WARRIOR FRAMEWORK

USER GUIDE

DOCUMENT HISTORY

Revision	Description	Author	Date
1	Preliminary	Brian Jishi	3/23/2014
2	Release 1	Brian Jishi	6/9/2015
3	Release 2	Brian Jishi	6/17/2015
4	Release 3	Sathyamoorthy Radhakrishnan	6/28/2015
5	Release 4 – Ninja (1.0)	Brian Jishi	9/8/2015
6	Release 5 – Ninja (1.5)	Sathyamoorthy Radhakrishnan	1/7/2016
7	Release 6 – Ninja (1.6)	Sathyamoorthy Radhakrishnan	1/20/2016
8	Release 7 – Ninja (2.0)	Sanika Kulkarni	9/29/2016

TABLE OF CONTENTS

WARRIOR USER GUIDE

KATANA USER GUIDE

WARHORN USER GUIDE

1.0 WHAT IS WARRIOR?

Warrior is a generic automation framework that automates software tasks and processes, and also functional, performance and solutions testing. It is designed and implemented by the SII team at Fujitsu Network Communications. Warrior is a keyword driven framework that uses xml based test data. Warrior was designed to allow users to effectively create reusable generic keywords and leave the execution intelligence out of the keywords to be handled by Warrior. The execution of the keyword will be handled by the framework and designed in the testcases and suites files. Warrior testcases, suites and projects are created in xml. These files may be edited in any xml editor or using Katana. Katana is a web based tool that simplifies editing and creation of xml testcases, suites and projects. Katana provides keyword searching, keyword documentation review and keyword, testcase and project execution instructions.

With Warrior, testers can execute keywords and testcases sequentially or in parallel with a simple selection in the xml file. In addition, users can implement conditions and loops for keyword execution, design keyword, testcase, and suite error handling by making selections in the xml file, eliminating the need for duplicate and tedious python scripts.

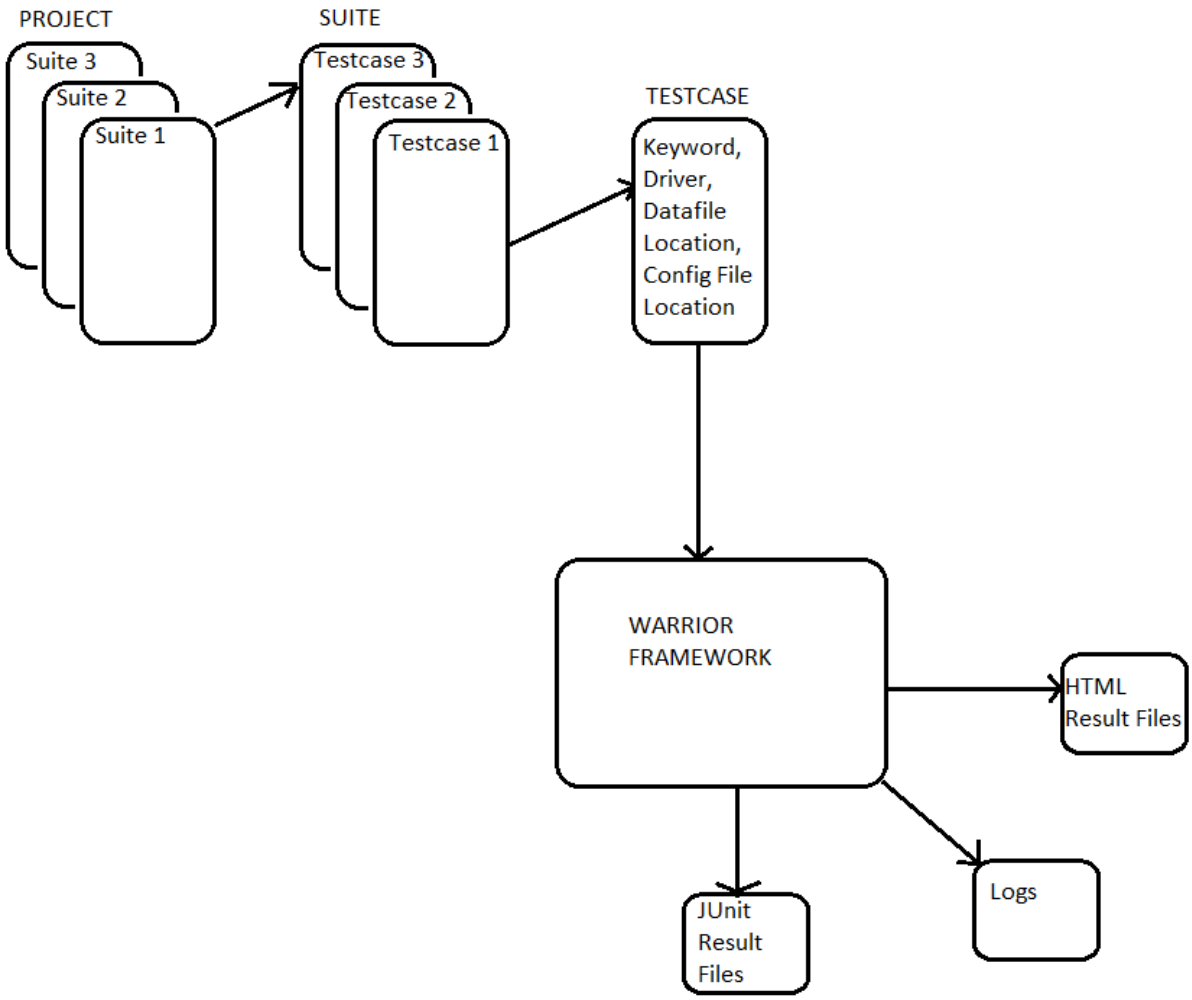
Warrior was designed for testing end to end solutions. Its architecture allows for testing multiple devices under test within a single testcase, suite or project file. For example, from a single testcase, warrior can execute keywords on a windows based system, a Network Element, a web-based application and a Linux based VM. Warrior also allows the integration of test sets such as Wireshark, Spirent and others within the testcase. In addition, Warrior can also integrate testcases created in other frameworks and execute them in conjunction with warrior testcases. Currently, Quick Test Pro (QTP), CSE and Selenium plugins are supported. Warrior's ability to integrate multiple systems, test sets, and frameworks within a single execution simplify the design and execution of end to end testing.

1.1 WHAT MAKES WARRIOR GENERIC?

Warrior's structure can be essentially broken down in the following components:

1. Framework
2. Keywords
3. Testcases (also, Suites and Projects)
4. Data Files

A diagrammatic representation of Warrior:



The above diagram gives a quick overview of what each component does in Warrior. The testcase stores the data file location and other execution details like which driver and keyword to use, whether a particular keyword impacts the testcase results, and so on. The data file stores the system information, the driver acts an interface between the testcase and the keyword, and keyword executes the command sent to it by the testcase. At the end, Warrior generates html result files, JUnit result files, and Logs that provide a complete documentation each and every keyword's execution results.

A project is a collection of Suites, which in turn is a collection of Testcases.

Every component here is independent of every other component. Exactly this, makes Warrior a customizable automation framework. Given a set of testcases and data files, changing the system information in the datafile or using a different data file altogether, lets you test different systems by

changing just a single file. Using a different testcase, or by changing the existing testcase, you can test different scenarios without writing a single line in Python.

Warrior comes with a set of Core (or built-in) Keywords that help you get started with automation. Those keywords may not be sufficient for you if you have specific requirements, in that case, Warrior allows you to write your own drivers and keywords, and those new, specific keywords can be used in your testcases.

All this makes Warrior one of the most generic and reusable automation framework out there.

1.2 WHAT NOT TO DO IN WARRIOR?

Warrior was designed to allow users to effectively create reusable generic keywords and leave the execution intelligence out of the keywords to be handled by Warrior. In order for your automation efforts to be successful using Warrior, please follow these recommendations:

- Do not write keywords as one large python script to be executed. Design your effort and break up the script into re-usable keywords. Warrior is not a python scripting tool. If your goal is to write testcases as python scripts, Warrior should not be used.
- Use the recommended Warriorspace directory structure for creating test cases, suites and projects. It will simplify the use of Warrior.
- Do not combine product drivers into one product driver. These drivers were designed to allow for smaller keyword files grouped by functionality or any logical demarcation you choose. If you combine into one driver, your keyword files will be large over time.

2.0 WARRIOR INSTALLATION

Warrior comes with a handy installation tool – Warhorn (Warhorn User Guide: Section 1) – that automates (ba-dum-tsss!) the process of installing Warrior on your system. Warhorn will install Warrior, the necessary keywords and Warriorspaces, and Warrior recommended dependencies for you when it is executed. More information about Warhorn can be obtained at the end of this document in the Warhorn User Guide (Section 1)

If you still want to proceed with installing Warrior without using Warhorn, then here are the steps for the manual install:

The Warrior automation framework is located in the following git repository:

http://rtx-swtl-git.fnc.net.local/scm/war/warrior_main.git

Development activities on Warrior framework will be done on the master branch of the above repository. Stable tested versions of Warrior framework will be marked as tags in the master branch at regular intervals.

After cloning specific version check the release notes available in warrior_main/release_notes.txt for details specific to the release.

Procedure to get a specific version of Warrior:

Clone warrior main

```
$ git clone http://rtx-swtl-git.fnc.net.local/scm/war/warrior_main.git
```

Go to warrior_main directory

```
$cd warrior_main
```

To check the list of versions available, execute "git tag --list" command

```
$ git tag –list
```

Response:

```
v1.0
v1.1
v1.2
v1.3
v1.4
v1.5
$
```

To check the current version you are at, execute "git branch" command

```
git branch
```

Response:

```
* master
$
```

- * indicates the active version.

- In the above example master is the active version.

- If the active version is master it means you are using a version of warrior framework that is under development. It is not a stable tested version.

4. To switch to a specific version from master, execute '`git checkout <version_name>`' command.

```
$ git checkout v1.4
```

Response:

```
Note: checking out 'v1.4'.
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.
If you want to create a new branch to retain commits you create, you may do
so (now or later) by using -b with the checkout command again. Example:
git checkout -b <new-branch-name>
```

```
HEAD is now at 146313d... reduce command timeout, wait after timeout=60
$
```

Execute git branch command to verify the active version.

```
git branch
```

Response:

```
* (HEAD detached at v1.4)
master
$
```

- * indicates the active version

Switch from one version to another (current=v1.4, switch to v1.3)

```
$ git checkout v1.3
```

Response:

```
Previous HEAD position was 146313d... reduce command timeout, wait after
timeout=60
HEAD is now at ecb6373... WAR-180, handle and prompt on timeout
$
```

Executing this command would let you verify the active version:

```
$ git branch
```

Response:

```
* (HEAD detached at v1.3)
master
$
```

Switch to master branch again.

```
$ git checkout master
```

Response:

```
Previous HEAD position was ecb6373... WAR-180, handle and prompt on timeout
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$
```

Executing this command would let you verify the active version:

```
$ git branch
```

Response:

```
* master
$
```

3.0 WARRIOR TOOLS

Warrior comes with a set of handy tools that help in easing up the Warrior setup, testcase, suite, and project creation, and their verifications.

3.1 WARHORN: AN OVERVIEW

Warhorn is an installation tool that automates the process of installing Warrior and external Keywords and Warriorspace for you. Warhorn also installs the dependencies that Warrior recommends you should install. Warhorn allows you to customize your local Warrior to suit your requirements.

A more detailed and step-by-step documentation on how to install and run Warhorn can be found in Warhorn User Guide: Section 1

3.2 KATANA: AN OVERVIEW

Katana is Warrior's web based testcase creation and execution tool. From Katana, users can create testcases, suites, projects, and input data files. Testcases, suites, projects, and input data files in Katana are web based forms that once saved, create the appropriate xml files. Each field in the form has a tooltip explaining its functionality. Katana also allows you to execute testcases, suites, and projects from its execute tab.

To start using Katana, refer to Katana User Guide: Section 1

3.3 IRONCLAW: AN OVERVIEW

Ironclaw is Warrior's xml verification tool. Ironclaw can be used to verify the structure of the xml files used by Warrior. Ironclaw will automatically detect the xml type (test case, test suite, project) and do the following as applicable:

3.3.1 VERIFICATION OF TESTCASE XML FILES:

Once Ironclaw identified that the xml is a test case xml. It will do the following:

1. Verify conformance to test case XSD.
2. Verify that the Input Data File defined in the xml file exists.
3. Verify the validity of keywords: will check if the keyword is in the corresponded actions package.
4. Verify that the driver exists in the Product Drivers folder.

3.3.2 VERIFICATION OF TESTSUITE XML FILES:

Once Ironclaw identified that the xml is a test suite xml. It will do the following:

1. Verify conformance to suite XSD.
2. Verify all test cases listed in suite exist.
3. Verify each test case in the suite as described in Section 7.0 of this User Guide

3.3.3 VERIFICATION OF PROJECT XML FILES:

Once Ironclaw identified that the xml is a project xml. It will do the following:

1. Verify conformance to project XSD.
2. Verify all test suites listed in project exist.
3. Verify each test suite as described in Section 8.0 of this User Guide

3.3.4 RUNNING IRONCLAW

To run Ironclaw execute the following command:

```
Warrior –ironclaw /path/testcase.xml
```

3.4 MOCKRUN: AN OVERVIEW

MockRun is a Warrior Tool which mocks the device under test to help automation teams run the Tests without the device (hardware) required for that test or its software simulation. MockRun helps automation teams design, develop, and deliver tests without being dependent on hardware or software availability, thus, helping agile teams increase their velocity of achieving automation goals.

Currently MockRun is supported only on CLI modules, i.e. it mocks and verifies all the Keywords in the CLI driver.

To run MockRun, type in this command:

```
python Warrior –mockrun path/to/testcase
```

When using `–mockrun`, the test script is validated for Errors and Warnings. These Errors and Warnings are logged in the Execution directory under the 'mock run' with a time stamp. To summarize these errors and warnings, you can use the `–summary` option:

```
python Warrior -summary –mockrun path/to/testcase
```

3.5 VIRTUAL MACHINE WITH WARRIOR ENVIRONMENT: AN OVERVIEW

To help a new user start off smoothly with Warrior, a virtual machine that can be run in a Virtual Box can be downloaded from:

```
\\rtxnasop01\public\Warrior\VM
```

This is `rtxnasop01` – an FNC server that stores a Virtual Box image which has the complete Warrior environment set up for you. This machine has:

1. Warrior (including Ironclaw and MockRun) – set to the latest version
2. Warhorn
3. Katana
4. All dependencies recommended by Warrior pre-installed.
5. Access to the Fujitsu git repositories
6. The PyCharm IDE

   More information about PyCharm can be found here: `https://www.jetbrains.com/pycharm/`
7. The Eclipse IDE with the PyDev extension

   More Information can be found here: `https://eclipse.org/` and `http://www.pydev.org/`
8. The IDLE

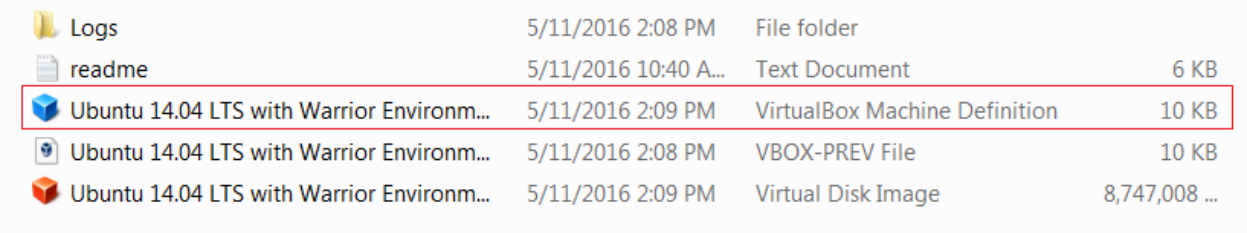
   More information can be found here: `https://docs.python.org/2/library/idle.html`
9. Java 8

Getting the Virtual Machine started is very simple.

