Magento Test Automation Framework User's Guide (v 2)

Table of Contents

Purpose and Scope

Purpose

This document is intended for specialists who are either interested in or/and involved into using Magento Test Automation Framework (hereinafter named as MTAF). The framework is used to run automated test against a thoroughly installed Magento application. Thus, it possible to say that the document is basically aimed on Magento Quality Assurance specialists and PHP developers.

Note
Be aware that using MTAF requires advanced knowledge of XML, YAML, and PHP skills in installing the necessary working environment, creating functional tests, executing the tests created, and using test results properly.

The current document also explains all the mentioned requirements as well as milestones of using MTAF.

Scope

This section presents a brief overview of the document's content.

- Conventions - this chapter describes style conventions used in the document.
- Glossary - this chapter explains all the specific terms used in the current document.
- MTAF Environment Requirements - this chapter lists the software, required to use MTAF.
- Supported Web Browsers - this chapter presents the list of Web browsers supporting Magento, and thus MTAF.
- Introduction to Magento Automated testing - this chapter describes benefits and workflow of automated testing. It also provides the list of tools and frameworks used to maintain automated testing.
- MTAF File Formats - this chapter provides the list of file formats.
- MTAF Logical Structure - this chapter provides a graphical scheme of MTAF logics.
- MTAF Physical Structure - this chapter provides an overview of files used for Magento Test Automation Framework - their physical location and content.
- MTAF Configurations - this chapter provides configurations to Selenium server and Selenium client.

- Running Tests - this chapter provides description of ways of running tests.
- Results of Running Tests - this chapter provides examples of test results display.
- MTAF Instructions - this chapter provides instructions on custom UIMap, DataSet, test script and test suite creation.
- Appendix A - this appendix provides instructions on installing add-ons for Firefox browser.

Conventions

- **Bold style** - is used to highlight paths to the files (as well as for file names) mentioned in the document.
- `Code style` - is used to highlight samples of code used in the document.

Glossary

Term	Description
Magento Test Automation Framework (MTAF)	A software package used for running repeatable functional tests against a normally installed Magento application. MTAF is used for both: performing the actual testing and writing test automation scripts. Test automation scripts, created within the framework, can be used for testing most of the Magento functionality. This is a cross-platform solution (in other words it does not have any dependency on operating systems). MTAF allows QA specialists to quickly develop all kinds of tests for the current Magento version, and thus the tests can be reused at any time. Framework users can always run a single test separately, or bunch of tests (test suite), or all tests.
Test Automation Script	This is a PHP class, dependent on the PHPUnit framework and the Selenium library. The script initiates running a specific test case, or a suite of test cases.
DataSet	A data file or a set of data files, required for running automated tests. Such file(s) are created in the YAML format. For more information refer to the MTAF Logical Structure section
UI Map	The concept of using UI Map is to define, store, and serve the UI elements of an application or a website. In case of Magento (website), the definition of UI elements depends on the Selenium technology, which uses strings, and contains XPATH in order to locate/define UI elements. In other words, UI Map is a repository of test script objects, which correspond to UI elements of the application being tested.
User	Quality Assurance Engineer, PHP Developer, or any other person with similar skills and responsibilities.
Document Object Model (DOM)	This is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents. Aspects of the DOM (such as its "Elements") may be addressed and manipulated within the syntax of the programming language being in use. The public interface of a DOM is specified in its application programming interface (API).

MTAF Environment Requirements

Prior to start using Magento Test Automation Framework, make sure that you have the following software installed:

- PHP 5.2.0 or later
- PHPUnit 3.5.13 or later
- Java Run-time Environment (JRE) 1.6 or later
- Integration Development Environment - it is recommended to use NetBeans 6.9.1 or later. Alternatively it is possible to use other frameworks such as Zend, Eclipse, and etc.
- Selenium Remote Control (RC) 1.0.3 or later, 2.0 rc2
- TortoiseSVN (it is recommended, but optional)
- Magento Community Edition 1.5 or later

Note
In addition, make sure that the following requirements are met:
- Selenium RC server must be able to locate the browser's '.EXE' file in order to successfully run a browser. For more information about this, refer to the Selenium official website http://seleniumhq.org/docs/.
- A custom browser profile must be created to be used by Selenium for running automated test. For more information about this, refer to the Selenium official website http://seleniumhq.org.

Supported Web Browsers

- Mozilla Firefox 3.x or later
- Google Chrome
- Internet Explorer 6.0 and later

- Safari 5 or later

Introduction to Magento Automated Testing

Testing (also known as “quality assurance”) is an essential part of a software development process. While testing intermediate versions of products being developed, Magento quality assurance team (hereinafter “QA team”) needs to execute a number of tests. In addition, prior to publishing every new version of Magento platform, it is mandatory that the version is passed through a set of “regression” and “smoke” tests. Most of all such tests are standard for every new version of Magento products, and therefore can be automated in order to save human resources and time for executing them.

Benefits of using automated testing are the following:

- Simplified testing procedures, which use already existing automated tests
- Reduction of tests’ time execution and human resources required
- Complete control over the tests’ results (“actual results” vs “expected results”)
- Possibility to quickly change test’s preconditions and input data, and thus re-run the tests dynamically

At the moment, there are two major approaches used for automated testing: Unit Testing (hereinafter “UT”), and Automated Functional Testing (hereinafter “AFT”).
Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently examined for proper operation. In procedural programming a unit may be an individual function or procedure. In object-oriented programming a unit is usually an interface, such as a class. In Magento, the UT approaches is almost not used, because it requires significant time and resources to establish dependencies between different modules.
AFT, in terms of Magento, means an automated process used to diagnose whether the application's functionality meets all the functional requirements or not. Using the AFT approach, first of all guarantees that specific code (function/feature) accomplishes corresponding tasks successfully. Secondly, running functional tests confirms that application operates exactly the way it is expected.

