suites

You can save multiple tests as a test suite with **File → Save Test Suite**. If you are just learning to use the Recorder, give `example/testscripts/` directory below the Vaadin TestBench installation directory as the target directory.

An entire test suite is executed as a single JUnit test. However, the success or failure is reported for the entire suite, which may be undesired. It is usually more useful to get the result for each test separately.

Test suites are nevertheless useful for composing larger tests from separate test phases. For example, you could have all your tests as test suites, and each would have a "login test" as the first phase, followed by some test case specific phases. You can also use the `includeTest` command for the same purpose

3.8. Invalid Tests

Tests can become invalid due to intentional changes to the application. Normally it involves changes in elements so that the Vaadin TestBench element locator can't find the correct element. (refer to known problems)

If the problem is that an element can't be found press ▶ to playback the test and find the problem position and check with the **Find** button that the element can't be found (eg. It's not a problem with timing that the element just isn't available yet). Re-selecting the element is simplest with the **Select** button that will update the target.

4. Compiling and Executing JUnit Tests

4.1. Overview

JUnit allows running tests remotely on a variety of different web browsers installed in grid nodes. The test scripts need to be first compiled from the HTML scripts saved with Recorder to Java classes. Vaadin TestBench Library includes a converter from the HTML format to Java source files, which you can then compile with a Java compiler.

The recommended way to compile and run JUnit tests is to use an Ant script, as described in Section 4.2: *Configuring the Ant Script*. The subsequent sections give more details regarding the tasks and their configuration options and describe how to use the tools from command-line.

A complete testing process involves the following tasks:

1. Start Vaadin TestBench Grid Hub

2. Start Vaadin TestBench Remote Control on each test node

3. Build the application to be tested

4. Convert and compile the tests

5. Deploy the application to be tested (start the server)

6. Remove old error screens for screenshot directory

7. Run the compiled JUnit tests

8. Collect test results

9. Clean up temporary files (source and compiled java files)

10. Undeploy the application (stop the server)

11. Analyze test results

The exact order of the tasks may vary, especially for the conversion and compilation of tests, which can be done also before the step 1 or somewhere between the steps 3 and 5. Also deployment and the collection of test results varies.

Building the application and how it is deployed is out of the context of this manual. Java applications are typically built with build systems such as Ant or Maven. Such a build system can be integrated with a continuous integration system that not only builds the application, but

thinking of U and I vaadin }>

also runs tests and collects test results. Support for JUnit tests in Vaadin TestBench allows integration with any continuous integration system that supports JUnit tests.

4.2. Configuring the Ant Script

The recommended way to compile and run JUnit tests is to use an Ant script. An example script is given in the `examples` directory in the installation package. It will convert, compile, and run the tests.

You can make the settings either by editing the Ant script or by giving the settings as property definitions from the command-line, for example:

```
$ ant -Dcom.vaadin.testbench.tester.host=localhost ...
```

4.2.1. Mandatory Settings

Define required system property values to the java virtual machine:

`com.vaadin.testbench.tester.host`

> Host name or IP address of the Vaadin TestBench Grid Hub. For example, "`localhost`". The port number should not be given in this parameter. If you have just a single test node and do not need the hub, you can give the address of the node.

`com.vaadin.testbench.deployment.url`

> Base URL of the Vaadin application to be tested, for example, "`http://demo.vaadin.com/`".

`com.vaadin.testbench.screenshot.directory`

> Base directory for screenshot reference and error images. The reference images are expected to be stored under the `reference` subdirectory. Error images are stored by JUnit under the `error` subdirectory. On the first run, there are no reference images; you should copy "accepted" screenshots from the `error` directory to the `reference` directory.

`browsers`

> A comma-separated list of target environments on which the tests should be run. The list entries must match targets defined in the configuration of remote controls in the grid nodes. They must also be listed in the configuration of the hub. The predefined browser names are system-browser pairs.

The default settings in the example test script are:

```
<!-- Host name or IP address of the host running  -->
<!-- TestBench RemoteControl or TestBench Hub.     -->
<property name="com.vaadin.testbench.tester.host"
          value="127.0.0.1" />

<!-- Base URL where the testable application is    -->
<!-- deployed                                       -->
```

```
<property name="com.vaadin.testbench.deployment.url"
        value="http://demo.vaadin.com/" />

<!-- Browsers to use for testing -->
<property name="browsers"
        value="winxp-ie8,winxp-firefox35" />

<!-- Base directory for screenshots.              -->
<property name="com.vaadin.testbench.screenshot.directory"
        value="screenshots" />
```

4.2.2. Optional Settings

Optional property values that can be used:

`com.vaadin.testbench.screenshot.softfail`

> If "`true`", a test is allowed to continue even if a screenshot comparison fails.
>
> If "`false`", the test is interrupted and testing will continue with the next test.