Make sure you have Virtualbox installed on your system. You can download Virtualbox from https://www.virtualbox.org/wiki/Downloads

The, copy over the entire Ubuntu 1.04 LTS with Warrior Environment folder from the rtxnasop01 server and on to your machine.

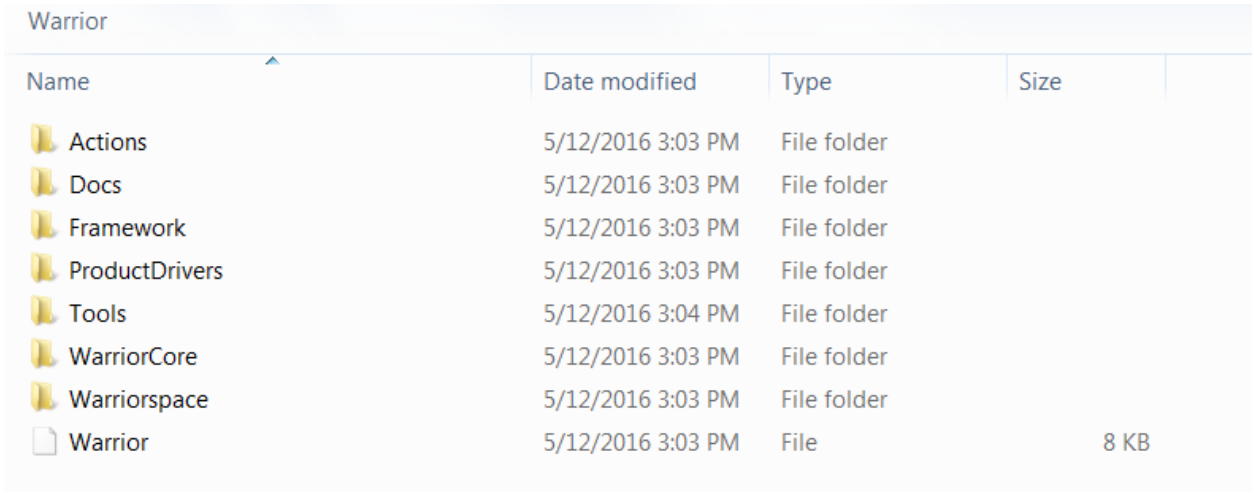
Open the Ubuntu 1.04 LTS with Warrior Environment folder, you will see the following files inside:



Double click on the marked file and it will open in the Virtualbox.

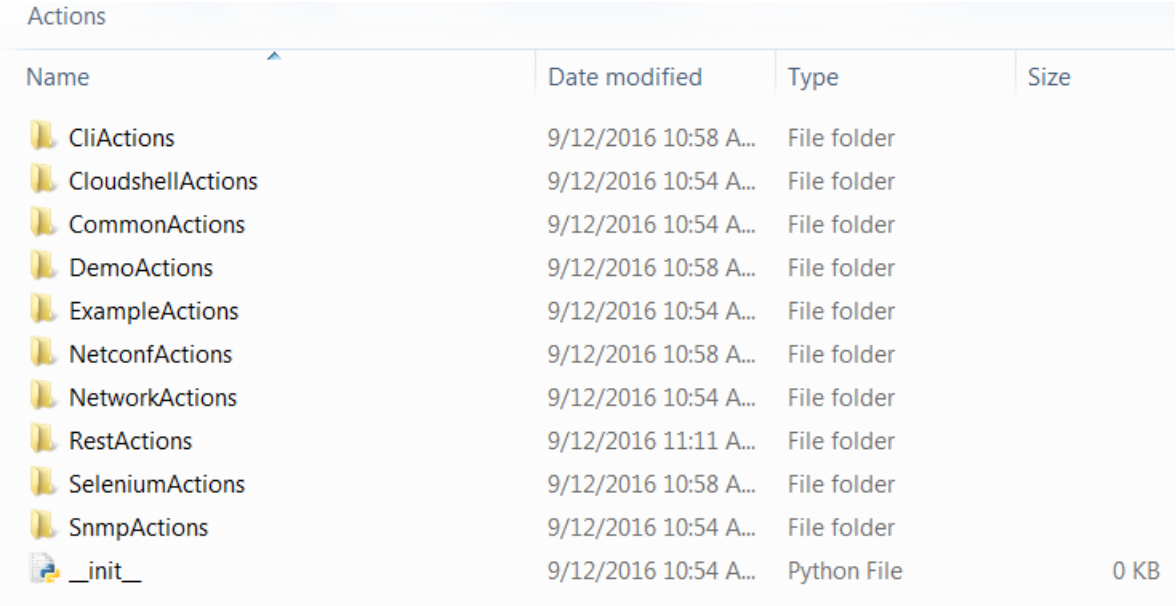
4.0 UNDERSTANDING THE WARRIOR DIRECTORY STRUCTURE

There are in total seven directories and one Warrior executable in the Warrior directory. What to expect in each directory and its utility has been explained in detail below. Whether you are a testcase developer or a keyword developer, acclimatizing yourself to Warrior directory structure is very important for a good understanding of Warrior. This is the current Warrior directory structure:



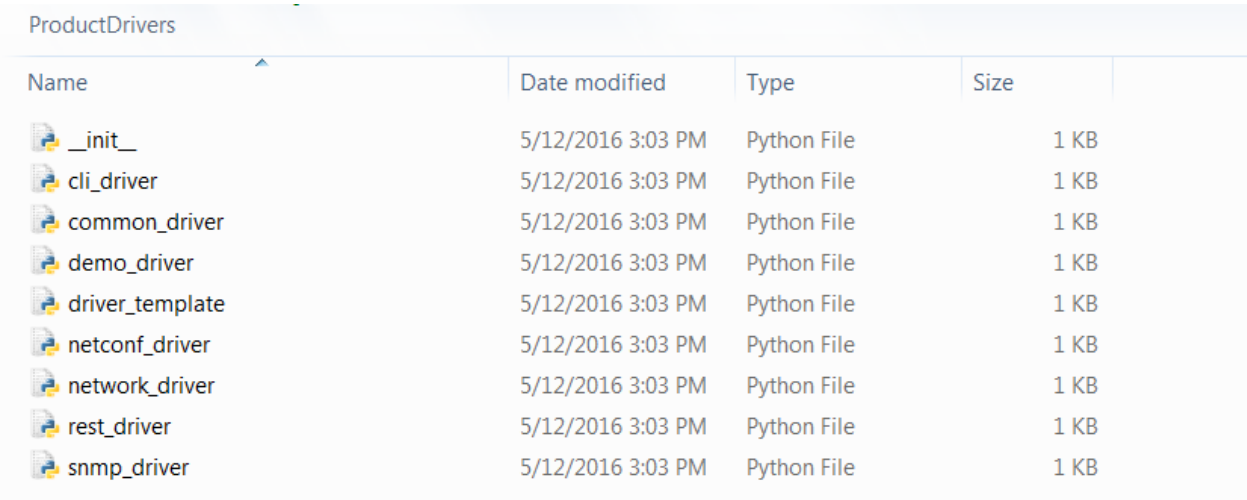
4.1 ACTIONS DIRECTORY

Warrior's Actions directory hosts the keyword driven libraries. The python libraries that contain the keyword functions are located in this directory. Any future keyword function development will need to be added to either an existing library or to a new library within the Actions directory. All these libraries are explained in detail in Section 5.0 of this User Guide



4.2 PRODUCTDRIVERS DIRECTORY

The Warrior driver files reside in the Product Drivers directory. Drivers are python files that make up the execution layer of the Warrior framework. The product drivers will interact with the xml based test case and start the execution of the steps based on the product driver indicated in the step. A user should be modifying or creating new product drivers. For a new product, the user must create a new product driver from the driver template provided in the Drivers directory.



The following are the existing drivers:

Product Driver	Description

cli_driver	Product driver that invokes execution of all keyword functions that can be used for command line interface testing.
common_driver	Product driver that invokes execution of all the common keyword functions that can be re-used among multiple products.
demo_driver	Product driver that invokes execution of all demo test related keyword
driver_template	Sample driver to be used in creating a new product driver. The instructions on how to create the new product driver are detailed as part of the comments in the code.
netconf_driver	Product driver that invokes execution of all keyword functions that can be used for netconf interface testing.
network_driver	Product driver that invokes execution of all keyword functions that can be used for all network related testing.
rest_driver	Product driver that invokes execution of all keyword functions that can be used for REST interface testing.
snmp_driver	Product driver that invokes execution of all SNMP keywords.

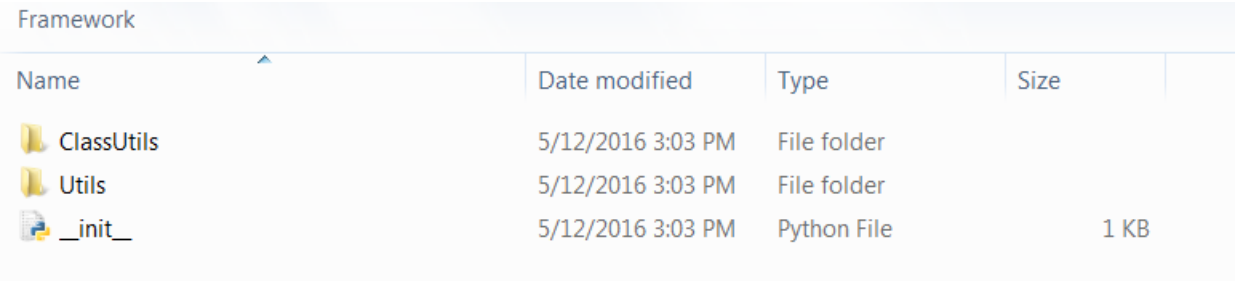
4.3 FRAMEWORK DIRECTORY

Warrior's system layer is in the Framework directory. The generic framework libraries that support the functional and execution layer are located in this directory. These libraries provide services such as data access, reading, writing, and log file creation, parsing and test results processing. Full documentation for all the modules can be obtained in the warrior_docs repository located at

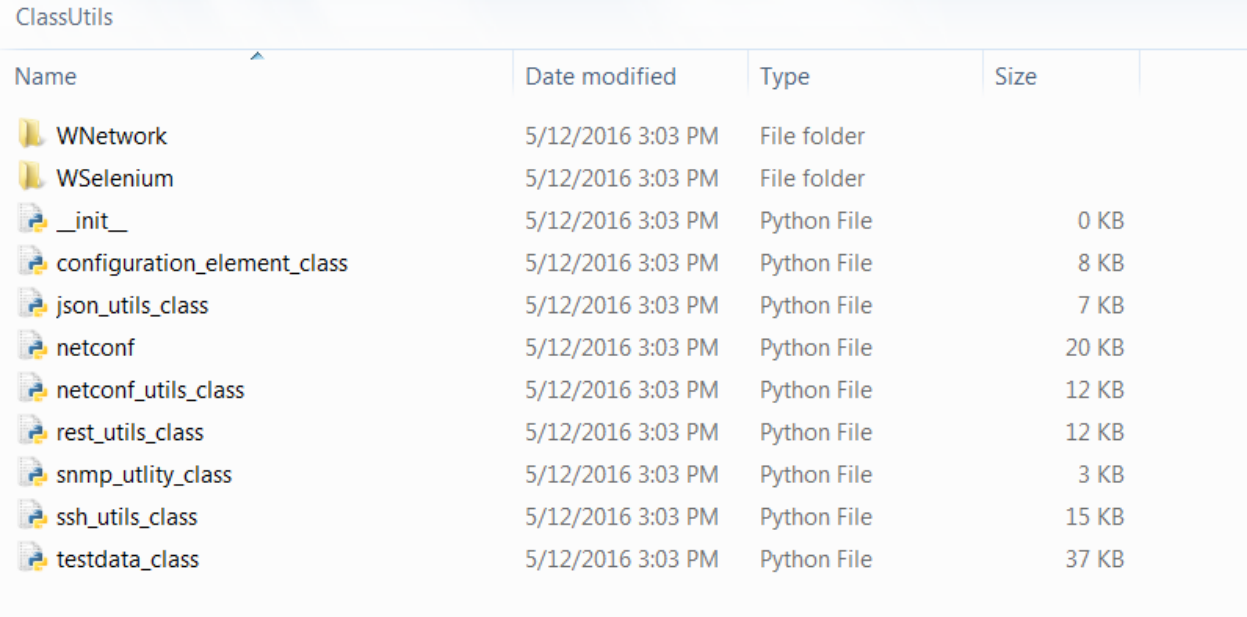
http://rtx-swtl-git.fnc.net.local/projects/WAR/repos/warrior_docs/browse

in `main/source/index.html`.

This Framework directory has two subdirectories: ClassUtils, and Utils.



Further, the ClassUtils directory has two subdirectories – WNetwork and WSelenium and eight files out of which one is netconf.py. This particular file was added in as a replacement for the ncclient dependency that Warrior had until v1.8. All the other files are class files that provide specific utilities for objects created by you. Every file has been described in detail below.



1. configuration_element_class.py

This module provides the following utilities for the ConfigurationElement object. The functions in this class provide utilities for finding a match against a regular expression, function for replacing variables with their values, functions for parsing data, getting a node in the tree and so on.

2. json_utils_class.py

This module provides API for JSON related operations. This class provides you with functions that can sort a JSON object, calculate difference between two JSON objects and files, writes JSON to a file, and print JSON.

3. netconf.py

This file was added in as a replacement for the ncclient dependency that Warrior had until v1.8. It is recommended that this file not be modified.

4. netconf_utils_class.py

These are API for operations related to NetConf Interfaces. The Requests package is used here. This class provides functionalities to open and close an SSH connection to a Netconf system. It also provides functionalities to get, edit, copy, and delete configuration from the datastore, lock, unlock, validate, and get configuration system, create and wait for subscriptions.

5. rest_utils_class.py

API for operations related to REST Interfaces. The Requests package is used in this module. This module provides you with HTTP methods such as post, get, put, patch, delete, options, and head.

6. snmp_utils_class.py

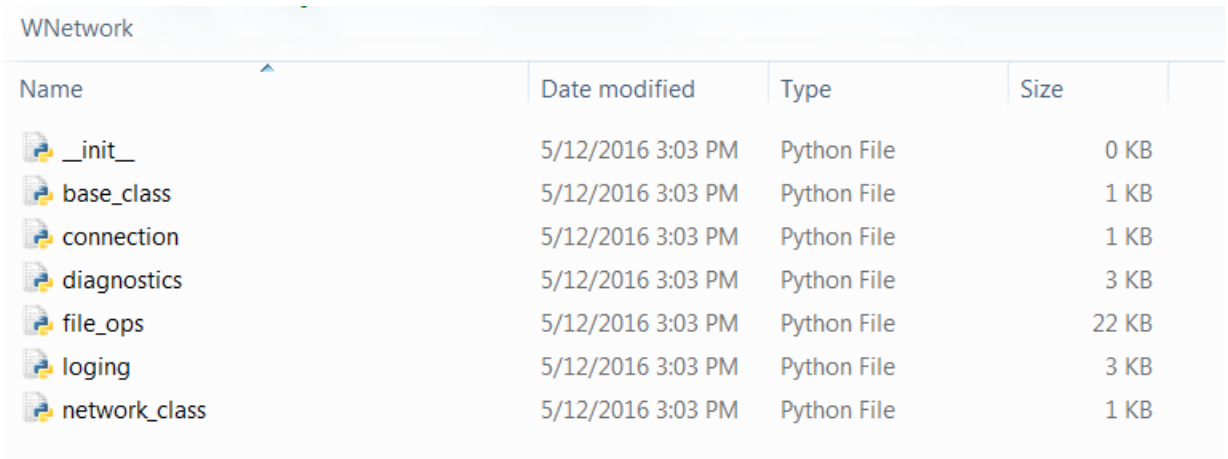
This is SNMP utility module using the python PYSNMP module. This module provides functionalities for generating commands, creating a communityData object, creating a UDP transport object, creating an MIB object, to check for Octet string.

7. testdata_class.py

This module has all testdata related class and methods that provide utilities for substituting the [VAR_SUB] patterns, resolving, getting, and validating the iteration patterns, expand, validate, and verify iteration patterns provided in each verification search provided for the command and more.