The workflow of using AFT in Magento applications can be presented as follows:

1. First of all it is required to identify tasks that an application has to accomplish.
2. Second, a set of necessary input data has to be created.
3. Third, expected results have to be defined in order one can judge that an application (a requested feature) works correspondingly.
4. Fourth, a QA personnel executes a test.
5. Finally, a QA personnel compares expected results with actual results, and decides whether the test has been passed successfully.

Magento Test Automation Framework (hereinafter “MTAF”) uses the following tools and frameworks:

- **PHPUnit**. PHPUnit is a unit testing framework for the PHP programming language. Its purpose is to find mistakes in PHP source code. The major benefit of using PHPUnit is a possibility to establish testing infrastructure, and to re-use it as much as needed, by only creating unique parts for every particular test.
- **Selenium**. Selenium is a well know open source testing framework, which is widely used for testing Web-based applications. It contains a set of different software tools each with a different approach to support test automation. These tools provide a rich set of testing functions specifically geared to the needs of web application testing of all types. Such operations are highly flexible, providing many options for locating UI elements and comparing expected test results against actual application behavior. One of the Selenium’s key features is the support of executing tests on multiple browser platforms. Selenium uses simple, but at the same time powerful Document Object Model (DOM). It also allows exporting tests to several programming languages and frameworks.
  The framework includes the following tools:
  - **Selenium Integrated Development Environment (IDE)** – is a prototyping tool for building test scripts. In fact, for the time being, it is a Firefox plugin that provides an “easy-to-use interface” for developing automated tests. Selenium IDE has a recording feature, which records user actions as they are performed and then exports them as a reusable script in one of many programming languages that can be later executed. It contains a context menu that allows you to first select a UI element from the browser’s currently displayed page and then select from a list of Selenium commands with parameters pre-defined according to the context of the selected UI element. In fact, one can record and/or playback automated tests without a necessity to know test scripting language. This is not only a time-saver, but also an excellent way of learning Selenium script syntax.
  - **Selenium Remote Control (RC)** – is a test tool that allows you to write automated web application UI tests in any programming language against any HTTP website using any mainstream JavaScript-enabled browser.
  - **Selenium Web Driver** – is a tool for writing automated tests of websites. It aims to mimic the behavior of a real user, and as such interacts with the HTML of the application.
  - **Selenium-Grid** – is a Selenium derivative that allows the developers and or QA personnel to access Selenium Remote Controls without worrying about where the Selenium Remote Controls are. As a developer and/or QA, one can create a script as it would be for Selenium Remote Control and run it as normal.

MTAF File Formats

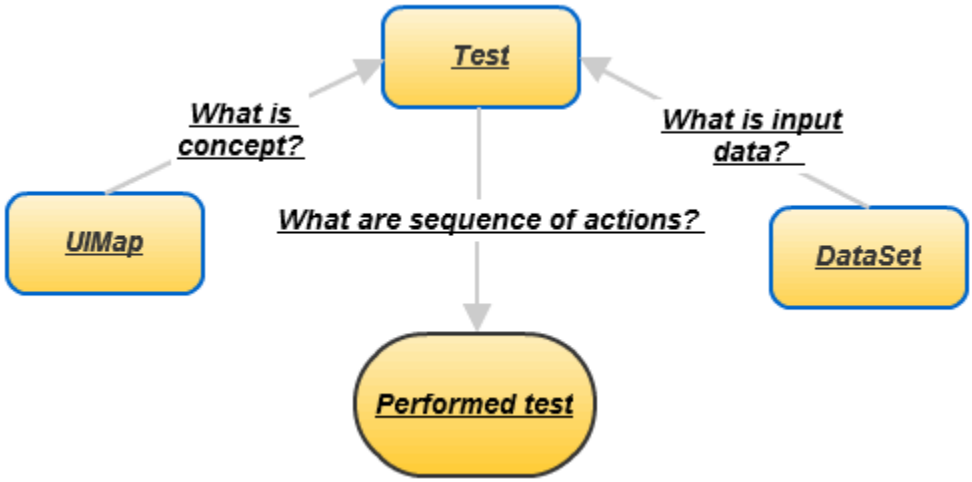
There are three file formats used for the MTAF files:

- XML – there is only one meaningful XML file used in MTAF - **phpunit.xml**. This is simply a configuration file.
- PHP – in MTAF, PHP files are basically used in two situations. First of all, PHP files are test cases files, located in the "Tests" directory (for more information refer to the MTAF Physical Structure chapter). Test cases can be run depending on configuration made in the **phpunit.xml** file, and use the input data set specified in the DataSet YAML files. Second, PHP files are also used as a framework library files.

- YAML – This acronym stands for "YAML Ain't Markup Language". Originally it meant "Yet Another Markup Language", but was changed to emphasize its purpose as data-oriented, rather than document markup. At the moment, YAML is the most popular human-readable data serialization language. Its major advantages are: simplicity, intelligibility, flexibility, expression and extensibility. It is easily implemented and uses data structure, which is native to both: programming languages and the whole data model. Most MTAF files are created in the YAML format. Those can present either UImap files or data set files.

MTAF Logical Structure

The following illustration explains basics of MTAF logics.