`com.vaadin.testbench.screenshot.onfail`

> If "`false`", the screenshot when a test fails is not taken.

`com.vaadin.testbench.screenshot.cursor`

> If "`true`", makes a check if error is because of a cursor.

`com.vaadin.testbench.screenshot.block.error`

> Sets the amount of difference that causes a screenshot comparison to fail. Comparison is done by 16×16 pixel blocks. The difference limit is given as a fraction ($0 < x \leq 1$) of how much a block can differ from the reference block. Default is 0.025, which means 2.5% difference.

`com.vaadin.testbench.screenshot.reference.debug`

> If "`true`", writes extra output for debugging purposes.

`com.vaadin.testbench.screenshot.max.retries`

> Define how many times a screenshot should be taken again if not the same as the reference image. (Default is 2 retries)

`com.vaadin.testbench.screenshot.resolution` (recomended)

> Specify the screenshot resolution as <width>x<height> (eg. 1024x768) Browser windows will be resized so that canvas has given size (doesn't work with Opera, which requires a custom profile file "profile-opera.txt" in the Remote Controller folder).
>
> Note. If tests give faulty croppings and sizes on screenshots, check that the browser canvas fits into the desktop.

thinking of U and I **vaadin** }>

4.3. Converting HTML Tests

The tests are stored in HTML format and need to be first converted to Java source files before they can be compiled as executable JUnit tests.

Converting tests is done by:

➜ Using the **TestConverter** in the Vaadin TestBench Library

4.3.1. Using the TestConverter

The **TestConverter** is a utility included in the Vaadin TestBench Library. You can use it from the command line or with the `create-tests` target in the example Ant script.

To use the converter, you need to add the Vaadin TestBench Library into your build path:

➜ `vaadin-testbench-2.2.0.jar`

The library is located in the root directory of the Vaadin TestBench installation package.

4.3.2. Command-Line Interface

You can use the test converter from command-line as follows:

```
$ java com.vaadin.testbench.util.TestConverter <OutputDir>
<Browsers> <HTMLTestFiles>...
```

You also need to add the `vaadin-testbench-2.2.0.jar` in the class path, either with the `-cp` command-line parameter or with the `CLASSPATH` environment variable.

The parameters are:

Parameter	Description
OutputDir	Output directory where the generated Java source files should be written. The source files are organized in Java packages by test name and target environment name.
Browsers	A comma-separated list of target environments. The target names must match the entries in the hub and remote control configurations.
HTMLTestFiles	A space-separated list of test files in HTML format.

For example, assuming that you have recorded and saved a test by name `mytests/CalcTest.html`, as described in Section 3.7: *Saving Tests*, you could give the following command in the installation directory:

```
$ java -cp vaadin-testbench-2.0.0.jar
  com.vaadin.testbench.util.TestConverter
```

thinking of U and I **vaadin }>**

```
   java/ linux-firefox36 mytests/CalcTest.html

Using output directory: src
Generating test CalcTest for linux-firefox36 in
CalcTest.linux_firefox36
Creating src/CalcTest/linux_firefox36/CalcTest.java for
CalcTest
```

Here, we assume that we have a remote control capable of executing tests for the `linux-firefox36` target. The command would create the Java source file `src/CalcTest/linux_firefox36/CalcTest.java`.

The Java package name is determined from the test name and the target environment.

4.3.3. Ant Script

The example Ant script located in the example/ directory in the installation package includes a `create-tests` target, which runs the test converter for each HTML test file.

The target is called automatically by the default target in the script, but you can run it separately as well. For example:

```
$ cd example
$ ant -f test.xml create-tests
Buildfile: test.xml

create-tests:
     [echo] Using test scripts: '/opt/vaadin-testbench-
2.0.0/example/testscripts/demo.vaadin.com.html'
     [java] Using output directory: temp-dir/src
     [java] Generating test demo_vaadin_com for winxp-ie8
in demo_vaadin_com.winxp_ie8
     [java] Creating temp-
dir/src/demo_vaadin_com/winxp_ie8/demo_vaadin_com.java for
demo_vaadin_com
     [java] Generating test demo_vaadin_com for winxp-
firefox35 in demo_vaadin_com.winxp_firefox35
     [java] Creating temp-
dir/src/demo_vaadin_com/winxp_firefox35/demo_vaadin_com.jav
a for demo_vaadin_com

BUILD SUCCESSFUL
Total time: 1 second
```

The example script writes the Java source files under `temp-dir/src` directory, as defined with the `temp-dir` property in the script.