8. WNetwork

The WNetwork directory has all the utilities for a Network provided by Warrior.



8.1 base_class.py

Base class for Network package. This class in inherited in all classes of the Network package. Inheriting this class avoids the problem of calling object with **args/**kwargs while using super method (while invoking the methods of the Network package using an instance of Network class)

8.2 connection.py

This is the Warrior Network connectivity module. This has three defined functions: connect, connect_ssh, connect_telnet.

8.3 diagnostics.py

Warrior Network diagnostics module contains utilities to ping and trace the route from the remote host

8.4 file_ops.py

This is the Warrior Network File operations module which contains functionalities to start FTP and SFTP on the remote, FTP and SFTP from the remote host, and check if a certain file exists on remote host.

8.5 Loging.py

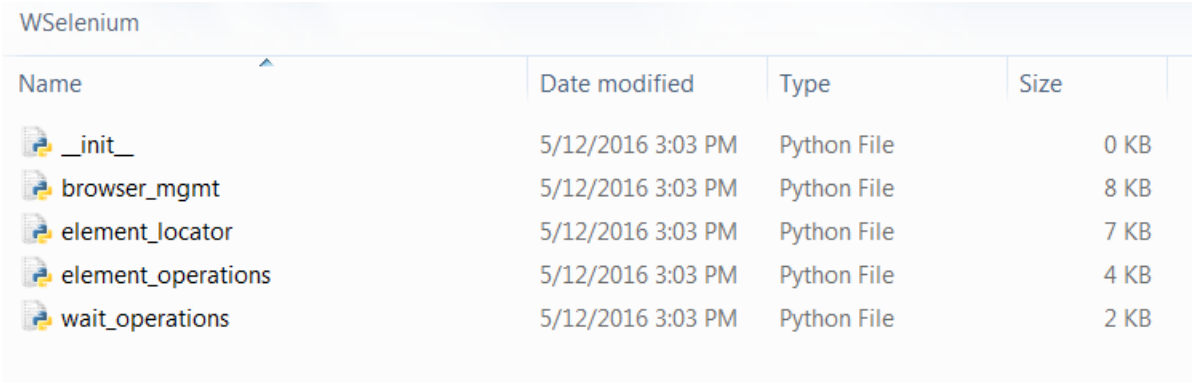
This module collects the response on a connected session as a separate thread. It can start and stop the thread, retrieve status its status, and collect logs generated by the thread.

8.6 Network_class.py

This is class that inherits all other classes in the Network package. Instance of this class may be used to invoke the methods of all the other classes in this package. While using the instance of this class, if the Base classes have same method name, then the method will be executed based on python's inheritance MRO (Method Resolution Order). Execute 'Network.__mro__' to find out the current MRO. Order is from left to right of the MRO.

9. WSelenium

The WSelenium package provides utilities for operations related to browser functionalities and element locators



9.1 browser_mgmt.py

This is the Selenium browser management library. It provides functionalities for opening and closing a browser, resizing the browser window, going back and reloading pages and so on.

9.2 Element_locator.py

This is the Selenium element locator library. It provides you with utilities that get elements by name, xpath, link text, partial link, ID, tag name, class name, and CSS selector.

9.3 Element_operation.py

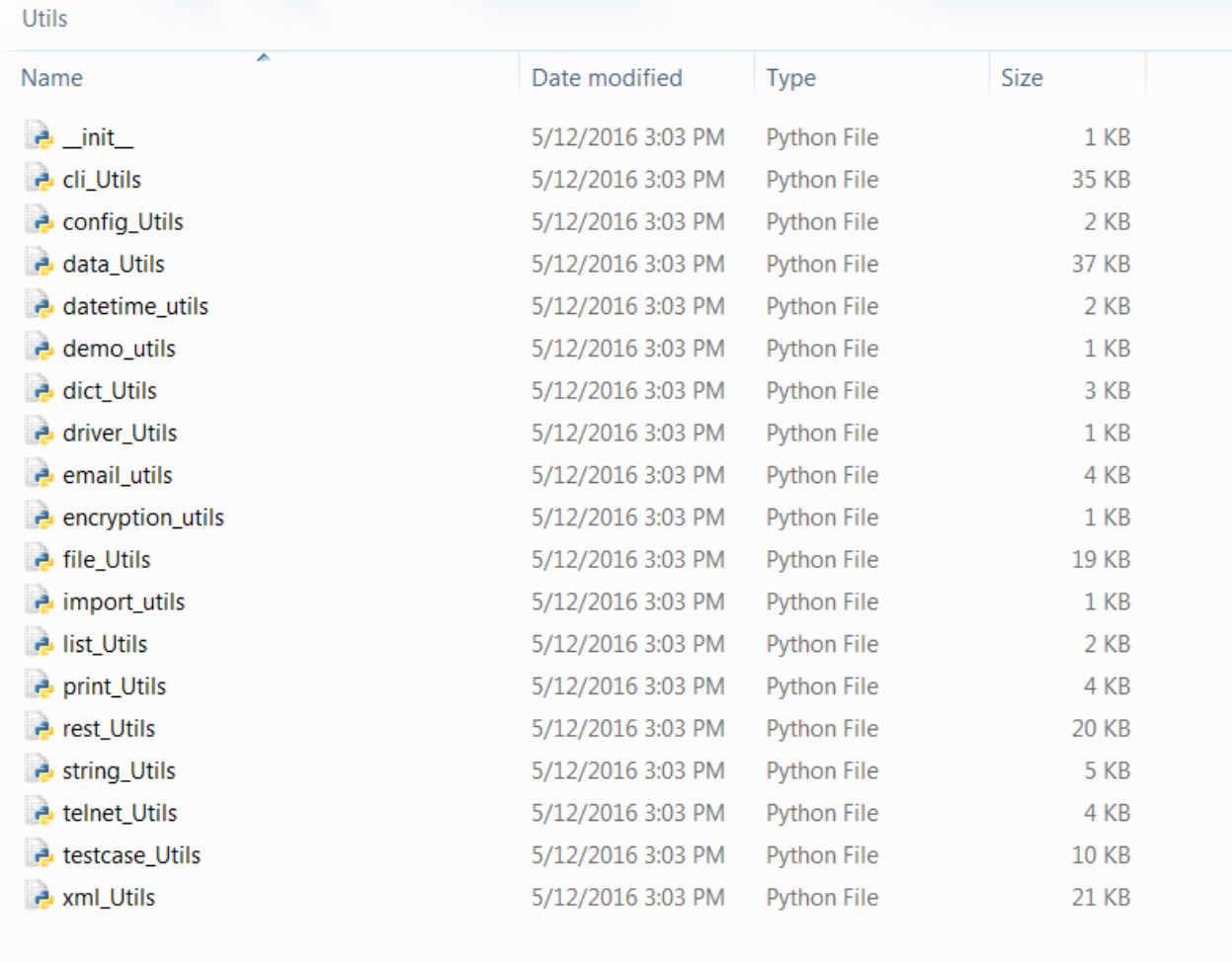
This is the Selenium element operations library. It provided functionalities for getting and clicking an element, performing an element action, sending keys, clearing text and so on.

9.4 Wait_operation.py

This is the wait or sleep operations library with functionalities for waiting till a particular stage of the HTML element is reached, and for getting a function provided by the locator.

10. Utils Directory

The Utils directory contains modules that provide various utilities. They have been described briefly below.



10.1 cli_Utils.py

This is the API for cli related operations. It has function that can connect and disconnect to SSH and Telnet sessions, send command and get the response of that command, get object session, connection port, and the dictionary containing all the responses, send ping and source ping, and more.

10.2 config_Utils.py

This module provides functionalities for setting the Logs directory, Log file, the datarepository, the Result file, the Debug file, and the JUnit file.

10.3 data_Utils.py

This is the API for testdata related functionality. This module provides utilities to get the system data, system and subsystem name, session ID, the variable configuration file, object from data repository, command details, command parameters, verification details, the variable configuration file details, REST

data, Netconf data, verify command response and response across multiple systems and other such functions. Please refer to the warrior_docs repository for full documentation.

10.4 datetime_Utils.py

This is the utility for date time functionalities such as getting the current time stamp.

10.5 demo_Utils.py

This API is for demo purposes only.

10.6 dict_Utils.py

This is the utility for functionalities related to dictionaries. This module provides utilities to convert string into dictionaries.

10.7 driver_Utils.py

This module gathers the argument information about the keywords, executes the keywords and reports the keyword status back to the product driver. This module will be deprecated in the next release of Warrior.

10.8 email_Utils.py

Provides utility to send email using SMPT

10.9 encryption_Utils.py

This file provides utilities for data encryption.

10.10 file_Utils.py

This file provides all utilities for file manipulation, creation, and deletion.

10.11 import_Utils.py

This file provides utilities for package verifications and imports.

10.12 list_Utils.py

This is the library to hold all APIs related to list operations

10.13 print_Utils.py

This is the Warrior Frameworks print library that provides utilities to print outputs on to the console and simultaneously marking them as 'Information', 'Error', 'Exception', and 'Debug'.

10.14 rest_Utils.py

This is the REST Utils library that provides utilities for all REST related functionalities like resolving the value of allow_redirects, timeouts, certificate, stream, data and other attributes that the user can provide through the testcase or the datafile.

10.15 string_Utils.py

This is the library to hold all APIs related to string operations

10.16 telnet_Utils.py

This library provides all telnet related utilities which can open a communication using the telnet protocol, connecting to an NE and closing it, writing and reading to and from the terminal, and getting the output until the prompt and returning the results.

10.17 testcase_Utils.py

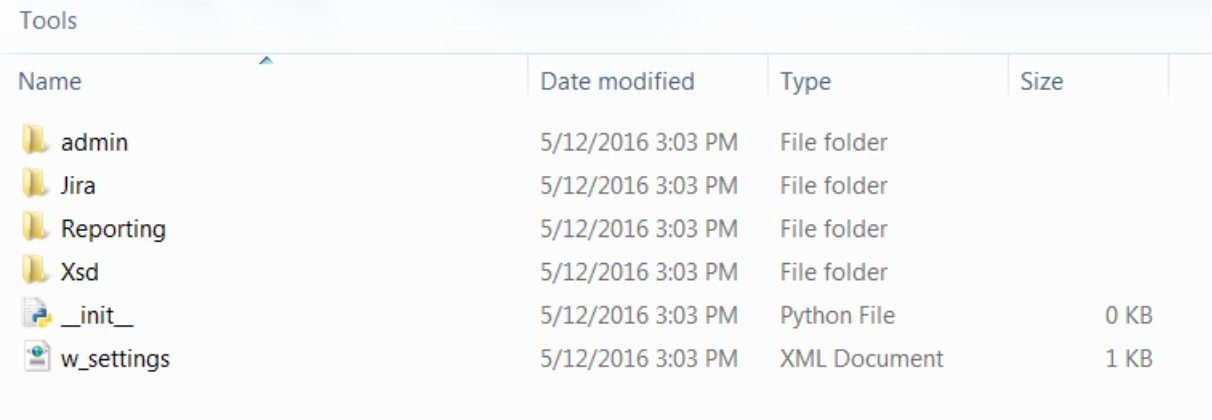
This module contains methods for testcase results related operations. The utilities provided by this module can report a substep status, open and close a file, create a root tag with the name Testcase, create tags with a particular name under the testcase tag, update the keyword result file, get value of impact, context, and onError from the testcase, and more such functionalities. Please refer to the full documentation available in the warrior_docs repository.

10.18 xml_Utils.py

This module contains methods for all xml related functionalities that can create a subelement, get value of an XML node by tag name and attribute, get the first and the last child of an XML node, get the root and the tree from a file, get child of an XML parent node, and so on. Please refer the warrior_docs repository for a complete documentation.

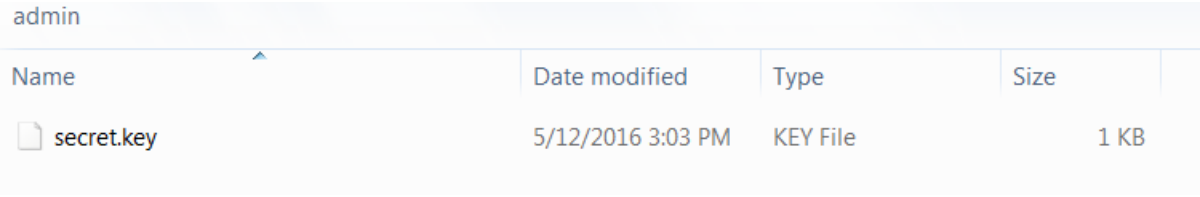
4.4 TOOLS DIRECTORY

The tools directory contains directories containing files specific to Warriors integration with other tools. There are four directories contained under this directory – admin, Jira, Reporting, Xsd and one file – w_settings.xml



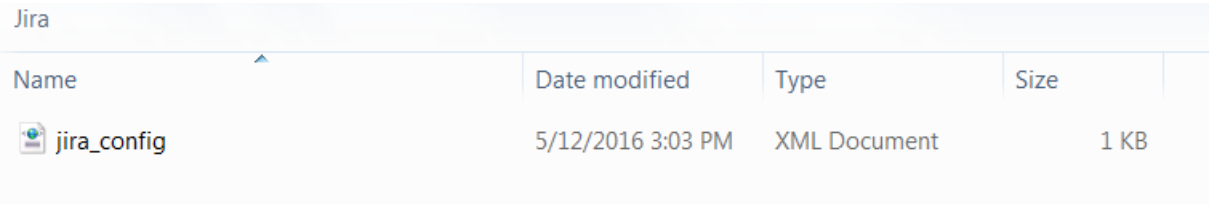
4.4.1 THE ADMIN DIRECTORY

Currently, the admin directory consists of a single file - the secret key file.



The secret.key is a plain text file that stores your AES 64 bit encoded key that encodes and decodes any sensitive information like passwords that may have to be entered into the XML file.

4.4.2 JIRA DIRECTORY



The Jira directory contains a jira.xml file to be configured be each installation. The XML file can contain multiple systems each being a separate Jira project. One project can be defined as a default project. When

executing Warrior through the CLI, if a Jira project is not specified, Warrior will use the project defined as default.