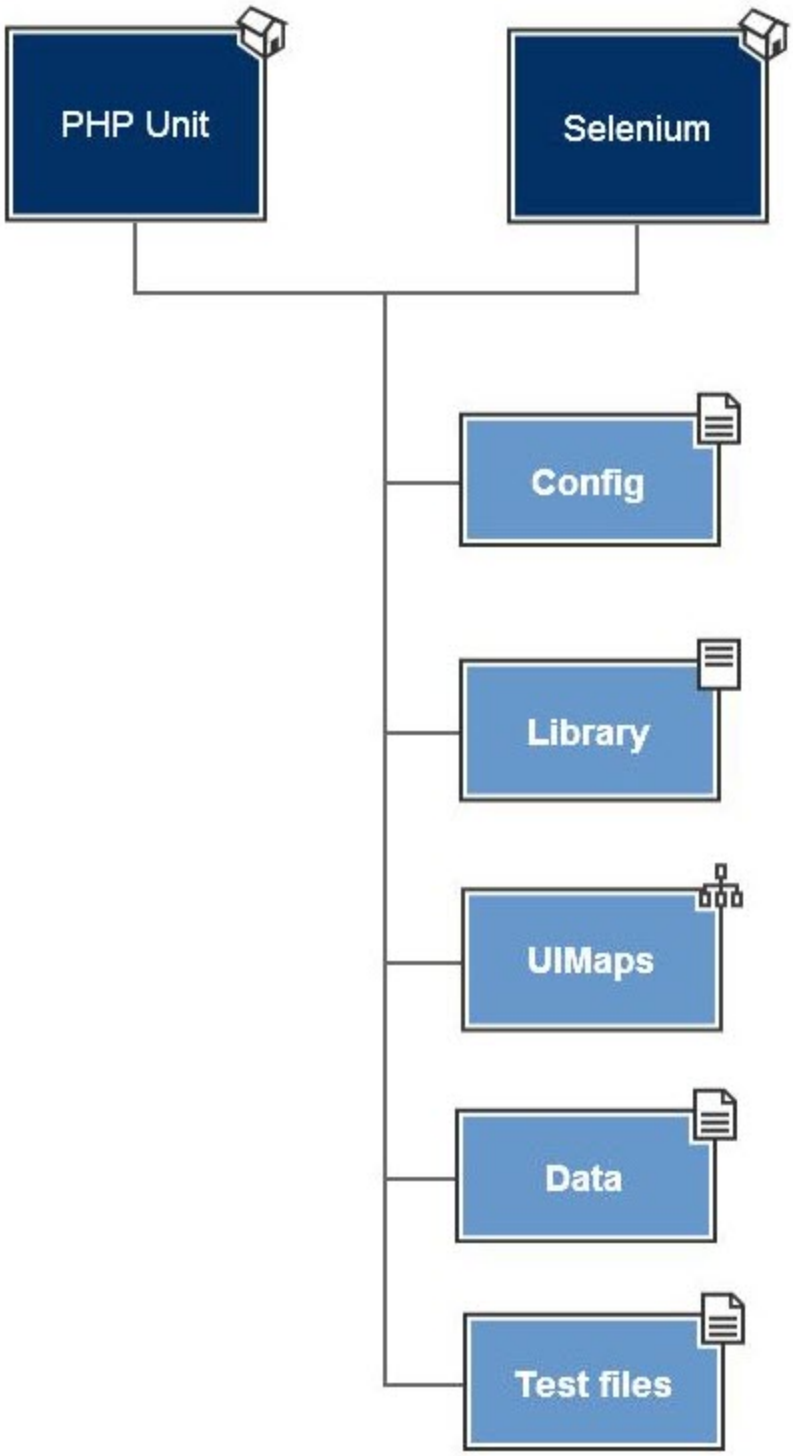
UIMap. UIMap is a concept for defining, storing, and serving UI elements of an application or a website. The UIMap file (YAML file) contains a set of ‘key-value’ pairs, where key is an alias of the UI element, and a value is the Xpath locator. In terms of UIMap, every page rendered by a browser is presented as a Document Object Model (DOM). Such model includes every single UI element that can exist on a page. Xpath locators, in fact, are paths to such UI element in a DOM model. Thus, using Xpathes, it is possible to link any UI element on a page (even ‘hidden ones’) with a method, that imitates an action being executed over it (a method in the MTAF library). For example, one can use UIMaps (Xpathes) and a “click” method from the MTAF library to reproduce clicking any UI element on the necessary Web page. An example of using such approach can be re-testing of the localized application.

DataSet. In terms of MTAF, DataSet is a pool of YAML files (data storage). These files describe data main-space, which 'test case' files can refer to. In other words, for search convenience purposes, that main-space is 'logically' splitted into several data files. Every data file can be 'related' to a specific functionality being tested (and thus, related to specific test case files).

Test. Generally speaking any test is considered as a single action or a sequence of actions, that defines whether a specific feature meets functional requirements. Digging down to the bottom of this term, it is possible to say that a test is a method in a class, inherited from the basic test case class. It calls a library function (or functions), which, in its turn, executes actions necessary to check the expected application's business logics.

MTAF Physical Structure

The following figure explains physical location of files required for Magento Test Automation Framework



As you can see from the picture above, all the Magento Test Automation Framework files are located in five main folders: **config**, **library**, **uimaps** , **data** and **tests** folders.

- **config** folder contains two major files: **browsers.yml** and **local.yml**. The **browsers.yml** file, in fact, is used as a template for configuring the Selenium client. For more information on using these files, refer to the QAA:Configuring Magento Test automation Framework section.
- **library** folder contains the whole class hierarchy from API framework.
- **uimaps** folder stores YAML files with the description of UIMaps for tests. It contains all UIMaps that are used in tests and can include some custom ones.
- **data** folder contains all of the input parameters for all tests. Test data can be loaded to this folder with the help of `loadData()` method. The example of loading data to the folder is provided below:

Example of Loading Data to data Folder

```
$storeData = $this->loadData('generic_store', Null, 'store_name');
```

- **tests** folder contains PHP files with test suites classes.

