Magento Test Automation Framework User's Guide

The Magento Test Automation Framework (MTAF) is a system of software tools used for running repeatable functional tests against the Magento application being tested. This framework can be used for automating most routine processes in Magento, such as:

- Registering or creating customers in the frontend and in the Admin Panel
- Adding addresses to customer accounts
- Creating websites, web stores, and store views

MTAF is used for both writing test automation scripts and for performing the actual testing. Test automation scripts created within the framework can be used for testing most Magento functionality which does not relate to an external system. This is a cross-platform solution (not dependent on a specific operating system). MTAF allows QA specialists to quickly develop all kinds of tests for the current Magento version, and the tests can be reused at any time. Framework users can run a single test independently, a group of tests together (a test suite), or all available tests from a single command.

This guide provides instructions on using the test automation framework. It explains the environment requirements for the MTAF and describes the process of running tests and reviewing the results.

This document is intended for specialists who are either interested or involved in using the Magento Test Automation Framework (hereinafter referred to as MTAF). This framework is used to run automated tests against a normally installed Magento application. Primarily, this document is aimed at Magento Quality Assurance specialists and PHP developers.

Note
Using the Magento Test Automation Framework requires advanced knowledge of XML, YAML, and PHP for installing the necessary working environment, creating functional tests, executing the tests created, and using test results properly.

Related Documents

- Magento Test Automation Framework Installation Guide

About this Document

This section is intended to clarify the structure, content and presentation of information in this document.

Conventions

The following text formatting conventions are used to accentuate specific types of information:

- **Bold** is used to highlight paths to files.
- ***Bold Italics*** is used to highlight file names.
- `Code Style` is used to highlight samples of code.

Glossary

Term	Description
Magento Test Automation Framework (MTAF)	The Magento Test Automation Framework (MTAF) is a system of software tools used for running repeatable functional tests against a normally installed Magento application. MTAF is used for both writing test automation scripts and for performing the actual testing. Test automation scripts created within the framework can be used for testing most Magento functionality which does not relate to an external system. This is a cross-platform solution (not dependent on a specific operating system). MTAF allows QA specialists to quickly develop all kinds of tests for the current Magento version, and the tests can be reused at any time. Framework users can run a single test independently, a bunch of tests together (a test suite), or all available tests.
Test Automation Script	A Test Automation Script is a PHP class dependent on the PHPUnit framework and the Selenium library. The script initiates the running of a specific test case or a suite of test cases.
DataSet	A DataSet is a data file or 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	A UI Map is used to define, store, and serve the UI elements of an application or website. In the case of Magento (website), the definition of UI elements depends on the Selenium technology, which uses strings and XPATH to locate and define UI elements. In other words, a UI Map is a repository of test script objects which correspond to UI elements of the application being tested.
User	When this document refers to the User, it refers to a Quality Assurance Engineer, PHP Developer, or other person with similar skills and responsibilities working with the MTAF.
Document Object Model (DOM)	The Document Object Model (DOM) is a cross-platform, 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 in use. The public interface of a DOM is specified in its application programming interface (API).

Scope

This section presents a brief overview of each chapter's contents.

- Introduction to Magento Automated testing describes the benefits and workflow of automated testing. It also provides the list of tools and frameworks used to maintain automated testing.
- MTAF Environment Requirements lists the software required to use MTAF and the list of Web browsers supported by Magento (and thus MTAF).
- MTAF File Formats describes the file formats used for MTAF files.
- MTAF Logical Structure provides a graphical representation of MTAF logic.
- MTAF File Structure provides an overview of files used for the MTAF (their location and content).
- Starting MTAF provides instructions for starting MTAF.
- Running Tests describes the process of running tests.
- Results of Running Tests provides examples of test results.
- MTAF Instructions provides instructions on custom UIMap, DataSet, test script and test suite creation.

Introduction to Magento Automated Testing

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

The benefits of using automated testing are:

- Simplified testing procedures which use existing automated tests
- Reduced test execution time and human resource requirements
- Complete control over the tests’ results (“actual results” vs. “expected results”)
- Freedom to quickly change a test’s preconditions and input data, and re-run tests dynamically

Currently, there are two major approaches used for automated testing: Unit Testing (hereinafter “UT”), and Automated Functional Testing (hereinafter “AFT”). In Magento, the UT approaches is rarely used, because it requires significant time and resources to establish dependencies between different modules.