To define a project as default, it should be marked as default="true". Only ONE project can be default. If multiple projects are defined as default, Warrior will use the first project marked as default. If no project is marked default, then Warrior will use the first project in the xml file. In the screen shot below the last project is marked default:

```
<?xml version="1.0" ?>

<jira>

    <system name="warrior" >
        <url>https://demojiraplat.atlassian.net/</url>
        <username>admin</username>
        <password>satmu1323*</password>
        <Assignee>Automatic</Assignee>
        <project_key>WAR</project_key>
    </system>

    <system name="project1">
        <url>https://demojiraplat.atlassian.net/</url>
        <username>admin</username>
        <password>satmu1323*</password>
        <Assignee>Automatic</Assignee>
        <project_key>PROJ</project_key>
    </system>

    <system name="project2" >
        <url>https://demojiraplat.atlassian.net/</url>
        <username>admin</username>
        <password>satmu1323*</password>
        <Assignee>Automatic</Assignee>
        <project_key>PROJ2</project_key>
    </system>

    <system name="project3" default="true">
        <url>https://demojiraplat.atlassian.net/</url>
        <username>admin</username>
        <password>satmu1323*</password>
        <Assignee>Automatic</Assignee>
        <project_key>PROJ3</project_key>
    </system>

</jira>
```

4.4.2.1 AUTO DEFECT CREATION

Automatically create JIRA defects against the default JIRA project from jira_config file:

```
./Warrior -ad path/to/tc1.xml path/to/tc2.xml path/to/ts1.xml path/to/ts2.xml
path/to/proj1.xml
```

Automatically create JIRA defects against the provided JIRA project from JIRA config file:

```
./Warrior -ad -jiraproj jiraproj name path/to/tc1.xml path/to/tc2.xml
path/to/ts1.xml
```

Create defects in default JIRA project for all the defects JSON files present in defects_directory1, defects_directory2:

```
./Warrior -ujd -ddir path/to/defects_directory1 path/to/defects_directory2
```

Create defects in default JIRA project for the defects JSON files defects_json1 and defects_json2:

```
./Warrior -ujd -djson path/to/defects_json1 path/to/defects_json2
```

Create defects in provided JIRA project for all the defects JSON files present in defects_directory1 and defects_directory2

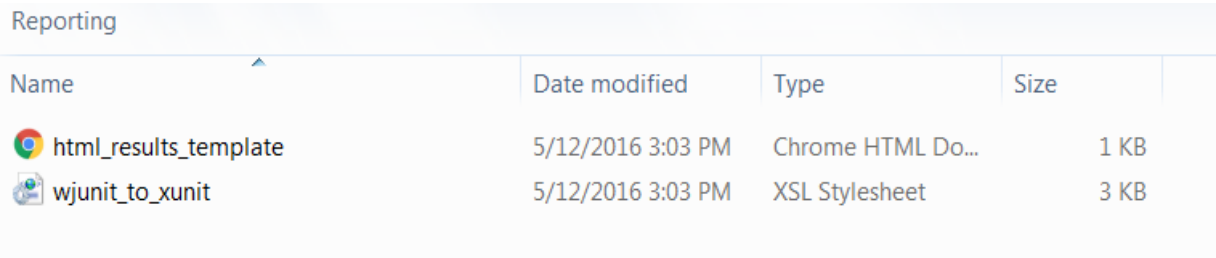
```
./Warrior -ujd -jiraproj jiraproj_name -ddir path/to/defects_directory1
path/to/defects_directory2
```

Create defects in provided JIRA project for the defects JSON files path/to/defects_json1 and path/to/defects_json2

```
./Warrior -ujd -jiraproj jiraproj_name -djson path/to/defects_json1
path/to/defects_json2
```

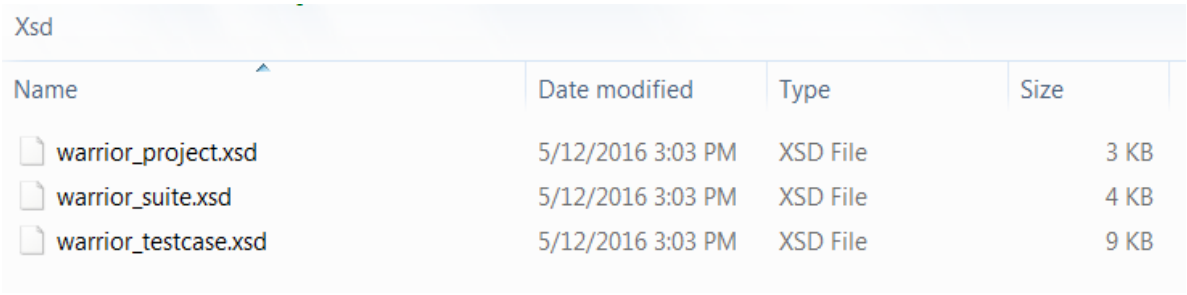
4.4.3 THE REPORTING DIRECTORY

This contains the files needed for the HTML file generation and reporting. It is recommended that these files not be modified.



4.4.4 THE XSD DIRECTORY

This directory contains the XSDs required by Ironclaw to verify if the Testcases, Suites, and Projects are correct. It is recommended that these files not be modified.



4.4.5 THE W_SETTINGS FILE

The w_setings.xml file that comes with Warrior looks something like this:

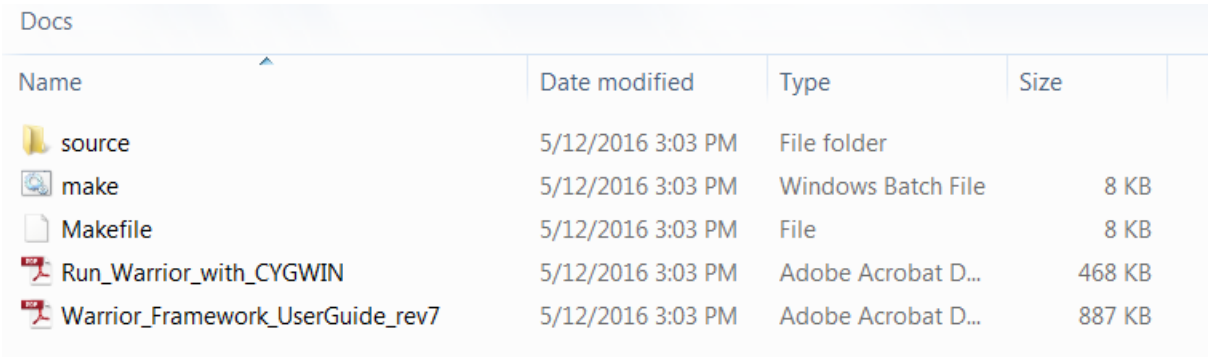
```
▼<Default>
  ▼<Setting name="def_dir">
     <Logsdir/>
    ▼<!--
       Optional, uses default, support absolute path, relative path and environment variable(${ENV.DEF_LOGSDIR})
     -->
     <Resultsdir/>
    ▼<!--
       Optional, uses default, support absolute path, relative path and environment variable(${ENV.DEF_RESULTSDIR})
     -->
   </Setting>
  ▼<Setting name="mail_to">
     <smtp_host/>
    ▼<!--
       Optional, use FNC smtp host smtp.fnc.fujitsu.com to notify test case execution results
     -->
     <sender/>
    ▼<!--
       Optional, use sender's email such as Warrior_PBO@fnc.fujitsu.com
     -->
     <receiver/>
    ▼<!--
       Optional, use receiver's email such as FNC.User@fnc.fujitsu.com
     -->
     <subject/>
    ▼<!--
       Optional, use this as subject line such as WARRIOR , additional test case execution results Pass/Fail and Test Execution File shall be appeneded to this
     -->
   </Setting>
 </Default>
```

Here you can add a path to the Logs directory and Results directory – so when you run Warrior, it would automatically use the paths given here, rather than the default.

Inside the second <Setting></Setting> tag, you can add the information about the host, sender, receiver, and subject so that after every Warrior run, you will receive an email with the results and logs attached to it. If these details are wrong or left unfilled, Warrior won't be able to send an email.

4.5 DOCS DIRECTORY

Warrior docs are generated using sphinx. The html files generated by Sphinx are located in the Docs directory of Warrior. Using a browser, open the Warrior_index.html file locate in the docs directory of Warrior to view the keywords. You will then be able to view and search for keywords and review the keyword documentation. The docs also will display the python libraries within the Framework directory.



source

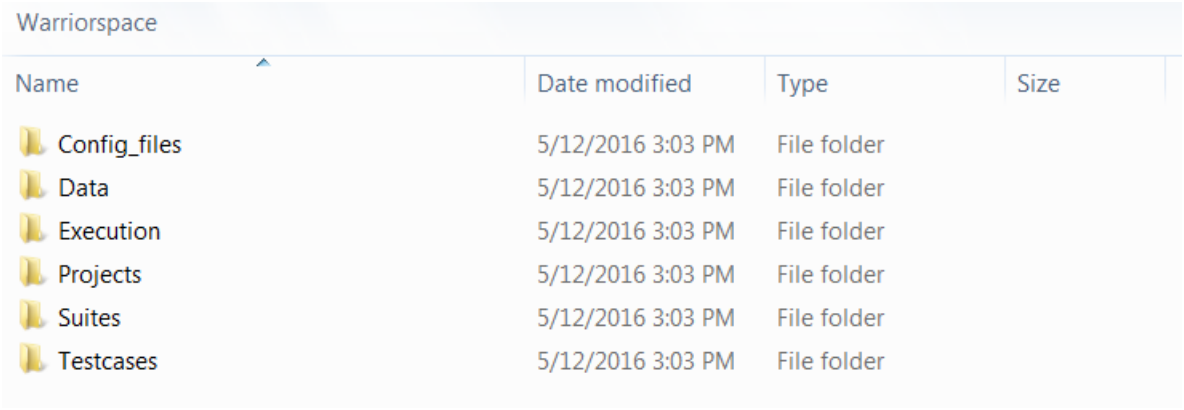
Name	Date modified	Type	Size
Actions.CliActions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.CommonActions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.CseActions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.DemoActions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.ExampleActions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.NetconfActions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.NetworkActions.Connection.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.NetworkActions.Diagnostics.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.NetworkActions.FileOps.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.NetworkActions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.QtpActions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.RestActions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.SnmpActions.rst	5/12/2016 3:03 PM	RST File	1 KB
Actions.SshActions.rst	5/12/2016 3:03 PM	RST File	1 KB
conf	5/12/2016 3:03 PM	Python File	10 KB
Framework.ClassUtils.rst	5/12/2016 3:03 PM	RST File	2 KB
Framework.ClassUtils.WNetwork.rst	5/12/2016 3:03 PM	RST File	2 KB
Framework.ClassUtils.WSelenium.rst	5/12/2016 3:03 PM	RST File	2 KB
Framework.rst	5/12/2016 3:03 PM	RST File	1 KB
Framework.Utils.rst	5/12/2016 3:03 PM	RST File	4 KB
modules.rst	5/12/2016 3:03 PM	RST File	1 KB
warrior_index.rst	5/12/2016 3:03 PM	RST File	1 KB

4.6 WARRIORCORE DIRECTORY

This directory contains the python files that compose Warrior's execution engine. Do not make any changes to the files in this directory.

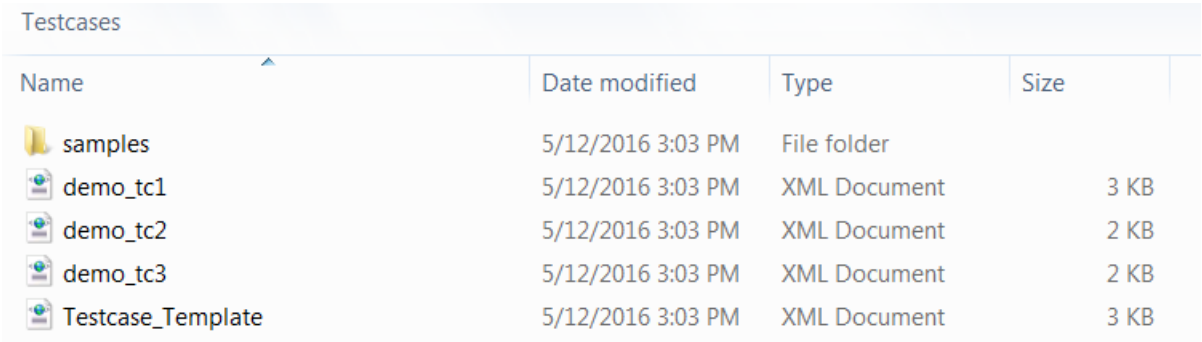
4.7 WARRIORSPACE DIRECTORY

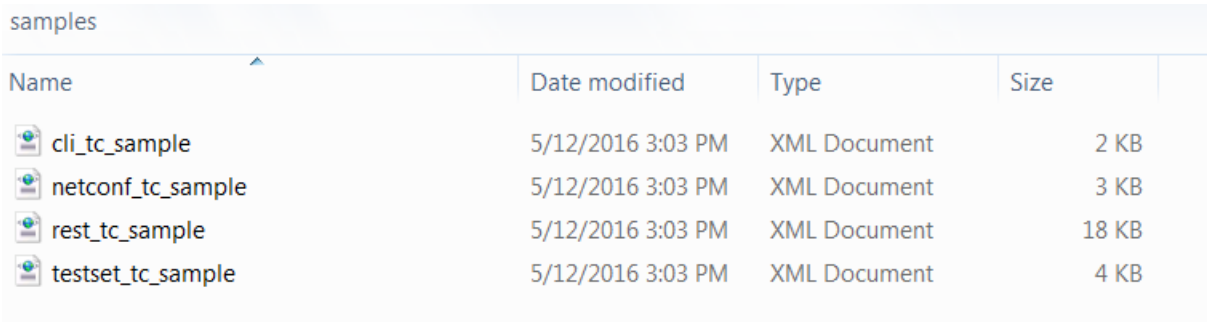
Warrior uses this directory as the default location for testcase, data files and execution results will reside.



4.7.1 TESTCASES DIRECTORY

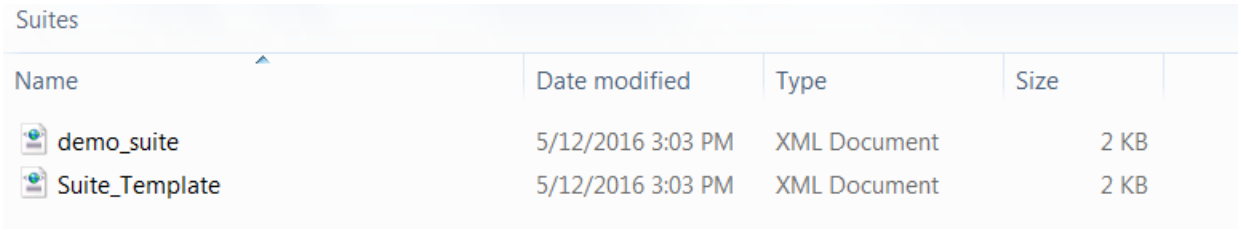
This directory will include the testcase.xml files that define the keywords to be executed. This is the default space provided by Warrior for you to save your testcases. You can use a different directory to save the testcases.





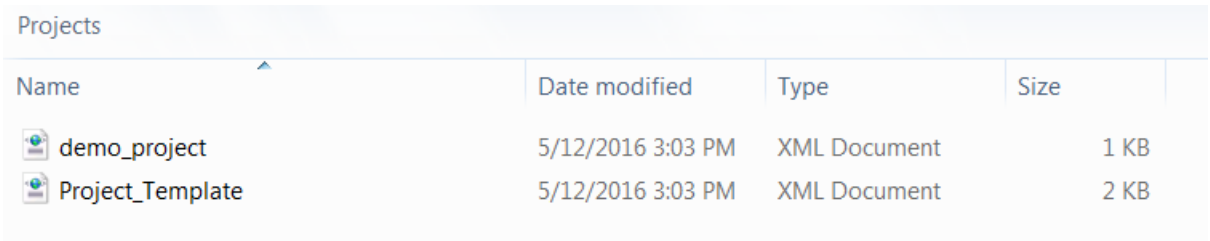
4.7.2 SUITES DIRECTORY

This directory will include the testsuite.xml files that define the test suites to be executed. This is the default space provided by Warrior for you to save your testsuites. You may use a different location.



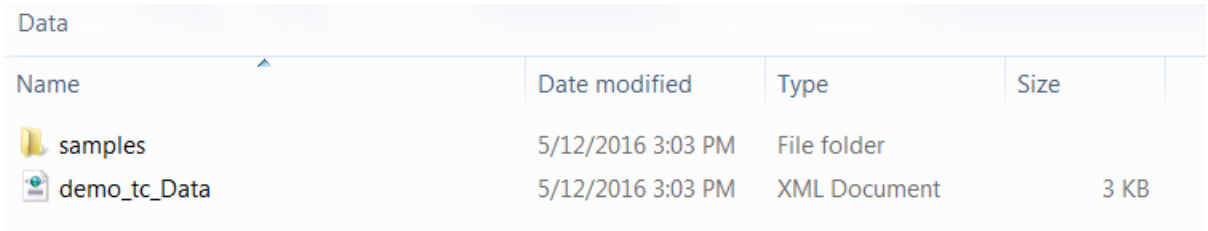
4.7.3 PROJECTS DIRECTORY

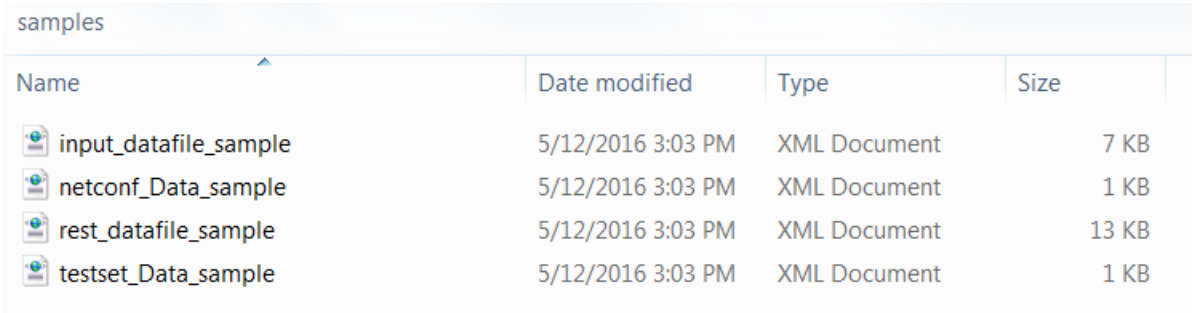
This directory will include the project.xml files that define the projects to be executed. This is the default space provided by Warrior for you to save your projects. You may use a different location.