MTAF Configurations

Prior to start using the MTAF, you need to properly configure Selenium Server and Selenium Client. To configure the necessary Selenium Server settings, simply run a proper set of commands in the command line. For example, it can be similar to the following:

Example of Selenium Server Settings

```
java -jar selenium-server-standalone-2.4.0.jar -trustAllSSLCertificate
```

Where `trustAllSSLCertificate` stands for overriding all https warnings.

For more information on configuring Selenium Server, refer to
http://release.seleniumhq.org/selenium-remote-control/0.9.2/doc/server/org/openqa/selenium/server/SeleniumServer.html

To configure Selenium Client, you should use either the **browsers.yml** file or the **local.yml** file. Both files are located **Config** sub-folder of your local MTAF installation folder.

Warning
Be aware that the **browsers.yml** file (the original configuration template) is always under the SVN version control. This means, that if you upload a new MTAF installation from the SVN repository, all the settings you have previously done with this file will be overridden. On the other hand, if you commit changes made in the **browsers.yml** file to the SVN repository, it may later on effect other MTAF users. Therefore, it is strongly recommended to make your local configuration changes ONLY within the **local.yml** file, which is similar to the **browsers.yml** one, and (if exists), is used instead. Moreover, this file is not controlled by SVN, so you can always perform your own configuration settings without effecting original configuration template.

The following includes a typical example of the "default" **browsers.yml** file and description of its content.

Example of browsers.yml File

```
// List of browsers which can be used
browsers:
  // Default browser settings
    // Browser name
    chrome : '&chrome'
        browser: '*chrome'
        // Host where it is installed and will run
        host: 127.0.0.1
        // Port on the Host where this browser will be available
        port: 4444
        // Parameter which sets up the frequency of browser running:
        // false - Browser starts every time when new tests are run and shuts down in the end of each
test (recommended).
        // true - Browser starts ONLY before the first test and shuts down after the last one.
        doNotKillBrowsers: false
        // Parameter sets up the count of tests which must be run
        // sequentially before the browser will forcibly shut down.
        browserTimeoutPeriod: 30000

    firefox: '&firefox'
        browser: '*firefox'
        host: 127.0.0.1
        port: 4444
        doNotKillBrowsers: false
        browserTimeoutPeriod: 30000

    iexplorer: '&iexplorer'
        browser: '*iexplorer'
        host: 127.0.0.1
        port: 4444
        doNotKillBrowsers: false
        browserTimeoutPeriod: 30000

    googlechrome: '&googlechrome'
        browser: '*googlechrome'
        host: 127.0.0.1
        port: 4444
        doNotKillBrowsers: false
        browserTimeoutPeriod: 30000
    default: *chrome

// List of applications which can be tested
applications:

  // Settings of Magento application
  magento: &magento

    // General link to the frontend
    frontendUrl : 'http://www.localhost.com/magento/'

    // General link to the backend
    adminUrl    : 'http://www.localhost.com/magento/admin/'

    // Login to the backend
    adminLogin  : admin

    // Password to the backend
    adminPassword : admin

  //Is used for installation
  basePath: /local/path/to/app/installation

  // Pointer to the default application setting.
  // Used when more than one application is described
  // and one of which is required to be used by default
  default: *magento
```

- `browsers` - lists and configures browsers that can be used to run both Magento client and admin branches.
    - `browser: '*chrome'` - the browser's identification that Selenium RC accepts. Keep in mind that operating system is set up for a specific 'default' browser, not Magento.
    - `host` - is an identification of the machine where Selenium RC is installed.
    - `port` - is basically used in situations when one needs to support several servers on a single host, and thus needs to use different ports.
    - `default` - identifies the default browser to be used for running a test or a suite of tests. In fact this 'browser identification' option is used only to specify the required browser for a single test run. Thus, you do not need to specify other browsers, but simply classify the necessary one.

In fact, as you can see from the above example, the **browser.yml** file presents description of browsers supported, and description of the Magento application itself - its location and installation paths.

Tip
Selenium RC can work on remote computers, which means that, in fact, it is not not actually important where the real tests are located, where the Magento version being tested is installed, and finally where the testing environment exists. For example, it may happen that the three mentioned instances are located on three different virtual machines. Still, if you properly configure settings such as `frontendUrl`, `adminUrl`, `adminLogin`, `adminPassword` and `browsers`, the MTAF engine will work properly.

- **phpunit.xml** which contains the list of test scripts (test cases) to be run.

phpunit.xml – is the standard PHPUnit configuration file. The file forms a suite (the list of test scripts/test cases) to be run. In other words, it contains the suite's name, list of suffixes, and list of directories that the necessary PHP files will be run from. For example:

Example of phpunit.xml File

```
<phpunit
    bootstrap="bootstrap.php"
    colors="false"
    convertErrorsToExceptions="true"
    convertNoticesToExceptions="true"
    convertWarningsToExceptions="true"
    stopOnFailure="false"
    syntaxCheck="false"
    verbose="true"
    strict="false"
    printsummary="true">
    <testsuites>
        <testsuite name="All Tests">
            <directory suffix="Test.php">tests</directory>
        </testsuite>
    </testsuites>
    <logging>
        <log type="coverage-html" target="./tmp/report" charset="UTF-8" yui="true" highlight="false"
         lowUpperBound="35" highLowerBound="70"/>
        <log type="coverage-xml" target="./tmp/coverage.xml"/>
        <log type="graphviz" target="./tmp/logfile.dot"/>
        <log type="json" target="./tmp/logfile.json"/>
        <log type="metrics-xml" target="./tmp/metrics.xml"/>
        <log type="plain" target="./tmp/logfile.txt"/>
        <log type="pmd-xml" target="./tmp/pmd.xml" cpdMinLines="5" cpdMinMatches="70"/>
        <log type="tap" target="./tmp/logfile.tap" logIncompleteSkipped="true"/>
        <log type="junit" target="./tmp/logfile.xml" logIncompleteSkipped="false"/>
        <log type="testdox-html" target="./tmp/testdox.html"/>
        <log type="testdox-text" target="./tmp/testdox.txt"/>
    </logging>
</phpunit>
```

Where:

- `<testsuite name="All Tests">` - name of the suite to be run
- `suffix="Test.php` – suffix
- `tests` – name of the directory, where tests will run from

Taking into account these particular settings of the configuration file, the result will be - the suite to be run contains all PHP files (tests) from the **tests** directory, which have the `Test.php` suffix. In case there are no files with the specified suffix, only the mentioned file will be run as a test.

Warning
All PHP files (test cases) to be executed, have to be named using a 'CamelCase' style (each element's initial letter capitalized within the compound and the first letter either upper or lower case). This means that if you have PHP files named, for example, as **File1Test.php**, or "Test1Test.php", those will be run within a test suite. However, if you name a test case as, for example, "test222.php", it will not be a part of the sample suite, because it does not meet neither suffix, nor style rules.

Running Tests

To find an appropriate way to run tests you need to consider the number of tests (whether it is a test suite, a single test or all tests), test sites (local, remote, distributed machines or cluster) and continuous integration practices. You also need to take into account testing purposes when choosing a way to run your tests, for example test script debugging, manual start or continuous integration. According to that you may choose to run your tests from Integrated Development Environment (IDE), command prompt or from continuous integration server.

Integrated Development Environment (IDE)

Integrated Development Environment (IDE) provides you with the great number of various tools and plugins, which help you to write, debug and run your tests. IDE fits the most for running all of your tests or specific test suites, rather than single tests. It is the simplest and the most precise way of working with tests, you may easily write a test script and debug it in IDE, but it does not provide you with the full control over the tests and continuous integration.

Command Prompt

Command prompt provides you with the full control over your tests and independence from platforms. You are able to start your tests manually. This option is most applicable when you already have debugged tests, therefore they can be run on production.
To be able to work from a command prompt you need to have a BAT file with the **phpunit.xml** configuration. The same logic is used for running tests from continuous integration server, which we will discuss later.
You may configure your **phpunit.xml** file to run all tests, a specific test suite or just a single test:

Example of All Tests Configuration

```
<testsuite name="All Tests">
      <directory suffix=".php">tests</directory>
</testsuite>
```

Example of Test Suite Configuration

```
<testsuite name="Test Suite">
      <directory suffix="Test.php">tests/Customer</directory>
      <directory suffix="Test.php">tests/Store</directory>
</testsuite>
```

Example of Single Test Configuration

```
<testsuite name="Single Test">
     <directory suffix="test_file _full_name">file_path_from_test_category</directory>
</testsuite>
```

In this way running tests require much less machine resources and tests are executed faster, but it does not provide you with the full debugging possibilities.

Continuous Integration Server

Running tests from continuous integration server is based on the logics of running tests from the command prompt. Tests are run automatically on the server using preconfigured **phpunit.xml** file. Server is being continuously updated and then it runs all the tests according to defined schedule. This way of running tests is totally automated unlike running tests from a command prompt.

Results of Running Tests

You may view results of running tests with the help of standard command prompt output. In case you need more precise error localization you

might want to use additional PHPUnit logging tools. Logs are stored in a separate folder **tmp** and you can always access them for analysis.

Command Prompt

After executing ***runtests.bat*** in the command prompt you can see current progress of running your tests. It provides common overview without detailed information:

Example of Test Results Display in Text File Format

```
..............F..............F.........................  63 / 228 ( 27%)
...S.......................................................... 126 / 228 ( 55%)
.........I.....E..........I.................................. 196 / 228 ( 75%)
..................................E.......................... 228 / 228 (100%)
Tests: 228, Assertions: 396, Failures: 2, Errors: 2, Incomplete: 2, Skipped: 1.
```

More details about PHPUnit results generation you can see on the website http://www.phpunit.de/manual/current/en/.

Logs

PHPUnit logging tools provide more detailed information about test results. As soon as you execute ***runtests.bat***, log files are created in the same directory and they contain the detailed information about test execution.
Log files are created in several different formats: plain text, XML, HTML etc. Several of them are provided in the examples below:

Example of Text Execution Progress Display in Text File Format

```
..............................E.........................  63 / 527 ( 11%)
.........F.................................................. 126 / 527 ( 23%)
............................................................ 189 / 527 ( 35%)
.....................................SSSS.................. 252 / 527 ( 47%)
............................................................ 315 / 527 ( 59%)
............................................................ 378 / 527 ( 71%)
............................................................ 441 / 527 ( 83%)
....................FFF.....F....SSS.......F................. 504 / 527 ( 95%)
.......................
```