4.3.4. Optional conversion settings

com.vaadin.testbench.converter.parameterFile

Define a .properties file to be used for parametrising TestBench test Values. Should be given as a system property (-Dcom.vaadin....parameterFile=VALUE in command-line or as

thinking of U and I vaadin }>

<sysproperty key="com.vaadin...parameterFile" value="VALUE" /> in ant script).

4.4. Compiling JUnit tests

The Java source files for the JUnit tests need to be compiled. You need include the Vaadin TestBench Library `vaadin-testbench-2.2.0.jar` in your class path. The library is needed also later when running the tests.

4.4.1. Command-Line Compilation

You can compile the tests with a Java compiler from the command-line, for example as follows:

```
$ javac -cp vaadin-testbench-2.0.0.jar -d classes
src/CalcTest/linux_firefox36/CalcTest.java
```

In the above example, we assume that the source file is located in the `src/` directory, where it was written in the example in Section 4.3.2 earlier. The compiled class files are written to `classes/`.

4.4.2. Ant Build Script

The example Ant script located in the `example/` directory in the installation package includes a `compile-tests` target, which compiles all the JUnit test Java sources.

For example:

```
$ ant -f test.xml compile-tests
Buildfile: test.xml

create-tests:
    [echo] Using test scripts: '/opt/vaadin-testbench-
2.0.0/example/testscripts/demo.vaadin.com.html'
    [java] Using output directory: temp-dir/src
    [java] Generating test demo_vaadin_com for winxp-ie8
in demo_vaadin_com.winxp_ie8
    [java] Creating temp-dir/src/demo_vaadin_com/winxp_
ie8/demo_vaadin_com.java for demo_vaadin_com
    [java] Generating test demo_vaadin_com for winxp-
firefox35 in demo_vaadin_com.winxp_firefox35
    [java] Creating temp-dir/src/demo_vaadin_com/winxp_
firefox35/demo_vaadin_com.java for demo_vaadin_com

compile-tests:
    [mkdir] Created dir: /opt/vaadin-testbench-
2.0.0/example/temp-dir/classes
    [javac] Compiling 2 source files to /opt/vaadin-
testbench-2.0.0/example/temp-dir/classes

BUILD SUCCESSFUL
Total time: 2 seconds
```

thinking of U and I **vaadin }>**

The example script writes the compiled classes under `temp-dir/classes` directory, as defined with the `temp-dir` property in the script.

4.5. Executing JUnit Tests

Execution of JUnit tests requires:

➜   All tests have been compiled as Java classes (Section 4.4)

➜   Vaadin TestBench Grid Hub is running (Section 2.5)

➜   One or more Vaadin TestBench Remote Controls is running (Section 2.6)

➜   Vaadin TestBench Library

JUnit and Selenium Remote Control Java Client are included in the Vaadin TestBench Library, `vaadin-testbench-2.2.0.jar`.

4.5.1. Test Execution Parameters

The following parameters must be defined:

`com.vaadin.testbench.tester.host`

Host name or IP address of the Vaadin TestBench Hub. For example, "`localhost`", as you would have in a test development setup. The address can also be the address of a Remote Control in a single test node installation described in Section 2.6.4: *Running Tests Without a Grid Hub*.

`com.vaadin.testbench.deployment.url`

The URL address of the Vaadin application to be tested. For example, " http://demo.vaadin.com/sampler/".

`com.vaadin.testbench.screenshot.directory`

Directory where screenshots will be stored, assuming that any are taken. The directory path can be relative to the directory where JUnit is executed or absolute.

4.5.2. Executing JUnit from Command-Line

Execution of JUnit requires the JUnit libraries, which are included in the Vaadin TestBench Library, and the compiled tests.