4.7.4 DATA DIRECTORY

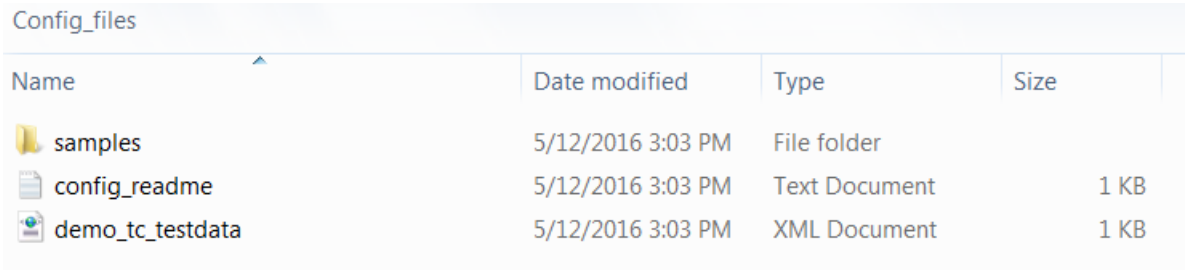
The Data directory will contain the input data files as indicated in the testcase.xml. If the testcase.xml does not define a file, Warrior will search for data file with the same name as the testcase_Data.xml. If a data is not found, Warrior will assume that the test case does not require a data file.

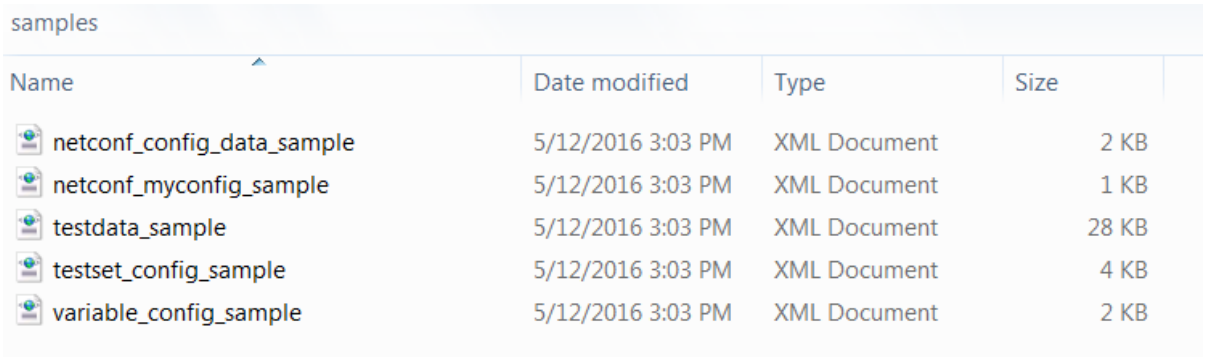




4.7.5 CONFIG_FILES DIRECTORY

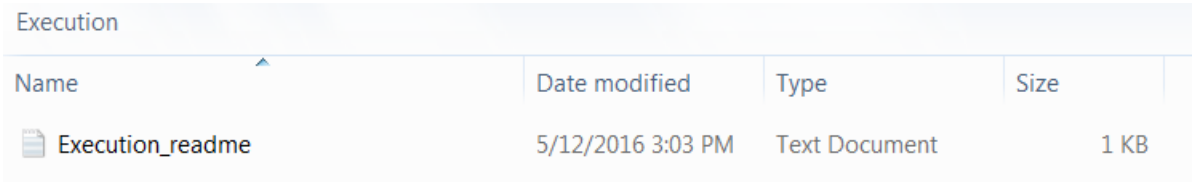
The Config_files directory will contain the configuration files as indicated in the input data files.





4.7.6 EXECUTION DIRECTORY

The Execution directory will be automatically created by Warrior under Warriorspace directory. If the Execution directory already exists, Warrior will use the existing directory. During the execution of a test case, suite or project, Warrior will create as subdirectory named after the time/date of the execution. Under the time/date directory, a project, suite or testcase directory will be created and the result and log files will be created under the applicable directory.

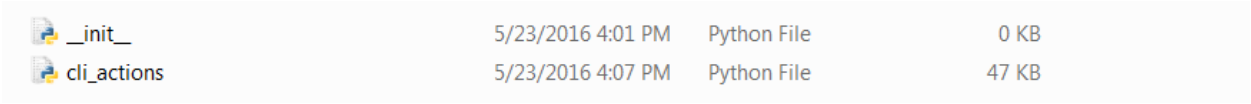


5.0 UNDERSTANDING THE BUILT-IN KEYWORDS

Warrior provides the user with a set of built-in generic keywords.

5.1 CLI ACTIONS PACKAGE

The CliActions package is the first Actions package that you will arrive at after opening the Actions directory. The CliActions class has methods (keywords) related to actions that can be performed on any command line interface.



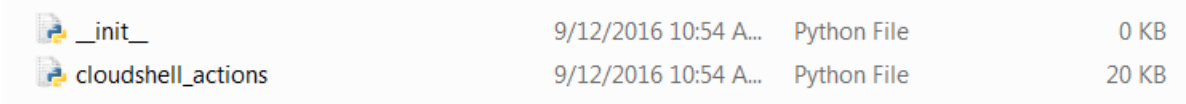
The cli_actions module includes these keywords:

Keyword	Description
connect	This is a generic connect that can connect to SSH/telnet based on the conn_type provided by the user in the input datafile.
disconnect	Disconnects/Closes session established with the system
connect_ssh	Connects to the SSH port of the given system or subsystems
connect_telnet	Connects to the telnet port of the given system and/or subsystem and creates a pexpect session object for the system
send_command	Sends a command to a system or a subsystem
send_all_testdata_commands	Sends all commands from all rows that are marked execute=yes from the testdata. This keyword expects the usage of warrior framework's recommended testdata xml files
send_commands_by_testdata_rownum	Sends all the commands from testdata that has row equal to the provided row_num. This keyword expects the usage of warrior framework's recommended testdata xml files.
send_commands_by_testdata_title	Sends all the commands from testdata that has title equal to the provided title. This keyword expects the usage of warrior framework's recommended testdata xml files
send_commands_by_testdata_title_rownum	Sends all the commands from testdata that has title/row equal to the provided title/row_num. This keyword expects the usage of warrior framework's recommended testdata xml files

set_session_timeout	Sets the timeout period for the SSH or telnet session
verify_session_status	Checks whether the SSH/telnet session is alive or not
connect_all	This is a connect-all operation that can connect to all the SSH or telnet based on the conn_type provided by the user in the input datafile.
disconnect_all	This is a disconnect-all operation that can disconnect all the SSH or telnet sessions based on the details provided by the user in the input datafile.

5.2 CLOUDSHELL ACTIONS PACKAGE

The CloudshellActions class has methods (keywords) related to actions that can be performed on a CloudShell system.

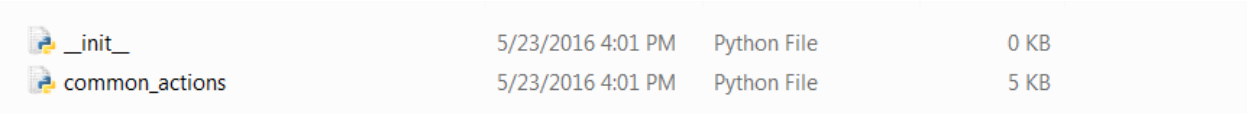


The following keywords are included in this package:

Keyword	Description
connect_to_cs	Logs in to API host using passed user credentials and domain
cs_logoff	To logoff from CloudShell
cs_create_reservation	Defines a reservation to be started immediately
cs_add_topology_to_reservation	Add Topology to reservation in CloudShell
cs_activate_topology	Activate Topology to reservation in CloudShell
cs_end_reservation	End the reservation in CloudShell
cs_add_users_to_reservation	Add one or more permitted users to the specified reservation.
cs_get_topo_details	To get the CloudShell topology details for a given path
cs_get_current_reservation	Retrieves current reservations for the specified owner
cs_get_reservation_details	Retrieves all details and parameters for a specified reservation
_create_cs_obj	Initializes the CloudShell object and logons to CloudShell returns the CloudShell object
_get_reservation_id	Fetch the reservation id from the xml response of get current reservation.

5.3 COMMON ACTIONS PACKAGE

The CommonActions package contains the keywords that are common to all products that are developed. The CommonActions class has methods (keywords) that are common for all the products.

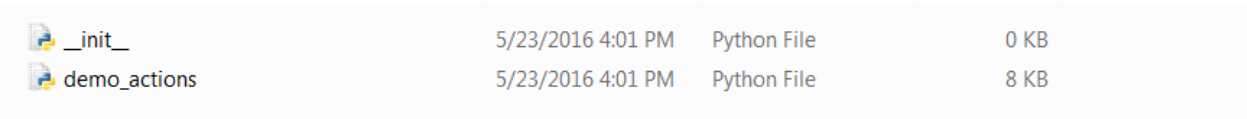


It contains the following keywords:

Keyword	Description
wait_for_timeout	waits (sleeps) for the time provided
get_system_type	Finds the system name in the datafile and returns the system type
verify_resp_data	Verify 'resp_pat' exist in the data repository
set_env_var	create a temporary environment variable the value will only stay for this run

5.4 DEMO ACTIONS PACKAGE:

The DemoActions package that has all demo test case related keywords. These are the keywords used in the Warrior Demo testcases. They are there for demo purposes only. The DemoActions class has methods (keywords) related to actions used in demo KW.

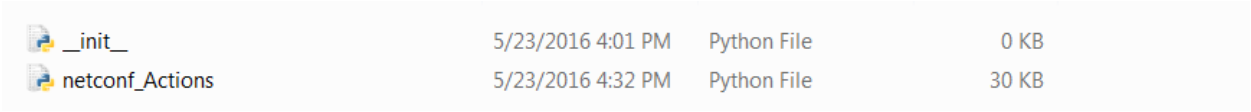


The following keywords are included in DemoActions:

Keyword	Description
check_lab_equipment	Call the pc_replacement or testset_calibration KW to validate lab PC or test set calibration are up-to-date or not.
pc_replacement	Verify lab PC is current if less than 4 years old, otherwise a replacement is required.
testset_calibration	Check if the test set calibration is current if less than 1 year old, otherwise, re-calibration is required.

5.5 NETCONF ACTIONS PACKAGE

The NetconfActions package contains all netconf related keywords. The NetconfActions class has methods (keywords) related to actions performed on any basic netconf interface.



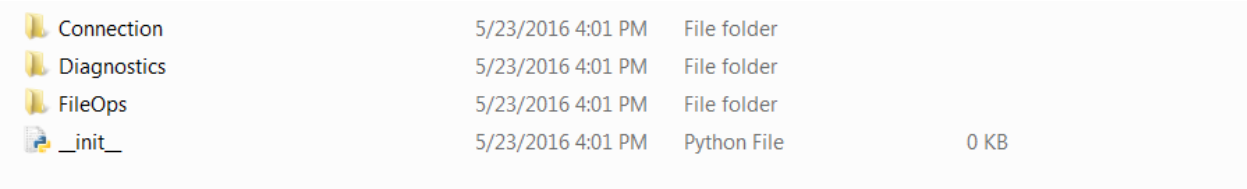
This package contains the following keywords:

Keyword	Description
connect_netconf	This is a generic connect that can connect to Netconf over SSH based system
close_netconf	Request graceful termination of the session and close the transport
get_config	Retrieve all or part of a specified configuration.
copy_config	Create or replace an entire configuration datastore with the contents of another complete configuration datastore
delete_config	Delete a configuration datastore
discard_changes	Revert the candidate configuration to the currently running configuration. Uncommitted changes will be discarded.
edit_config	Loads all or part of the specified config(from file) to the datastore
commit	Commit the candidate datastore as the device's new current configuration
lock	Lock the configuration system
unlock	Release the configuration lock
get	Retrieve operational state information.
kill_session	Force the termination of a NETCONF session (not the current one!)
validate	Validate the contents of the specified configuration.
request_rpc	Send RPC command
edit_config_from_string	Loads all or part of the specified config(not file) to the datastore
create_subscription	create-subscription to receive netconf event notification
waitfor_subscription	Wait for specified notification event

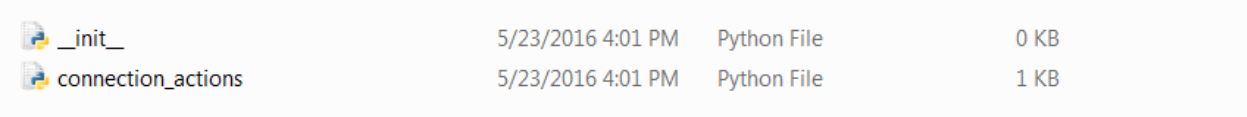
testfor_killsession	Kill-session test keyword create another session to same NE and kills it.

5.6 NETWORK ACTIONS PACKAGE

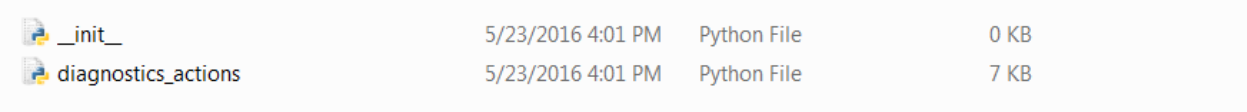
The NetworkActions package contains three subpackages – The Connection package, the Diagnostics package, and the FileOps package.



I. The Connections package contains the Connection keyword class, which currently does not have any Keywords in it.



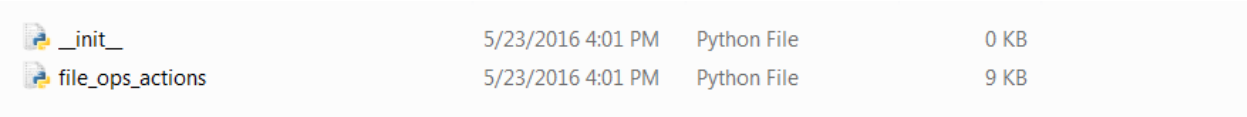
II. The Diagnostics package contains the Diagnostics keyword class which contains keywords related to Network Diagnostics Operations.



Currently, the Diagnostic package has two keywords:

Keyword	Description
ping_from_remotehost	This keyword will use connection session available in source system (provided in system name) and will ping from source system to destination system
traceroute_from_remotehost	This keyword will use connection session available in source system (provided in system name) and will execute traceroute from source system to destination system

III. The FileOps package contains the FileOpsActions class. This class contains the keywords related to the File Operations keyword class.

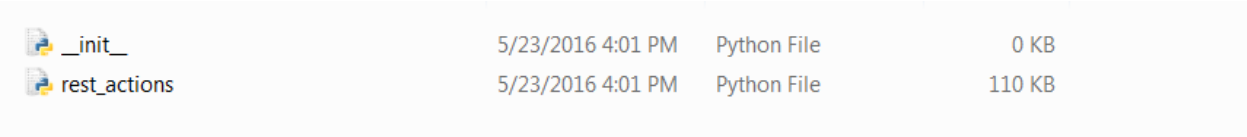


There are in total, two keywords in this class:

Keyword	Description
ftp_from_remotehost	FTP (test both put and get).This keyword can be used to transfer file from source _system to destination system or vice versa. It checks the size after transfer in both get and put.
sftp_from_remotehost	SFTP (test both put and get).This keyword can be used to transfer file from source _system to destination system or vice versa. It checks the size after transfer in both get and put.