Example of Results Display in JSON File Format


```
{
    "event": "suiteStart",
    "suite": "All Tests",
    "tests": 527
} {
    "event": "suiteStart",
    "suite": "AdminUser_DeleteTest",
    "tests": 2
} {
    "event": "testStart",
    "suite": "AdminUser_DeleteTest",
    "test": "AdminUser_DeleteTest::test_DeleteAdminUser_Deletable"
} {
    "event": "test",
    "suite": "AdminUser_DeleteTest",
    "test": "AdminUser_DeleteTest::test_DeleteAdminUser_Deletable",
    "status": "pass",
    "time": 45.594885826111,
    "trace": [],
    "message": ""
} {
    "event": "testStart",
    "suite": "AdminUser_DeleteTest",
    "test": "AdminUser_DeleteTest::test_DeleteAdminUser_Current"
} {
    "event": "test",
    "suite": "AdminUser_DeleteTest",
    "test": "AdminUser_DeleteTest::test_DeleteAdminUser_Current",
    "status": "pass",
    "time": 17.709816932678,
    "trace": [],
    "message": ""
} {

...
```

Example of Results Display in TAP File Format

```
TAP version 13
ok 1 - AdminUser_DeleteTest::test_DeleteAdminUser_Deletable
ok 2 - AdminUser_DeleteTest::test_DeleteAdminUser_Current
ok 3 - AdminUser_CreateTest::test_Navigation
ok 4 - AdminUser_CreateTest::test_WithRequiredFieldsOnly
ok 5 - AdminUser_CreateTest::test_WithUserNameThatAlreadyExists
ok 6 - AdminUser_CreateTest::test_WithUserEmailThatAlreadyExists
ok 7 - AdminUser_CreateTest::test_WithRequiredFieldsEmpty with data set #0 ('user_name', 1)
ok 8 - AdminUser_CreateTest::test_WithRequiredFieldsEmpty with data set #1 ('first_name', 1)
ok 9 - AdminUser_CreateTest::test_WithRequiredFieldsEmpty with data set #2 ('last_name', 1)
ok 10 - AdminUser_CreateTest::test_WithRequiredFieldsEmpty with data set #3 ('email', 1)
ok 11 - AdminUser_CreateTest::test_WithRequiredFieldsEmpty with data set #4 ('password', 2)
ok 12 - AdminUser_CreateTest::test_WithRequiredFieldsEmpty with data set #5 ('password_confirmation',
1)
ok 13 - AdminUser_CreateTest::test_WithSpecialCharacters_exeptEmail
ok 14 - AdminUser_CreateTest::test_WithLongValues
ok 15 - AdminUser_CreateTest::test_WithInvalidPassword with data set #0 (array('1234567890',
'1234567890'), 'invalid_password')
ok 16 - AdminUser_CreateTest::test_WithInvalidPassword with data set #1 (array('qwertyqw',
'qwertyqw'), 'invalid_password')
ok 17 - AdminUser_CreateTest::test_WithInvalidPassword with data set #2 (array('123qwe', '123qwe'),
'invalid_password')
ok 18 - AdminUser_CreateTest::test_WithInvalidPassword with data set #3 (array('123123qwe',
'1231234qwe'), 'password_unmatch')
ok 19 - AdminUser_CreateTest::test_WithInvalidEmail with data set #0 ('invalid')
ok 20 - AdminUser_CreateTest::test_WithInvalidEmail with data set #1 ('test@invalidDomain')
ok 21 - AdminUser_CreateTest::test_WithInvalidEmail with data set #2 ('te@st@magento.com')
ok 22 - AdminUser_CreateTest::test_InactiveUser
ok 23 - AdminUser_CreateTest::test_WithRole
ok 24 - AdminUser_CreateTest::test_WithoutRole
ok 25 - AdminUser_LoginTest::loginValidUser

...
```

Example of Results Display in HTML File Format

```
<html>
    <body>
        <h2 id="AdminUser_DeleteTest">
            AdminUser_Delete
        </h2>
        <ul>
            <li>
                test DeleteAdminUser Deletable
            </li>
            <li>
                test DeleteAdminUser Current
            </li>
        </ul>
        <h2 id="AdminUser_CreateTest">
            AdminUser_Create
        </h2>
        <ul>
            <li>
                test Navigation
            </li>
            <li>
                test WithRequiredFieldsOnly
            </li>
            <li>
                test WithUserNameThatAlreadyExists
            </li>
            <li>
                test WithUserEmailThatAlreadyExists
            </li>
            <li>
                test WithRequiredFieldsEmpty
            </li>
            <li>
                test WithSpecialCharacters exeptEmail
            </li>
            <li>
                test WithLongValues
            </li>
            <li>
                test WithInvalidPassword
            </li>
            <li>
                test WithInvalidEmail
            </li>
            <li>
                test InactiveUser
            </li>
            <li>
                test WithRole
            </li>
            <li>
                test WithoutRole
            </li>
        </ul>

...
```

Example of Results Display in Testdox-Text

```
AdminUser_Delete
 [x] test DeleteAdminUser Deletable
 [x] test DeleteAdminUser Current

AdminUser_Create
 [x] test Navigation
 [x] test WithRequiredFieldsOnly
 [x] test WithUserNameThatAlreadyExists
 [x] test WithUserEmailThatAlreadyExists
 [x] test WithRequiredFieldsEmpty
 [x] test WithSpecialCharacters exeptEmail
 [x] test WithLongValues
 [x] test WithInvalidPassword
 [x] test WithInvalidEmail
 [x] test InactiveUser
 [x] test WithRole
 [x] test WithoutRole

AdminUser_Login
 [x] Login valid user
 [x] Login empty one field
 [x] Login non existant user
 [x] Login incorrect password
 [x] Login inactive admin account
 [x] Login without permissions
 [x] Forgot empty password
 [x] Forgot password invalid email
 [x] Forgot password correct email
 [ ] Forgot password old password

...
```