AFT, for Magento, means an automated process used to diagnose whether the application's functionality meets defined functional requirements. Using the AFT approach confirms that the code for each specific function or feature accomplishes the corresponding tasks successfully. Secondly, running functional tests confirms that the application operates exactly as expected.

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

1. Identify tasks that the application (or feature) must accomplish.
2. Create necessary input data.
3. Define expected results to be used as a benchmark for application (or feature) functionality.
4. QA specialist executes a test.
5. QA specialist compares expected results with actual results, and determines whether the test has been passed successfully.

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

- **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 the ability to establish a testing infrastructure and to re-use it as often as needed, simply by creating unique parts for each particular test.
- **Selenium** is a well know open source testing framework, widely used for testing Web-based applications. It contains a variety of 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 Selenium's key features is support for executing tests on multiple browser platforms. Selenium uses a simple but powerful Document Object Model (DOM) which also allows tests to be exported to multiple programming languages and frameworks.
  The Selenium framework includes the following tools:
  - **Selenium Integrated Development Environment (IDE)** is a prototyping tool for building test scripts, 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, then exports them as a reusable script that can be later executed in one of many programming languages. 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; the commands have pre-defined parameters according to the context of the selected UI element. One can record and playback automated tests without needing to know a 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 for 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 developers and QA personnel to access Selenium Remote Controls without worrying about where the Selenium Remote Controls are located. As a developer or QA specialist, one can create a script as one would for Selenium Remote Control and run it as usual.

MTAF Environment Requirements

Before you start using the 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
- An 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, etc.)
- Selenium Remote Control (RC) 1.0.3 or later, 2.0 rc2
- TortoiseGIT (recommended, but optional)
- Magento Community Edition 1.5 or later

For details about installing and configuring the necessary applications, refer to the Magento Test Automation Framework Installation Guide.

Note

In addition, make sure that the following requirements are met:

- The 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 official Selenium documentation.
- A custom browser profile must be created to be used by Selenium for running automated tests. For more information about this, refer to the Magento Test Automation Framework Installation Guide.

MTAF Supported Web Browsers

- Mozilla Firefox 3.x or later
- Google Chrome
- Internet Explorer 6.0 and later
- Safari 5 or later

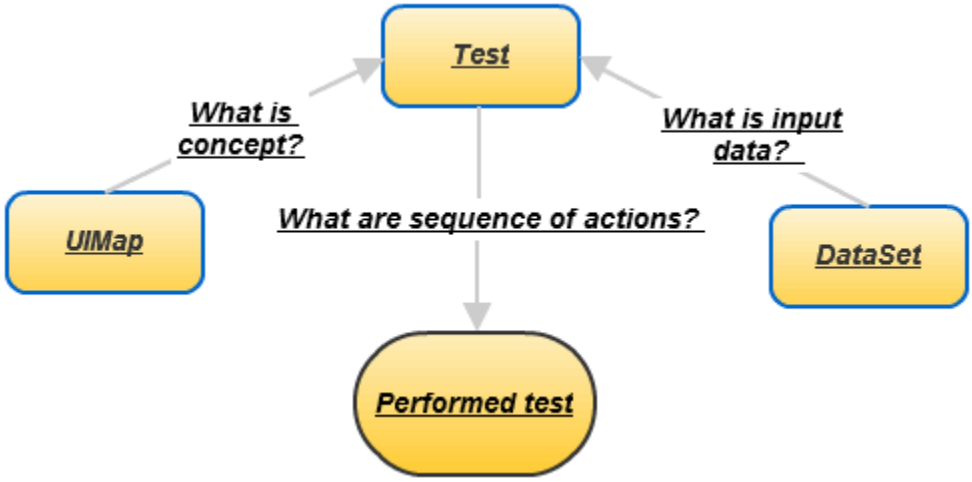
MTAF File Formats

There are three file formats used for MTAF files:

- **XML:** The only meaningful XML file used in MTAF is the ***phpunit.xml*** configuration file
- **PHP:** In MTAF, PHP files are basically used in two situations:
  - Test case files, located in the **tests** directory (test cases can be run based on the configuration defined in the ***phpunit.xml*** file, and use the input data set specified in the DataSet YAML files)
  - Framework library files
- **YAML:** Most MTAF files are created in the YAML format (a human-readable data serialization format); these can present either UIMap files or data set files.

MTAF Logical Structure

The following illustration explains the basics of MTAF logic.