5.7 REST ACTIONS PACKAGE

The RestActions package contains the RestActions class. This class has all the keywords related to REST operations.



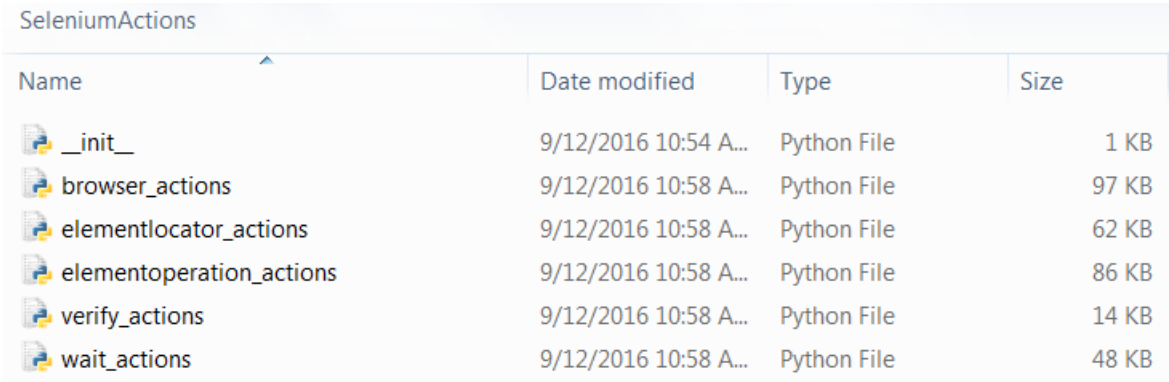
The REST Actions package contains the following keywords:

Keyword	Description
perform_http_post	Perform an http post actions and get the response. This keyword uses the warrior recommended Input datafile format for rest
perform_http_get	Perform an http get actions and get the response. This keyword uses the warrior recommended Input datafile format for rest
perform_http_put	Perform an http put actions and get the response. This keyword uses the warrior recommended Input datafile format for rest
perform_http_patch	Perform an http patch actions and get the response. This keyword uses the warrior recommended Input datafile format for rest

perform_http_delete	Perform an http delete actions and get the response. This keyword uses the warrior recommended Input datafile format for rest
perform_http_options	Perform an http options actions and get the response. This keyword uses the warrior recommended Input datafile format for rest
perform_http_head	Perform an http head actions and get the response. This keyword uses the warrior recommended Input datafile format for rest

5.8 SELENIUM ACTIONS PACKAGE

The SeleniumActions package contains the following files:



I. The browser_actions.py file contains all the keywords related to browser management.

Keyword	Description
browser_actions	This will launch a browser
browser_maximize	This will maximize the browser window.
navigate_to_url	This will navigate the browser tab to given URL.
navigate_forward	This will take you forward in the browser history.
navigate_backward	This will take you backward in browser history.
browser_refresh	This will refresh the browser window.
browser_reload	This will reload the browser window.
browser_close	This will close the browser window.
set_window_size	This will set the browser window to a particular size.
set_window_position	This will set the browser window to a particular position.
open_a_new_tab	This will open a new tab.
switch_between_tabs	This keyword will let you switch between all open tabs.

close_a_tab	This keyword will let you close an open tab.
get_window_size	This keyword will return window size.
get_window_position	This keyword will return the window position.
save_screenshot	This keyword will save a screenshot of the current browser window.
delete_cookies	This keyword will delete all cookies of a browser instance.
delete_a_cookie	This keyword will delete a particular cookie of a browser instance.

II. The elementlocator_actions.py file contains all the keywords related to locating elements on a web page.

Keyword	Description
get_element	This will get an element by the given locator
get_element_by_xpath	This will get an element by the element's xpath
get_element_by_id	This will get an element by the element's ID
get_element_by_selector	This will get an element by the element's CSS selector
get_element_by_link_text	This will get an element by the element's Link Text
get_element_by_partial_link_text	This will get an element by the element's Partial Link Text
get_element_by_tagname	This will get an element by the element's tag name
get_element_by_classname	This will get an element by the element's class name
get_element_by_name	This will get an element by the element's name

III. The elementoperations_actions.py file contains all the keywords related to performing actions on elements on a web page.

Keyword	Description
clear_text	This will clear text in the given element
click_an_element	This will simulate a click on the given element
clear	This will clear any text, checks on the given element
type_text	This will type text into a given element
fill_an_element	This will fill an element
send_keys_to_an_element	This will send keys like ENTER, COMMAND, F1 to the element
double_click_an_element	This will simulate a double-click on the given element
drag_and_drop_an_element	This will simulate a drag and drop on the given element

get_property_of_element	This will get property or attribute of the given element
check_property_of_element	This will get property or attribute of the given element

IV. The verify_actions.py file contains all the keywords related to verifications on elements.

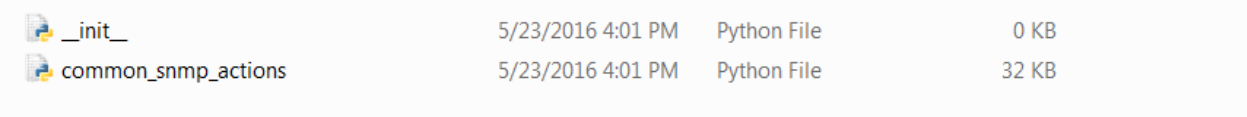
Keyword	Description
verify_page_by_property	This keyword will verify page by property.
verify_alert_is_present	This keyword will verify page by property.

V. The wait_actions.py file contains all wait related keywords.

Keyword	Description
set_implicit_wait	This keyword would permanently set the implicit wait time for given browser instance(s)
wait_until_element_is_clickable	This keyword would permanently set the implicit wait time for given browser instance(s)
wait_until_presence_of_element_located	This keyword would permanently set the implicit wait time for given browser instance(s)
wait_until_presence_of_all_elements_located	This keyword would permanently set the implicit wait time for given browser instance(s)
wait_until_visibility_is_determined	This keyword would permanently set the implicit wait time for given browser instance(s)
wait_until_visibility_of_element_located	This keyword would permanently set the implicit wait time for given browser instance(s)

5.9 SNMP ACTIONS PACKAGE

The SnmpActions package has the implementation of the standard SNMP protocol commands for SNMP v1 and v2c. It has the CommanSnmpActions class that contains all the keywords related to SNMP



There are seven SNMP keywords in this package:

Keyword	Description

snmp_get	snmp_get uses the SNMP GET request to query for information on a network entity
snmp_getnext	snmp_get_next uses the SNMP GETNEXT request to query for information on a network entity
snmp_walk	snmp_walk uses the SNMP WALK request to query for information on a network entity
snmp_set	snmp_set uses the SNMP SET request to query for information on a network entity
snmp_bulkget	snmp_bulkget uses the SNMP BULKGET request to query for information on a network entity
verify_trap	To verify the received trap in the SNMP manager
verify_snmp_action	Will Verify SNMP get/getnext/walk/getbulk actions.

6.0 UNDERSTANDING THE TESTCASE

A Warrior Testcase is an XML file written using specific tags that have been pre-defined by Warrior Framework. To correctly write a Warrior Testcase, understanding what each and every tag means and the correct structure of the testcase is very important. A testcase can be created using Katana. Please refer to Katana User Guide: Section 3 a complete guide to creating a testcase using Katana.

A simple testcase looks something like this:

```
<Testcase>
  <Details>
    <Name>Sample_Testcase</Name>
    <Title>Sample_Testcase</Title>
    <Engineer>Jo Smith</Engineer>
    <Date>2016-05-18</Date>
    <Time>11:12</Time>
    <State>New</State>
    <InputDataFile>../Data/Sample_Input_Data_file.xml</InputDataFile>
    <Datatype>Custom</Datatype>
    <default_onError action="next"/>
    <Logsdir/>
    <Resultsdir/>
    <ExpectedResults>PASS</ExpectedResults>
    <Category>Iterative</Category>
  </Details>
  <Steps>
    <step Driver="cli_driver" Keyword="connect" TS="1">
      <Arguments>
        <argument name="system_name" value="CLI[INT]"/>
        <argument name="ip_type" value="ip"/>
        <argument name="prompt" value=".*(%|#|\$)"/>
        <argument name="session_name" value="session5"/>
      </Arguments>
      <onError action="abort"/>
      <Description>Connect to the ssh/telnet port of the system</Description>
      <iteration_type type=""/>
      <Execute ExecType="Yes"/>
      <context>positive</context>
      <impact>impact</impact>
      <runmode type="rup" value="4"/>
    </step>
    <step Driver="cli_driver" Keyword="disconnect" TS="2">
      <Arguments>
        <argument name="system_name" value="CLI[INT]"/>
        <argument name="session_name" value="session5"/>
      </Arguments>
      <onError action="abort"/>
      <Description>
        Disconnects/Closes session established with the system/subsystem
      </Description>
      <iteration_type type=""/>
      <Execute ExecType="Yes"/>
      <context>positive</context>
      <impact>impact</impact>
      <runmode type="rmt" value="5"/>
    </step>
  </Steps>
  <Requirements>
    <Requirement/>
  </Requirements>
</Testcase>
```

Tags in a Testcase:

`<Testcase></Testcase>` forms the root tag of the testcase. This root tag helps Warrior identify this XML file as a Testcase. This tag has three child tags – Details, Steps, and Requirements.

1. The Details Tag

The `<Details></Details>` signifies that everything included inside this tag gives Warrior information about the Testcase. This tag has thirteen child tags – Name, Title, Engineer, Category, Date, Time, State, InputDataFile, Datatype, default_OnError, Logsdir, Resultsdir, and ExpectedResults.

1.1. The Name Tag

The tag holds the Testcase name. It must be unique. The name of the file and the text of this tag should match. This is a mandatory field.

1.2. The Title Tag

The title lets you add a title or a description for this testcase so that you can refer to it later on, to understand the purpose of this testcase. This is a mandatory field.

1.3. The Engineer Tag

This tag stores the Testcase developer's name. This is a mandatory field.

1.4. Category

Feature or Category. This field will be used to group testcases quickly. You can classify your testcase under any Category (like Sanity, or Alarm). This is an optional field, but it is highly recommended that you categorize your testcases.

1.5. The Date Tag

The Date tag stores the date of creation of this Testcase. This is an optional field, only defined just so that you can date stamp a Testcase.

1.6. The Time Tag

The Time tag stores the time of creation of this Testcase. This is an optional field, only defined just so that you can time stamp a Testcase. This is an optional field.

1.7. The State Tag

This tag contains the state of the Testcase, i.e., whether the Testcase is a new, released, test-assigned, or anything else that you might consider appropriate for the Testcase. This is an optional field.

1.8. The InputDataFile Tag

Location of the data file to be used by the keyword. If the testcase does not need to use a data file, then a No_Data is required. If this is left blank, Warrior will expect to find a data file named testcasename_data.xml in the Data directory of the Warriorspace.

1.9. The Datatype Tag

Determines the interaction of the testcase with the data file. If Iterative is selected, the testcase data file needs to use the system data file format. The steps of the testcase will run against the data in each system of the data file sequentially. If Custom is selected, the user can use a custom defined data file, with custom variables assigned to each keyword to access the input data file. The argument within a step, field will be used to access the data from the data file. If Hybrid is selected, each step gets run according to what has been specified against it. For more details on the Hybrid mode, please see “The iteration_type Tag”. The default is ‘Iterative’.

1.10. The default_OnError Tag

Default error handling for the testcase. This is where you set what happens if a testcase step fails. User can override the default by selecting the on error in the step section. If the step has an on error selection, that will be used instead of the default.

a. The Actions attribute

This is an attribute of the default_OnError. This is an optional attribute. If not set, Warrior will default to the “next”, that is, it would proceed to the next Testcase. You will be able to set different onError setting

per step if needed. That will be done at the step level. The options for this attribute are next, abort, and goto.

b. The Value attribute

This is an attribute of the default_OnError tag but it is necessary only if the action attribute is set to "goto". This attribute accepts value as a step number, that is, the number of the step that you want to jump to in case the current step throws an Error. Warrior will then directly go to executing the step corresponding to the number given if the current step throws an Error.

1.11. The Logsdir Tag

Location of directory where log files will be stored. If field is left blank Warrior will use the default location under the execution folder of the Warriorspace. This is therefore, an optional field.

1.12. The Resultsdir Tag

Location of directory where result files will be stored. If field is left blank, Warrior will use what is in the testcase, if that has been left blank, Warrior will use the location defined in the w_setting.py file, if that is also left blank, then Warrior will use the default under the Execution directory. This is, therefore, an optional field.

1.13. The ExpectedResults Tag

The ExpectedResults tag is for your benefit, so that you know if a particular testcase is supposed to Pass or Fail or Error out. This is an optional field.

2 The Requirements Tag

User can add as many requirements as needed. These requirements are for informational purposes only. The requirements entered in the test case will be shown in the xml result file.

2.1 The Requirement Tag

Every requirement needed for this testcase must be added inside a `<Requirement></Requirement>`. This is an optional field.

3 The Steps Tag

3.1 The Step Tag

User can add as many steps as needed but at least one step is required. Each step will have a Driver selection and a Keyword selection. User needs to indicate the product driver to be used in the step and the keyword.

3.1.1 The Arguments Tag

Section within the step where the arguments are detailed.

3.1.1.1 The Argument Tag

The argument tag is where you can pass arguments directly to the keywords through the Testcase. The name of the argument should be the value of the attribute 'name' in this tag, while the value to be passed for that argument should be the value of the 'value' attribute. The argument tag containing the system name only when the Custom mode is selected for the testcase.

3.1.2 The onError Tag

Set what happens if the step fails. This will override the default of the test case. Options are the same as the default on Error. This is an optional field.

3.1.3 The Description Tag

Text description for each step can be specified within this tag. This is an optional field but it is highly recommended that you fill it out.

3.1.4 The iteration_type Tag

This tag is required only when the Hybrid mode is selected. This can have three options – once_per_tc, once_per_iter, and standard. The 'standard' mode is the default where the step would run 'normally' – like how it would run, had it been an Iterative or Custom Datatype mode. The 'once_per_tc' runs that particular step only once per testcase while the 'once_per_iter' mode lets that particular step for every iteration that the set of steps go through.

3.1.5 The Execute Tag

This is an optional field that decides when a step should be executed. If this field is not set, the default will be to execute the keyword. The ExecType attribute has the options: If, If Not, Yes, No.

If: The keyword will be executed if the condition below is met, if the condition is not met, the Else action will be executed.

If Not: The keyword will be executed if the condition below is not met, if the condition is met, the Else action will be executed.

Yes: The Keyword will be executed without any conditions.

No: The keyword will not be executed.

<Rule> The condition is set in this element.

3.1.6 The Context Tag

Indicates if the step is a positive or negative test scenario. The default is 'positive'.

3.1.7 The Impact Tag

Used to decide if status of step will or will not impact the test case status. Options are Impact and noimpact with impact being the default.

3.1.8 The Runmode Tag

This tag can be filled up to run a testcase multiple times, until failure, or until pass. This tag has two attributes, 'type' and 'value'. In type, you can specify either 'RMT', 'RUF', or 'RUP', and in 'value', you can specify the maximum number of times you want the step to run. This is an optional tag.

7.0 UNDERSTANDING THE SUITE

A suite is a collection of testcase with execution instructions. A Warrior Testsuite is an XML file written using specific tags that have been pre-defined by Warrior Framework. To correctly write a Warrior Testsuite, understanding what each and every tag means and the correct structure of the testsuite is very important. A testsuite can be created using Katana. Please refer to Katana User Guide: Section 4 a complete guide to creating a testsuite using Katana.