Example of XML File Format

```
<testsuites>
    <testsuite name="All Tests" tests="194" assertions="713" failures="0" errors="0"
time="2974.288380">
    <testsuite name="AdminUser_CreateTest" file="D:\Work\selenium-saas\tests\AdminUser\CreateTest.php"
fullPackage="selenium.tests" package="selenium" subpackage="tests" tests="22" assertions="68"
                    failures="0" errors="0" time="491.137801">
        <testcase name="test_Navigation" class="AdminUser_CreateTest"
file="D:\Work\selenium-saas\tests\AdminUser\CreateTest.php" line="71" assertions="6"
time="26.675233"/>
        <!--...-->
    </testsuite>
    <!--...-->
</testsuites>
```

MTAF Instructions

After unpacking the downloaded project, the user will see the following structure of the Magento TAF:

Example of MTAF Structure

```
__/magento-afw-x.x.x
 |
 |__/config             // Contains files with Selenium Server configuration
 | |
 | |--browsers.yml      // Contains the list of browsers for testing
 | |--*.yml
 |
 |__/data                // Contains test Data Sets for running test cases
 | |
 | |--DataSetFile.yml    // Contains files with test data
 | |--*.yml
 |
 |__/lib                 // Contains Magento Test Automation Framework sources
 | |__/Mage
 |   |__/Selenium
 |     |--LibraryFile.php
 |     |--*.php
 |
 |__/tests               // Contains the files with test scripts divided by specific functional areas
 | |
 | |__/FunctionalityGroupName1  // example: /Customer/
 | | |--FGN1TestScript.php      // example: Register.php
 | |
 | |__/FunctionalityGroupName*
 |
 |__/uimaps              // Contains page elements UIMaps for all pages that need to be tested
 | |__/admin
 | | |--FunctionalityRelatesUIMAPFile.yml     // for example:  dashboard.yml
 | | |--*.yml
 | |__/front
 |   |--FunctionalityRelatesUIMAPFile.yml
 |   |--*.yml
 |
 |__phpunit.xml         // Contains the list of test cases that will be run
```

Custom TestScript Example

- Write a new TestScript by creating **CreateTest/test.php** with the following structure:

Example of Test Script Structure

```
class Something_Create_Test extends Mage_Selenium_TestCase
{
 protected function assertPreConditions()     // Preconditions
    {
        $this->loginAdminUser();             // Log-in
        $this->assertTrue($this->checkCurrentPage('dashboard'),
                'Wrong page is opened');
        $this->navigate('manage_stores');    // Navigate to System -> Manage Stores
        $this->assertTrue($this->checkCurrentPage('manage_stores'),
                'Wrong page is opened');
    }

 public function test_Navigation()          // Test case, which verify presence of a controls
on the page
    {
        $this->assertTrue($this->clickButton('create_store_view'),
                'There is no "Create Store View" button on the page');   // action: clickButton
        $this->assertTrue($this->checkCurrentPage('new_store_view'),
                'Wrong page is opened');
        $this->assertTrue($this->controlIsPresent('button', 'back'),
                'There is no "Back" button on the page');
        $this->assertTrue($this->controlIsPresent('button', 'save_store_view'),
                'There is no "Save" button on the page');
        $this->assertTrue($this->controlIsPresent('button', 'reset'),
                'There is no "Reset" button on the page');
    }

 /**
  * Create Store. Fill in only required fields.
  * Steps:
  * 1. Click 'Create Store' button.
  * 2. Fill in required fields.
  * 3. Click 'Save Store' button.
  * Expected result:
  * Store is created.
  * Success Message is displayed
  */
 public function test_WithRequiredFieldsOnly()
    {
        //Data
        $storeData = $this->loadData('generic_store', Null, 'store_name');
        //Steps
        $this->clickButton('create_store');
        $this->fillForm($storeData);
        $this->saveForm('save_store');
        //Verifying
        $this->assertTrue($this->successMessage('success_saved_store'), $this->messages);
        $this->assertTrue($this->checkCurrentPage('manage_stores'),
                'After successful creation store should be redirected to Manage Stores page');
    }
}
```

Creating Custom UIMap

Custom UIMap Template

- In general, the uimap template looks like the following:

Example of UIMap Template

```
_page_name
  |__mca: URL without BaseURL
  |
```

```
|__title: page title
|
|__uimap:
|  |
|  |__form: &formLink
|  |  |
|  |  |__tabs:
|  |  |  |
|  |  |  |__-
|  |  |  |  |
|  |  |  |  |__tab_name_1:
|  |  |  |  |  |
|  |  |  |  |  |__xpath: tabXpath
|  |  |  |  |  |
|  |  |  |  |  |__fieldsets:
|  |  |  |  |  |  |
|  |  |  |  |  |  |__-
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |__fieldset_name_1:
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |__xpath: fieldsetXpath
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |__buttons:
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |__fieldset_button_name_1: buttonXpath
|  |  |  |  |  |  |  |  |  |__fieldset_button_name_2: buttonXpath
|  |  |  |  |  |  |  |  |  |__fieldset_button_name_3: buttonXpath
|  |  |  |  |  |  |  |  |  |__ ...
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |__checkboxes:
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |__checkbox_name_1: checkboxXpath
|  |  |  |  |  |  |  |  |  |__checkbox_name_2: checkboxXpath
|  |  |  |  |  |  |  |  |  |__ ...
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |__dropdowns:
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |__dropdown_name_1: dropdownXpath
|  |  |  |  |  |  |  |  |  |__dropdown_name_2: dropdownXpath
|  |  |  |  |  |  |  |  |  |__ ...
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |__links:
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |__link_name_1: linkXpath
|  |  |  |  |  |  |  |  |  |__link_name_2: linkXpath
|  |  |  |  |  |  |  |  |  |__ ...
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |__multiselects:
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |__multiselect_field_name_1: Xpath
|  |  |  |  |  |  |  |  |  |__multiselect_field_name_2: Xpath
|  |  |  |  |  |  |  |  |  |__ ...
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |__fields:
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |__field_name_1: fieldXpath
|  |  |  |  |  |  |  |  |  |__field_name_2: fieldXpath
|  |  |  |  |  |  |  |  |  |__ ...
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |__radiobuttons:
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |__radiobutton_name_1: radiobuttonXpath
|  |  |  |  |  |  |  |  |  |__radiobutton_name_2: radiobuttonXpath
|  |  |  |  |  |  |  |  |  |__ ...
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |__required: [required_field_name1, required_field_name2, ....]
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |__-
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |__fieldset_name_2:
```