In the following example, we assume that we have converted and compiled a test case for the Calc application and we have a remote control running and capable of testing the `linux-firefox36` target.

```
$ java -cp vaadin-testbench-2.0.0.jar:classes
    -Dcom.vaadin.testbench.tester.host=localhost
    -Dcom.vaadin.testbench.deployment.url=http://demo.vaadi
n.com/Calc/
    org.junit.runner.JUnitCore
    CalcTest.linux_firefox36.CalcTest
```

thinking of U and I **vaadin** }>

```
JUnit version 4.5
Time: 17.964
There was 1 failure:
1) testlinux_firefox36(CalcTest.linux_firefox36.CalcTest)
junit.framework.AssertionFailedError: Test was missing
reference images.
        at junit.framework.Assert.fail(Assert.java:47)
        ...
FAILURES!!!
Tests run: 1,  Failures: 1
```

In this example, the test failed because we did not have the reference images installed. See Section 4.6: *Comparing Screenshots* for instructions on copying the reference images after the first run.

4.5.3. Executing JUnit from an Ant Script

Apache Ant has an optional `<junit>` task, which makes it easier to run JUnit tests from Ant.

You need to:

➔ Include Vaadin TestBench Library and the compiled tests in the class path

➔ Define the required parameters with `<jvmarg>` elements

➔ List the *source files* of the tests in a `<batchtest>` element

See the `run-tests` target in `example/test.xml` for an example of using the task.

```xml
<junit fork="yes">
    <classpath>
        <fileset dir=".."
            includes="vaadin-testbench-*.jar" />
        <pathelement path="${temp-dir}/classes" />
    </classpath>

    <!-- Optional -->
    <formatter type="brief" usefile="false" />

    <!-- The required parameters -->
    <jvmarg value="-Dcom.vaadin.testbench.tester.host=$
{com.vaadin.testbench.tester.host}" />
    <jvmarg value="-Dcom.vaadin.testbench.deployment.url=$
{com.vaadin.testbench.deployment.url}" />
    <jvmarg value="-
Dcom.vaadin.testbench.screenshot.directory=$
{com.vaadin.testbench.screenshot.directory}" />

    <batchtest>
        <fileset dir="${temp-dir}/src">
            <include name="**/**.java" />
        </fileset>
    </batchtest>
```

```
</junit>
```

4.5.4. JUnit Output

The output of JUnit depends on how it was executed. When executed from command-line, it writes output to standard output. Ant provides possibilities to format the output and to write it to a file. Certain other build systems and continuous integration systems integrate with JUnit and support collecting the results.

Screenshots with errors are written to the `errors/` subdirectory under the screenshot directory given as a parameter. See 4.6: *Comparing Screenshots* for more details.

4.6. Comparing Screenshots

Vaadin TestBench allows taking screenshots of the web browser, as described in Section 3.6.1: *Comparing Screen Capture Images: screenCapture*.

When the tests are executed with JUnit, the captured images are compared to reference images. If the images differ more than the allowed amount, as defined by the block error parameter (see Section 4.2.2: *Optional Settings*), the test produces an **AssertionFailedError** exception. The comparison results in the same error also when the reference image is missing altogether.

4.6.1. Screenshot Comparison Error Images

Screenshots with errors are written to the `errors/` subdirectory under the screenshot directory given as a parameter for JUnit.

For example, the error caused by a missing reference image in the example in Section 4.5.2: *Executing JUnit from Command-Line* is written to `screenshot/errors/CalcTest_Linux_Firefox_3_CalcPicture.png`. The image is shown in Figure 7.

Figure 7: A screenshot taken by a test run

When taking screenshots, the browser is maximized to full screen. Only the page view area in the browser is captured.

Known issues:

➔   Using dual screen can cause trouble.

4.6.2.   Reference Images

Reference images are expected to be found in the `reference/` subdirectory under the screenshot directory given as a parameter for JUnit. To create a reference image, just copy a screenshot from the `errors/` directory to the `reference/` directory.

For example:

```
cp
  screenshot/errors/CalcTest_Linux_Firefox_3_CalcPicture.png
  screenshot/reference/
```

Now when the proper reference image exists, rerunning the test in Section 4.5.2: *Executing JUnit from Command-Line* 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, e.g.

```
CalcTest_Linux_Firefox_3_CalcPicture.png
CalcTest_Linux_Firefox_3_CalcPicture_1.png
CalcTest_Linux_Firefox_3_CalcPicture_2.png
```

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

4.6.3.   Visualization of Differences in Screenshots with Highlighting

Vaadin TestBench supports advanced difference calculation of 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/` directory as the screenshots from the failed tests.

The differences in the images are highlighted with blue squares. 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.

thinking of U and I **vaadin }>**

Figure 8 shows a difference report with three differences. Date fields are a typical cause of differences in screenshots.

Figure 8: A highlighed error image and the reference image

4.6.4. 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.

thinking of U and I vaadin }>