A typical testsuite looks something like this:

```
▼<TestSuite>
  ▼<Details>
     <Name>Sample_Suite</Name>
     <Title>Sample_Suite</Title>
     <Engineer>Jo Smith</Engineer>
     <Date>05/17/2016</Date>
     <Time>13:33:58</Time>
     <type exectype="sequential_testcases"/>
     <State>New</State>
     <default_onError action="next"/>
     <Resultsdir/>
     <IDF>../Data/Sample_Input_Data_File.xml</IDF>
   </Details>
  ▼<Requirements>
     <Requirement/>
   </Requirements>
  ▼<Testcases>
    ▼<Testcase>
       <path>../Testcases/Sample_Testacase.xml</path>
       <context>positive</context>
       <runtype>sequential_keywords</runtype>
       <onError action="goto" value="2"/>
       <runmode type="rmt" value="2"/>
       <impact>impact</impact>
     </Testcase>
    ▼<Testcase>
       <path>../Testcases/demo_tc2.xml</path>
       <context>positive</context>
       <runtype>sequential_keywords</runtype>
       <onError action="next" value=""/>
       <runmode type="ruf" value="5"/>
       <impact>impact</impact>
     </Testcase>
   </Testcases>
 </TestSuite>
```

Tags in a Testsuite:

`<Testsuite></Testsuite>` forms the root tag of the testsuite. This root tag helps Warrior identify this XML file as a Testsuite. This tag has three child tags – Details, Testcases, and Requirements.

1.0 The Details Tag

The <Details></Details> signifies that everything included inside this tag gives Warrior information about the Testsuite. This tag has thirteen child tags – Name, Title, Engineer, Date, Time, Type, State, InputDataFile, Datatype, default_OnError, Logsdir, Resultsdir, and ExpectedResults.

1.1 The Name Tag

The tag holds the Testsuite name. It must be unique. The name of the file and the text of this tag should match. This is a mandatory field.

1.2 The Title Tag

The title lets you add a title or a description for this testcase so that you can refer to it later on, to understand the purpose of this testcase. This is a mandatory field.

1.3 The Engineer Tag

This tag stores the Testcase developer's name. This is a mandatory field.

1.4 The Date Tag

The Date tag stores the date of creation of this Testsuite. This is an optional field, only defined just so that you can date stamp a Testsuite.

1.5 The Time Tag

The Time tag stores the time of creation of this Testsuite. This is an optional field, only defined just so that you can time stamp a Testsuite.

1.6 The Type Tag

The exectype attribute is used to determine how the test cases within the suite should be executed. The options are 'Parallel, Sequential, Run_Multiple_Time, Run_Until_Fail, Run_Until_Pass.' The

Max_Attempts attributes is used with Run_Multiple, Run_Until_Fail, and Run_Until_Pass to determine how many times to run the suite or what is the max attempts value if a test case does not fail. The default exectype is sequential.

The Parallel option indicates that all the testcases within the suite are to be run in parallel, all at the same time.

The Sequential option indicates that all the testcases within the suite are to be run in sequence, one after the other.

The Run Multiple option indicates that the sequence of test cases (i.e. suite) will be executed as many times as indicated in the max attempts attribute.

The Run until Fail option indicates that the sequence of test cases (i.e. suite) will be executed until the suite Fails or the max attempts have been reached.

The Run until Pass option indicates that the sequence of test cases (i.e. suite) will be executed until the suite Pass or the max attempts have been reached.

1.7 The State Tag

This tag contains the state of the Suite, i.e., whether the Suite is a new, released, test-assigned, or anything else that you might consider appropriate for the Testcase. This is an optional field.

1.8 The default_OnError Tag

Default error handling for the suite. This is where you set what happens if a testcase fails. User can override the default by selecting the on error in the testcase section. If the testcase has an on error selection, that will be used instead of the default.

1.8.1 The Action attribute

This is an attribute of the default_OnError. This is an optional attribute. If not set, Warrior will default to the “next”, that is, it would proceed to the next testcase. You will be able to set different onError setting per testcase if needed. That will be done at the testcase level. The options for this attribute are next, abort, and goto.

1.8.2 The Value attribute

This is an attribute of the default_OnError tag but it is necessary only if the action attribute is set to "goto". This attribute accepts value as a testcase number, that is, the number of the testcase that you want to jump to in case the current testcase throws an Error. Warrior will then directly go to executing the testcase corresponding to the number given if the current testcase throws an Error.

1.9 The Resultsdir Tag

Location of directory where result files will be stored here. If field is left blank Warrior will use what is in the testcase, if that is left blank then, Warrior will use the location defined in the w_setting.py file, if that is also left blank, then Warrior will use the default under the Execution directory. This is, therefore, an optional field.

1.10 The IDF Tag

Path to Data file associated with the test suite. If a data file is listed, that data file will overwrite the data file used in the test cases in the suite. If a data file is not listed, the data files within the test cases will be used. This is therefore, an optional field.

2.0 The Requirements Tag

User can add as many requirements as needed.

2.1 The Requirement Tag

Every requirement needed for this testcase must be added inside a `<Requirement></Requirement>`. This is an optional field.

3.0 The Testcases Tag

User can add as many testcases as needed but at least one testcase is required.

3.1 The Path Tag

This tag is for the location of the testcase. You can enter a relative path. This is a mandatory field.

3.2 The Context Tag

The context of each testcase can be either positive or negative. This determines if the testcase is expected to Pass or Fail. The default is 'positive'.

3.3 The runtype Tag

This runtype tag determines if the sequential_keywords, or parallel_keywords. Default is sequential keywords. If sequential keywords is selected, the keywords (or steps) within the test case will all run in sequence. If the parallel_keywords is selected, then the keywords (or steps) within the test case will all run in parallel.

3.4 The onError Tag

Set what happens if the testcase fails. This will override the default of the testsuite. Options are the same as the default on Error. The default is 'next'.

3.5 The Runmode Tag

This tag has two attributes - "type" and "value". The value provided to the type attribute indicates whether this particular testcase should run multiple times (RMT), run until failure (RUF), or, run until pass (RUP). The attribute value tales in the number of attempts that this testcase should go through before completing the execution of this testcase – in case of RUP and RUF, the value given in this attribute would be treated as the maximum number of attempts that Warrior would execute this testcase till it passes/fails, while for RMT, Warrior would run the testcase for as many times as indicated. This is an optional field.

3.6 The impact Tag

Used to decide if status of test case will or will not impact the test suite status. Options are Impact and noimpact with impact being the default.

8.0 UNDERSTANDING THE PROJECT

A Project is a collection of testsuites. A Warrior Project is an XML file written using specific tags that have been pre-defined by Warrior Framework. To correctly write a Warrior Project, understanding what each and every tag means and the correct structure of the project is very important. A project can be created using Katana. Please refer to Katana User Guide: Section 5 a complete guide to creating a project using Katana.

A typical Project looks something like this:

```
▼<Project>
  ▼<Details>
     <Name>Sample_Project</Name>
     <Title>Sample_Project</Title>
     <Engineer>Jo Smith</Engineer>
     <State>New</State>
     <Date>05/17/2016</Date>
     <Time>14:19:30</Time>
     <default_onError action="next"/>
   </Details>
  ▼<Testsuites>
    ▼<Testsuite>
       <path>../Suites/Sample_Suite.xml</path>
       <onError action="next" value=""/>
       <impact>impact</impact>
     </Testsuite>
    ▼<Testsuite>
       <path>../Suites/Suite_Template.xml</path>
       <onError action="next" value=""/>
       <impact>impact</impact>
     </Testsuite>
    ▼<Testsuite>
       <path>../Suites/demo_suite.xml</path>
       <onError action="next" value=""/>
       <impact>impact</impact>
     </Testsuite>
   </Testsuites>
</Project>
```

A project is a collection of suites with execution instructions. A project template and a sample project are located in Warriorspace/projects directory. The root node is `<Project></Project>`. This root node has two children – the `<Details></Details>` tag, and the `<Testsuite></Testsuite>` tag

1.0 The Details Tag

This tag contains all the details about the Project. The following are its child tags:

1.1 The Name Tag

This tag contains the project name. It must be unique. This name should match the file name. This is a mandatory tag.

1.2 The Title Tag

This is the project title or description. You can describe what this project does and what all kind of suites it contains in this tag. This is also a mandatory tag.

1.3 The Engineer Tag

This tag contains the name of the Project designer. This is a mandatory tag.

1.4 The State Tag

This tag contains the state of the Project, i.e., whether the Project is a new, released, test-assigned, or anything else that you might consider appropriate for the Testcase. This is an optional field.

1.5 The Date Tag

The Date tag stores the date of creation of this Project. This is an optional field, only defined just so that you can date stamp a Project.

1.6 The Time Tag

The Time tag stores the time of creation of this Project. This is an optional field, only defined just so that you can time stamp a Project.

1.7 The default_OnError Tag

This is the default behavior if any of the suite fails. User can override the default by selecting the on error in the testsuite section. If the testsuite has an on error selection, that will be used instead of the default.

1.7.1 The Actions attribute

This is an attribute of the default_OnError. This is an optional attribute. If not set, Warrior will default to the "next", that is, it would proceed to the next testsuite. You will be able to set different onError setting per testsuite if needed. That will be done at the testsuite level. The options for this attribute are next, abort, and goto.

1.7.2 The Value attribute

This is an attribute of the default_OnError tag but it is necessary only if the action attribute is set to "goto". This attribute accepts value as a testsuite number, that is, the number of the testsuite that you want to

jump to in case the current testsuite throws an Error. Warrior will then directly go to executing the testsuite corresponding to the number given if the current testsuite throws an Error.

2.0 The Testsuite Tag

You can add as many testsuites you want in a project, but at least one testsuite is required.

2.1 The Path Tag

This tag contains information about the location of the testsuite. You can enter a relative path. This is a mandatory field.

2.2 The onError Tag

Set what happens if the testsuite fails. This will override the default of the project. Options are the same as the default on Error. This is an optional field.

2.3 The Impact Tag

This tag will determine if the failure of this test suite will impact the overall pass/fail status of the project. Setting it to impact will cause the project to fail if the suite fails. Setting it to no impact will not cause the project to fail if the suite fails. The default is 'impact'.

9.0 UNDERSTANDING THE INPUT DATA FILE (IDF)

A typical datafile looks something like this:

```
<credentials>
  <system name="http_bin_1">
    <url>http://httpbin.org/post</url>
    <cookies>{"cookie_name": "this_cookie"}</cookies>
  </system>
  <system name="http_bin">
    <subsystem name="bin_1">
      <url>http://httpbin.org/put</url>
      <user>Jo</user>
      <password>536ett</password>
      <content_type>json</content_type>
      <expected_response>200</expected_response>
    </subsystem>
    <subsystem name="bin_2">
      <url>http://httpbin.org/delete</url>
      <content_type>json</content_type>
      <expected_response>500</expected_response>
    </subsystem>
    <subsystem name="bin_3">
      <url>http://httpbin.org/patch</url>
      <expected_response>200</expected_response>
      <user>Jo</user>
      <password>536ett</password>
    </subsystem>
  </system>
  <system name="http_bin_3">
    <url>http://httpbin.org/get</url>
    <user>Jo</user>
    <password>536ett</password>
    <content_type>json</content_type>
    <expected_response>200</expected_response>
    <params>{'key1': 'value1', 'key2': ['value2', 'value3']}</params>
  </system>
</credentials>
```

This is a system based format that is recommended by Warrior. This format is required for most of the built-in keywords and for the iterative and hybrid mode.

Here, the root of the data file is `<credentials></credentials>`. The system-based format requires all the child nodes of the `<credentials></credentials>` to be `<system></system>`. Now, all the information that is required can be filled inside the `<system></system>`.

A `<system></system>` can have multiple subsystems inside it. These subsystems should be added in as `<subsystem></subsystem>` child tags under the `<system></system>` tags and the corresponding information requited should be added as child tags to the `<subsystem></subsystem>` tags. This can also be added as attribute name and value in the `<subsystem></subsystem>` tags.

This data file is provided as the input datafile in Warrior Testcase.

This data files supports the following

- System with data for the system
- A system with subsystem with data only for the subsystems.

The case of system with subsystem with data for both system and subsystem is NOT supported. In such a case user should create the data for the system as a separate subsystem within the system. Substituting values form environment variables:

Substituting values from environment variables is supported in input datafile

To reference the environment variables use the pattern `${ENV.variable_name}` where variable_name is the name of the variable in the environment settings of the Operating System

E.g. `${ENV.IPADDR}` will be replaced with the value of `IPADDR` variable in the environment settings

To create a datafile in Katana, you can refer to Katana User Guide: Section 6 of this document.

10.0 THE DATA REPOSITORY

Warrior provides a data repository within a test case, test suite, and project. The data repository within a test case contains all the data returned by a keyword stored as dictionary entry. The data returned by the keywords is stored automatically in the data repository and is accessible to the other keywords within the test case.

Warrior provides a data repository within a test suite as well. The data repository within a test suite contains all the testcase statuses. The status of each executed testcase in that suite is available for every other testcase to access.

Similarly, Warrior provides a data repository within a project. The data repository within a project contains all the suite statuses. The status of each executed suite in that project is available for every other suite to access.

Warrior Framework expects all data returned by a keyword should be returned in a python dictionary. This returned dictionary will be stored in the test case, testsuite, or project data repository as a key-value pair. Keywords will have to access the testcase data repository to get the data returned from a previous keyword, testcases in a suite will have access to testcase statuses returned from the previous testcases, and suites in a project will have access to suite statuses returned from the previous suites.

If a previous keyword has returned `api_resp (string)`, `session_id (object)` to the testcase data repository. In order to use these values in the next keyword you would have to access that in the keyword using this function:

```
Utils.data_Utils.get_object_from_datarepository
```

And use it in the keyword as:

```
api_resp = Utils.data_Utils.get_object_from_datarepository('api_resp')
session_id = Utils.data_Utils.get_object_from_datarepository('session_id')
```

11.0 TESTCASE DESIGN FEATURES

11.1 RUNNING KEYWORDS IN SEQUENCE

Warrior keywords defined within a test case will run sequentially by default. When executing a testcase in Warrior, the keywords will execute sequentially.

11.2 RUNNING KEYWORDS IN PARALLEL

Warrior keywords defined within a test case can be executed in parallel. Follow the steps below to run the keywords in parallel:

1. Create a test case to contain the keywords that need to be run in parallel.
2. Create a test suite or modify an existing one.
3. In the test case block of the Suite, add the test case created in step 1
4. Set the runtype tag to: `<runtype>parallel_keywords</runtype>`

11.3 DESIGNING A NEGATIVE KEYWORD

Negative keywords are keywords that are expected to fail. If a negative keyword fails, the keyword will be marked as a Pass in the test case results. All keywords in Warrior are positive keywords by default unless marked as negative. Follow the steps below to mark a keyword as a negative one:

1. Create a test case to contain the keywords to be executed.
2. After selecting the driver and keyword, set the context tag to Negative. `<context>negative</context>`
3. If the context tag is not present, Warrior will default keyword's context to be positive.

11.4 KEYWORDS DESCRIPTION

While creating a keyword, a user has the option of setting a WDesc variable within the keyword that would be set to a string describing the keyword behavior. The string assigned to WDesc will be read by Warrior at execution and displayed with the keyword name in the test case xml result file. Katana, Warrior's Web based test case creation tool will also read the WDesc string and display it in the step description field.

11.5 CONDITIONAL KEYWORDS EXECUTION

Warrior allows the user to set conditions for each keyword's execution. The user will be able to set the execution condition to always run, to never run, to run if a condition is met, or to run if a condition is not met. Follow the procedure below to set the execution conditions:

1. Create a test case to contain the keywords to be executed.

2. After selecting the driver and keyword, set the execute tag for each keyword

   a. `<Execute ExecType='If, If Not, Yes, No' > <Rule Condition= 'variable' Condvalue='1' Else='next, abort, goto' Elsevalue='2'/> </Execute>`

   b. Execute type : Yes = always run , No= to never run, If = run if a condition is met, If Not = to run if a condition is not met

   c. Rule Condition: The variable being evaluated for the condition.

   d. Condvalue: The value expected for the condition to be met.

   e. Else: Action to be taken if condition is not met

   f. Elsevalue: Step number in case Else was goto.