|__xpath: fieldsetXpath
|__ ...
|__ ...
|__-
|__fieldset_name_3:
|__xpath: fieldsetXpath
|__ ...
|__ ...
|__-
|__tab_name_2:
|__xpath: tabXpath
|__fieldsets:
|__-
|__fieldset_name_4:
|__xpath: fieldsetXpath
|__ ...
|__ ...
|__buttons:
|__button_name_1: buttonXpath
|__button_name_2: buttonXpath
|__messages:
|__success_saved_message: Xpath

```
    |  |  |__success_deleted_message: Xpath
    |  |  |__error_message: Xpath
    |  |  |__ ...
```

Custom UIMap Example

- Create a new *.yml file under /magento-afw-x.x.x/uimaps, using either the UIMap template or creating a custom one using the example:

Example of Custom UIMap

```
# 'Manage Stores' page
manage_stores:
  mca: system_store/
  title: Stores / System / Magento Admin
  uimap:
    form:
      fieldsets:
        -
          manage_stores:
            xpath: div[@id='storeGrid']
            buttons:
              reset_filter: button[span='Reset Filter']
              search: button[span='Search']
            links:
              select_store_view: td[normalize-space(@class)='a-left last']/a[text()='%NAME%']
            fields:
              website_name: input[@id='filter_website_title']
              store_name: input[@id='filter_group_title']
              store_view_name: input[@id='filter_store_title']
    buttons:
      create_website: button[span='Create Website']
      create_store: button[span='Create Store']
      create_store_view: button[span='Create Store View']
    messages:
      success_saved_store: li[normalize-space(@class)='success-msg']//span[text()='The store has been
saved.']
      success_saved_store_view: li[normalize-space(@class)='success-msg']//span[text()='The store view
has been saved']
      success_saved_website: li[normalize-space(@class)='success-msg']//span[text()='The website has
been saved.']
      success_deleted_store: li[normalize-space(@class)='success-msg']//span[text()='The store has
been deleted.']
      success_deleted_store_view: li[normalize-space(@class)='success-msg']//span[text()='The store
view has been deleted.']
      success_deleted_website: li[normalize-space(@class)='success-msg']//span[text()='The website has
been deleted.']
```

Creating Custom DataSet

Custom DataSet is a file which contains sets for form filling.

- Each set corresponds to each variant of form filling.
- Each set contains following pairs: *forms_field_name* : *value*

Custom DataSet Example

Create a new *.yml file under /magento-afw-x.x.x/data, for example with the following structure:

Example of Custom DataSet Structure

```
generic_store_view:                     // list of fields with test data value
  store_name: Main Website Store
  store_view_name: Test Store View Name
  store_view_code: test_store_view_code
  store_view_status: Enabled

all_fields_store_view:
  store_name: Main Website Store
  store_view_name: Test Store View Name 2
  store_view_code: test_store_view_code_2
  store_view_status: Enabled
  store_view_sort_order: 1
```

Creating Custom TestScript

Creating a custom test script is easy:

- **First**, you need to know what you want to test
- **Second**, create trivial TestCase
- **Third**, create TestScript, based on TestCase and example of code below

Creating Test Suite

The next step - Creating a test suite.

We`ll be use test-cases which was created for Auto-Testing.

Scenario 1 - Verifying Catalog Advanced Search page with valid data:

1. Open Main Page (Frontend);
2. Go to Catalog Advanced Search page;
3. Fill in all fields with valid data;
4. Select Tax Class - None;
5. Click "Search" button;

Expected result: Product should be displayed on the Advanced Search Results page;

Scenario 2 - Verifying Catalog Advanced Search error message with empty fields:

1. Open Main Page (Frontend);
2. Go to Catalog Advanced Search page;
3. Leave all fields empty;
4. Click "Search" button;

Expected result: Error message should be displayed "Please specify at least one search term.";

Appendix A. Installing Add-Ons for Firefox

To install add-ons for Firefox go to **Tools>Add-ons** menu and then use the following instructions:

- Firebug: allows you to edit, debug, and monitor CSS, HTML, and JavaScript in your application being tested
- Selenium IDE: allows you to record, edit, and debug Selenium tests
- ScreenGrab: saves entire webpages as images
- FirePath: is a Firebug extension that adds a development tool for editing, inspecting, and generating XPath 1.0 expressions, CSS 3 selectors and JQuery selectors
- Web Developer: is an extension which adds various web developer tools to the browser
- Remember Certificate Exception: allows you to automatically add Certificate Exception