A **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 each key is the alias of a UI element, and the corresponding value is an Xpath locator. In terms of the UIMap, each page rendered by a browser is presented as a Document Object Model (DOM). Such a model includes every single UI element that can exist on a page. Xpath locators are paths to such UI element in a DOM model. Thus, using Xpath expressions, it is possible to link any UI element on a page (even a hidden one) with a method that imitates an action being executed upon it (a method in the MTAF library). For example, one can use UIMaps (Xpath expressions) and a "click" method from the MTAF library to reproduce clicking any UI element on the specified web page. An example of using such an approach can be the re-testing of the localized application.

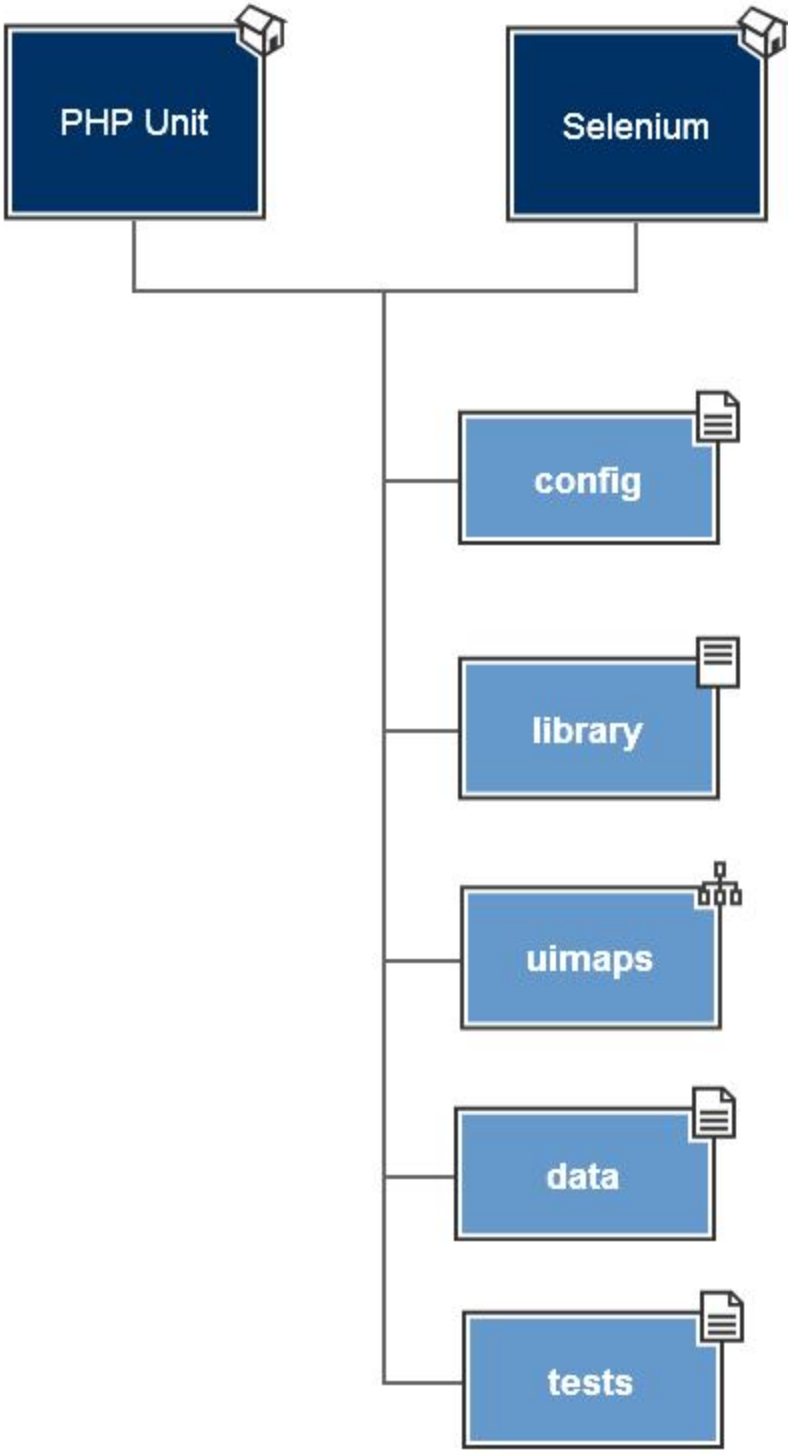
A **DataSet** is a pool of YAML files (data storage) which describe the data main-space being referred to by the test case files. For search convenience purposes the data main-space is logically split into several data files. Each data file can be related to a specific functionality being tested (and thus, related to specific test case files).

A **Test** is, generally speaking, a single action or sequence of actions that evaluates whether a specific feature meets functional requirements. It can be said that a test is a method in a class which is inherited from the basic test case class. The test calls a library function (or functions), which in turn executes the actions required to check the application's expected business logic.

MTAF File Structure

As you can see from the diagram at the right, all MTAF files are located in five main directories: **config**, **library**, **uimaps**, **data** and **tests**.

- The **config** folder contains two major files: ***config.yml*** and ***local.yml***. The ***config.yml*** file is used as a template for configuring the Selenium client. For more information on using these files, refer to the MTAF Configuration section.
- The **library** folder contains the entire class hierarchy from the API framework.
- The **uimaps** folder stores YAML files with descriptions of UIMaps for tests. It contains all UIMaps used in tests and may include some custom ones.
- The **data** folder contains all of the input parameters for all tests. Test data can be loaded to this folder with the help of the `loadData()` method as follows:
  ```
  $storeData = $this->loadData('generic_store', Null, 'store_name');
  ```
- The **tests** folder contains PHP files with test suite classes.



Starting MTAF

1. Run Selenium Server.
2. Execute runtests.bat (in Windows) through the command prompt for tests running.
3. Executing runtests.sh for Linux/*NIX OS is not implemented yet

Running Tests

When choosing the appropriate method for running tests you must consider the quanitity of tests (whether it is a single test, a test suite 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, such as test script debugging, manual start or continuous integration. Based on these factors, you may choose to run your tests from the Integrated Development Environment (IDE), from the command prompt or from the continuous integration server.

Integrated Development Environment (IDE)

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

The following instructions are specific to the NetBeans IDE. If you are using a different IDE, refer to that product's documentation for instructions.

Before running tests, you must add MTAF to the IDE.

Running All Tests

1. Select the project node in the file tree (Projects window).
2. Right-click this node, then select 'Test" or press Alt+F6.

Running Individual Tests

1. Select the test file (the file node) you want to run.
2. Right click the file in the file tree (Projects window), then click 'Run'.
   or
   Press Shift+F6 to run the test opened in the edit window.

Command Prompt

The command prompt provides you with full control over your tests and independence from platforms. It also enables you to start your tests manually. This option is most preferable when you have already debugged the tests and they can be run in production.

To work from the command prompt you need to have a BAT file with the ***phpunit.xml*** configuration. The same logic is used for running tests from a continuous integration server, which we will discuss later.

You can 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>
```

Running tests this way requires far fewer machine resources and the tests are executed more quickly, but it does not provide you with full debugging capabilities.

Continuous Integration Server

Running tests from a continuous integration server is based on the same logic as running tests from the command prompt. Tests are run automatically on the server using a preconfigured ***phpunit.xml*** file. The Magento installation on the server is continuously updated and then it runs all the specified tests according to the defined schedule. Unlike running tests from a command prompt, this method of running tests is totally automated.

Results of Running Tests

You may view results of running tests with the help of standard command prompt output. If you need more precise error localization you may want to use additional PHPUnit logging tools. Logs are stored in a separate **tmp** folder, and you can always access them for analysis.

Command Prompt

After executing **runtests.bat** from the command prompt you can see the current progress of running your tests. It provides an 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.
```

Legend:

- "." = test PASSed
- "S" = test SKIPped
- "I" = test INCOMPLETE
- "F" = test FAILed
- "E" = ERROR occurred during test

More details about PHPUnit results generation are available on the PHPUnit website.

Logs

PHPUnit logging tools provide more detailed information about test results. Once you execute ***runtests.bat***, log files containing detailed information about test execution are created in the same directory.

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 for the Magento TAF:

Example of MTAF Structure

```
__/magento-afw-x.x.x
 |
 |__/config             // Contains files with Selenium Server configuration
 | |
 | |--config.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
 | |__/paypal_ui
 |   |--FunctionalityRelatesUIMAPFile.yml
 |   |--*.yml |
 |__phpunit.xml         // Contains the list of test cases that will be run
```

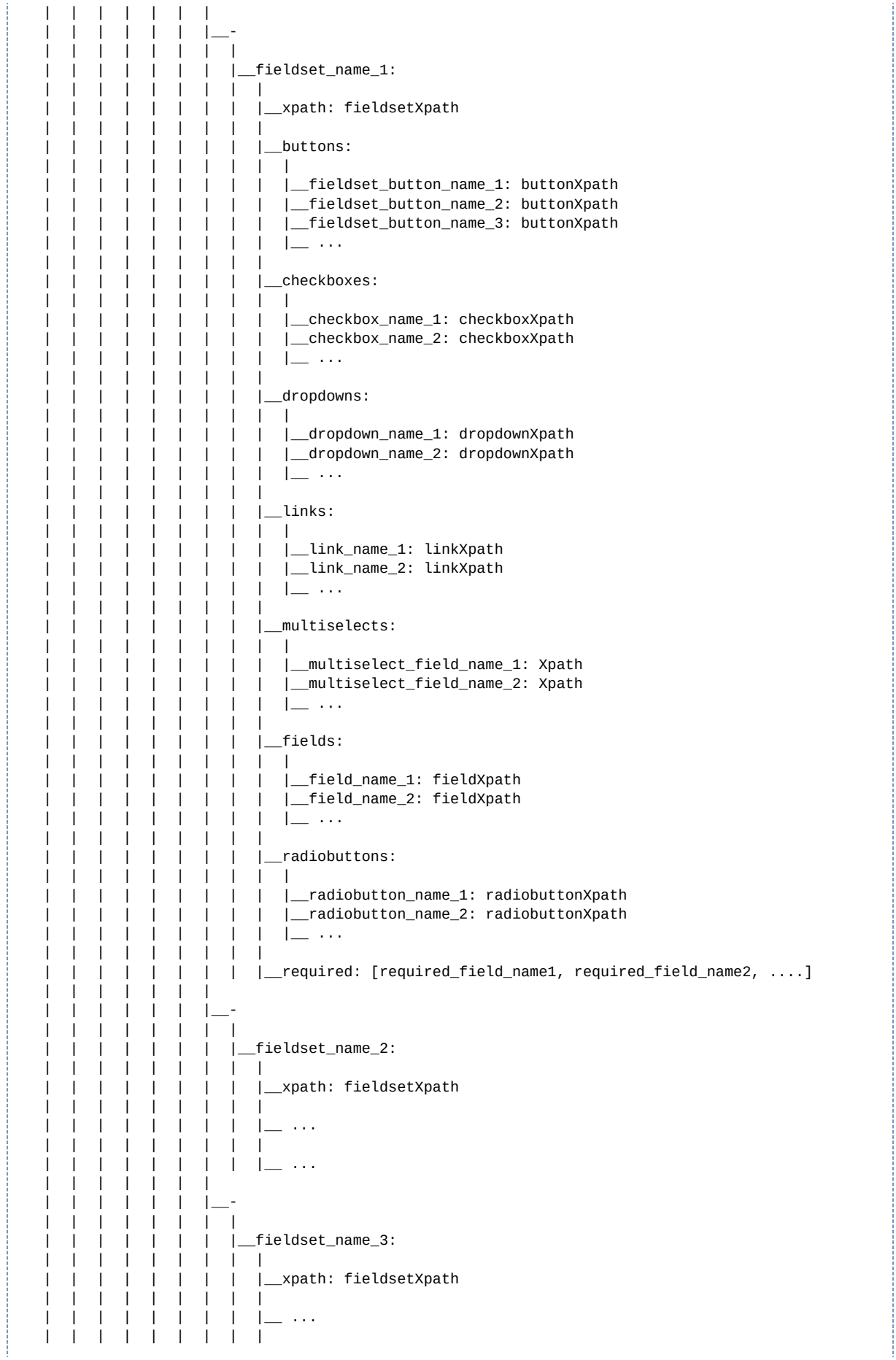
Creating a Custom UIMap

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**, either by using the UIMap template or by 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 a Custom DataSet

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

- Each set corresponds to a variation of form filling responses.
- Each set contains the following pairs: *forms_field_name* : *value*.

Custom DataSet Example

Create a new ***.yml*** file under **/magento-afw-x.x.x/data** 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 a Custom TestScript

Creating a custom test script is easy:

1. Know what you want to test.
2. Create a trivial TestCase.
3. Create a TestScript based on the TestCase and on the sample of code below.

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 a Test Suite

The next step is creating a test suite. In this example, we`ll be using test-cases which were created for Auto-Testing.

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

1. Open the Main Page (Frontend).
2. Navigate to the Catalog Advanced Search page.
3. Complete all fields with valid data.
4. Select Tax Class: None.
5. Click the "Search" button.

Expected result: The product should be displayed on the Advanced Search Results page.

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

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

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