11.6 DESIGNING A NEGATIVE TESTCASE

Negative test cases are test cases that are expected to fail. If a negative test case fails, the test cases will be marked as a Pass in results. All test cases in Warrior are positive test cases by default unless marked as negative. Follow the steps below to mark a test case as a negative one:

1. Create a test suite to contain the test cases to be executed.

2. After selecting the test case path, set the context tag to Negative. `<context>negative</context>`

3. If the context tag is not present, Warrior will default test case context to be positive.

11.7 RUNNING A TESTCASE IN SEQUENCE

Warrior test cases defined within a test suite can be executed in parallel. Follow the steps below to run the test case in parallel:

1. Create the test case that you would like to run in parallel.

2. Create a test suite or modify an existing one.

3. Set the type tag in the test suite : `<type exectype ='Sequential'/>`

4. In the test case block of the Suite, add the test case created in step 1

5. Execute the test suite. The test cases will run in sequence.

11.8 RUNNING A TESTCASE IN PARALLEL

Warrior test cases defined within a test suite can be executed in parallel. Follow the steps below to run the test case in parallel:

1. Create the test case that you would like to run in parallel.

2. Create a test suite or modify an existing one.

3. Set the type tag in the test suite : `<type exectype ='Parallel'/>`

4. In the test case block of the Suite, add the test case created in step 1

5. Execute the test suite. The test cases will run in parallel.

11.9 RUNNING A TESTCASE MULTIPLE TIMES

Warrior test cases defined within a test suite can be executed Multiple Time. Follow the steps below to run the test case/s multiple times:

1. Create a test suite or modify an existing one.

2. Set the type tag in the test suite : `<type exectype = 'Run_Multiple', Max_Attempts='5' />`

3. Execute the test suite. The test cases in the suite will run multiple times as indicated in the Max_Attempts attribute.

11.10 RUNNING A TESTCASE UNTIL FAILURE/PASS

Warrior test cases defined within a test suite can be executed until failure. Follow the steps below to run the test case/s multiple times until failure:

1. Create a test suite or modify an existing one.

2. Set the type tag in the test suite : `<type exectype = 'Run_Until_Failure, Max_Attempts='5' />`. If you want to run the suite until it passes, set the type tag in the test suite : `<type exectype = 'Run_Until_Pass, Max_Attempts='5' />`

3. Execute the test suite. The test cases in the suite will run until the suite fails or the suite is executed as indicated in the Max_Attempts attribute.

11.11 ERROR HANDLING IN WARRIOR

Warrior allows error handling to be handled at the keyword, testcase, testsuite, and project level. The error handling can be set to instruct Warrior what to do in case of a failure. The options are to proceed to the next step, test case or suite, abort execution or go to another test case, step or suite. If the error handling instruction is not present, Warrior will default to proceed to next step/test case/suite as applicable.

1. Test case Error Handling: User can set a default error handling instruction for the entire test case using the default_OnError tag. `<default_OnError action = 'goto' value='9' />`. If this tag is not present, Warrior will default to proceed to next step.

2. Step Error Handling: When an error is encountered during execution of a keyword, the keyword will fail. User has the option of indicating what to do in case of a keyword failure on a step by step basis. Each step can have its own error handling. Use the following tag to set the step error handling : `<onError Action='next, abort, goto' Value='9' />`

a. The step error handling will supersede the test case default error handling.

b. If step error handling is not defined, the default error handling will be used. If the default error handling is not set as well, Warrior will default to go to next step in case of a step failure.

3. Test Suite Error Handling: User can set a default error handling instruction for the entire test suite using the default_OnError tag. `<default_OnError action = 'goto' value='9' />`. If this tag is not present, Warrior will default to proceed to next test case.

a. Within the suite, a user has the option of indicating what to do in case of a test case failure on a test case by test case basis. Each test case can have its own error handling. Use the following tag to set the test case error handling : `<onError Action='next, abort, goto' Value='9' />`

b. The test case error handling will supersede the test suite default error handling.

c. If test case error handling is not defined, the default error handling will be used. If the default error handling for the suite is not set as well, Warrior will default to go to next test case in case of a test case failure.

4. Project Error Handling: User can set a default error handling instruction for the entire Project using the default_OnError tag. `<default_OnError action = 'goto' value='9' />`. If this tag is not present, Warrior will default to proceed to next test suite.

a. Within the project, a user has the option of indicating what to do in case of a test suite failure on a test suite by test suite basis. Each test suite can have its own error handling. Use the following tag to set the test suite error handling : `<onError Action='next, abort, goto' Value='9'/>`

b. The test suite error handling will supersede the project default error handling.

c. If test suite error handling is not defined, the default error handling for the project will be used. If the default error handling is not set as well, Warrior will default to go to next test suite in case of a test suite failure.

12.0 HOW TO CREATE KEYWORDS

Keywords in Warrior are created in Action files that are imported by the product drivers. If you are creating a new keyword file, do the following:

- Create a directory under the Actions directory. Use the following naming convention: NameofyourchoiceActions
- Inside the directory newly created, create your python file following the naming convention: nameofyourchoice_Actions.py

Warrior uses standard Python programming for its keyword development, however Warrior requires users to adhere to the following rules:

1. The keywords can be methods of a class or they can be an independent function in a library
2. When using classes for keywords:
    - Users should always use the default __init__ for a class as shown below, and can extend it to add more attributes within __init__
    - Multiple classes are allowed in a single file
    - Class inheritance is not supported for keyword classes
    - In Warrior, method names are keywords and hence the method names should be unique within the driver's packages.
    - Users are allowed to use a mix of classes and independent functions in a single actions file as long as their names are unique.
    - Static methods with decorators are not currently supported by the Warrior framework for the Action classes.
    - Each keyword (along with its driver) is a step in the warrior test case. A keyword can have multiple sub-steps that are implemented in the code.

Sample Keyword Class:

```
""" This is the actions file, keywords are programmed here """
""" Import any python library you want to use below this line """

import time

""" Import Warrior Framework Utilities below this line"""
""" As a recommendation always use the below three lines in every actions
file"""

import Framework
import Framework.Utils as Utils
from Framework.Utils.print_Utils import print_info, print_debug,
print_warning, print_error


""" WARRIOR KEYWORD TEMPLATE AND RULES """

""" Warrior uses standard Python programming for its keyword development,
however Warrior requires the users to adhere to the following rules:

Supported Keyword types

The keywords can be methods of a class or they can be a independent function
in a library

When using classes for keywords:
Users should always use the default __init__  for a class as shown below,
and can extend it to add more attributes within __init__
Multiple classes are allowed in a single file
Class inheritance is not supported for keyword classes
In Warrior, method names are keywords and hence the method names should be
unique within the driver's packages.
Users are allowed to use a mix of classes and independent functions in a
single actions file as long as their names are unique.
Static methods with decorators are not currently supported by the Warrior
framework for the Action classes.

Each keyword (along with its driver) is a step in the warrior test case. A
keyword can have multiple sub-steps that are implemented in the code """


class CliActions(object):

    """
    Default __init__ field must be used when using classes for keywords.
    """
    def   init  (self):
        self.resultfile = Utils.config_Utils.resultfile
        self.datafile = Utils.config_Utils.datafile
        self.logsdir = Utils.config_Utils.logsdir
        self.filename = Utils.config_Utils.filename
```

```
        self.logfile = Utils.config_Utils.logfile

""" Sample method (keyword) with a single substep and some of the most
commonly used functions from the data_Utils is given below.

*** Please remember this is only a sample keyword to help a beginner in
Warrior Framework, and may not work exactly ****

"""

    def connect_ssh(self, system_name, session_name=None,
    expected_prompt='.*'):

        """
        KEYWORD DOCUMENTATION: A recommended style for keyword documentation
        is given below


        Connects to the ssh port of the the given system and creates a
        pexpect session object for the system

        :Arguments:
1. system_name(string) = Name of the system from the input
                datafile
            2. session_name(string) = name of the session to the system
            3. expected_prompt(string) = prompt expected in the terminal

        :Returns:
            1. status(bool)= True / False
            2. session_id (dict element)= key, value

            key = if session_name is not none = system_name_session_name,
            if session_name is none = system_name

            value = a pexpect session object for the system.

        """

        """ THE DESCRIPTION (optional):

        All keywords can have a description variable that gives details of
        what is being performed by the keyword
        It should be a string enclosed within double or single quotes, no
        variable substitutions to be used here as it is used as just a text
        for reporting purpose. This string will be used in result file for
        reporting a step. If this variable is missing warrior will just
        report  the keyword name in step description of the result file.

        """

        wdesc = "Connect to the ssh port of the system and creates a session
                 and return it if successful"

        """
        PSUBSTEP (optional) :

        All keywords can start with the below function-call to pSubStep with
```

```
        a single argument describing in short the sub-step's functionality
        If this function-call is missing warrior will not report any substep
        in the test case result file"""

        Utils.testcase_Utils.pSubStep('Connect to ssh of
                                       {0}'.format(system_name))


"""
SOME USEFUL Information:

Warrior provides a data repository within a testcase and a test suite. The
data repository within a testcase contains all the data returned by a
keyword stored as dictionary entry. The data returned by the keywords is
stored automatically in the data repository and is accessible to the other
keywords within the testcase. The process by which keywords can access the
data returned by other keywords is described in item #4 below.

Warrior also provides a data repository within a test suite. The data
repository within a testsuite contains all the data returned by the keywords
of the testcases within the suite stored as dictionary entry. The data
returned by the keywords is stored automatically in the data repository and
is accessible to the other keywords within the test suite. The process by
which keywords can access the data returned by other keywords is described
in item #4 below.

1. Custom Logfile for a Keyword:
--------------------------------
Warrior framework creates a logfile by default with the name of the testcase
located in Warriorspace\Execution\DateTime\logs.

If the keyword is written as class method then it is available to the
keyword as self.logfile.

If the keyword is written as an independent function then it is available
for the user in the test case data repository.

However user may want to create a separate logfile for each keyword,
to create a custom logfile for a keyword use the below function from
file_Utils

logfile = Utils.file_Utils.getCustomLogFile(self.filename,
self.logsdir,'NE_TL1_{0}'.format(system_name))


2. Adding Notes to test case result file:
-----------------------------------------
Use testcase_Utils.pNote to add notes to the result xml file refer to
testcase_Utils.py for more functions related to testcase result file
reporting

Utils.testcase_Utils.pNote("Logfile= %s" % logfile)


3. Get details from input datafile:
-----------------------------------
```

```
Warrior supports the use of a system input datafile. A sample system data
file is located in Warriorspace\Data. The get_credentials method from
data_Utils can be used to retrieve data from an xml tag in the system file.

For eg: In-order to get the ip, ssh_port, username, password, prompt of a
system called 'dpoe03' from the input datafile.

credentials = Utils.data_Utils.get_credentials(self.datafile, 'dpoe03',
['ip', 'ssh_port', 'username', 'password', 'prompt'])


4. Sharing data between keywords:
-----------------------------------
In Warrior Framework all data to be returned by a keyword should be returned
in a python dictionary, and this data will be stored in a the test case or
test suite data repository as a key-value pair. Keywords will have to access
the test case data repository to get the data returned from a previous
keyword.

For eg:
Let's say a previous keyword has returned ne_location(string),
session_id(pexpect object) to the test case data repository,order to use
these values in the next keyword use
'Utils.data_Utils.get_object_from_datarepository' and pass the key to it.

ne_location = Utils.data_Utils.get_object_from_datarepository('ne_location')
session_id = Utils.data_Utils.get_object_from_datarepository('session_id')


***Sample code of the keyword***

"""

        wdesc = "Connect to the ssh port of the system and creates a session
                 and return it if successful"

        Utils.testcase_Utils.pSubStep('Connect to ssh of
                                       {0}'.format(system_name))

        output_dict = {}
        logfile = Utils.file_Utils.getCustomLogFile(self.filename,
                  self.logsdir, 'NE_TL1_{0}'.format(system_name))
                  Utils.testcase_Utils.pNote(system_name)
                  Utils.testcase_Utils.pNote(logfile)
                  Utils.testcase_Utils.pNote(Utils.file_Utils.getDateTime())

        if session_name is None:
            session_id = system_name
        elif session_name is not None:
            session_id = system_name + session_name

        credentials = Utils.data_Utils.get_credentials(self.datafile,
                      system_name, ['ip', 'ssh_port', 'username',
                      'password', 'prompt'])
        ne_location =
        Utils.data_Utils.get_object_from_datarepository('ne_location')
```

```
        if credentials is False:
            status = credentials
        else:
            if credentials['prompt'] is not None:
                expected_prompt = credentials['prompt']
                session_object =
                Utils.cli_Utils.connectSSH(credentials['ip'],
                credentials['ssh_port'], credentials['username'],
                credentials['password'], logfile,
                prompt_expected=expected_prompt)

                if type(session_object) is pexpect.spawn:
                    output_dict[session_id] = session_object
                    status = True
                else:
                    status = False

        """
        Reporting status of a substep:
        ------------------------------
        After executing a substep in a keyword, use
        Utils.testcase_Utils.report_substep_status(status) to report the
        status of that substepto the result file.

        If a keyword has multiple sub-steps use this for each substep in the
        keyword.

        """

        Utils.testcase_Utils.report_substep_status(status)


        """
        Returning Keyword status and data:
        ----------------------------------
        Each keyword should return the status of the keyword in a return
        command in the keyword function.

        Any Data generated in the keyword can also be returned as a key
        value pair in a python dictionary.

        This key-value pair will be stored in the  data repository for use
        by other keywords.

        """

        return status, output_dict

        """
        RETURN TYPES

        1. Supported return types from keywords.
            a. Only status: True or False only
            b. Only dictionary.
            c. status, dictionary
            d. dictionary, status
        2. If a keyword does not return a status, it will be declared as
```

```
           failure in the testcase results.
        3. When a keyword returns a dictionary the testcase-datarepository
           will be updated with the dictionary returned by the keyword.
        4. If a keyword returns any unsupported value the keyword will be
           Declared as failure.

        *** Sample Return statement ***
        """

        return status, output_dict
```

13.0 HOW TO CREATE A TESTCASE

A Warrior Testcase is an XML file written using specific tags that have been pre-defined by Warrior Framework. To correctly write a Warrior Testcase, understanding what each tag means and the correct structure of the testcase is very important. To understand the significance each tag, please refer to section 6.0 of this User Guide.

You can use the Warrior Tool – Katana to create a new testcase or, you can create a testcase directly. To create a create case without Katana, you should:

1. Open an xml editor – like Notepad++, Gedit, or any other that is installed on your system

2. Start creating tags – for reference, you can refer to the final figure in Katana User Guide: Section 3 which shows a complete testcase, or you can follow the Katana steps (Katana User Guide: Section 3) and go on creating the tags and adding their respective values in your editor. Be sure that you close every tag that you open and that every attribute value is enclosed in double quotation marks. You can refer to -

http://www.xmlobjective.com/category/guide/

to better understand how to create a valid xml file.

3. Once you are satisfied with the Testcase that you have created, it is recommended that you run this file through Ironclaw (Section 3.3) so that you are sure that this is a valid Testcase.

14.0 HOW TO CREATE A SUITE

A Warrior Suite is an XML file written using specific tags that have been pre-defined by Warrior Framework. To correctly write a Warrior Suite, understanding what each tag means and the correct structure of the Suite is very important. To understand the significance each tag, please refer to Section 7.0 of this User Guide.

You can use the Warrior Tool – Katana to create a new testsuite or, you can create a testsuite directly. To create a create case without Katana, you should:

1. Open an xml editor – like Notepad++, Gedit, or any other that is installed on your system

2. Start creating tags – for reference, you can refer to the final figure in Katana User Guide: Section 4 which shows a complete testsuite, or you can follow the Katana steps (Katana User Guide: Section 4) and go on creating the tags and adding their respective values in your editor. Be sure that you close every tag that you open and that every attribute value is enclosed in double quotation marks. To better understand how to create a valid xml file, you can refer to :

http://www.xmlobjective.com/category/guide/

3. Once you are satisfied with the testsuite that you have created, it is recommended that you run this file through Ironclaw (Section 3.3) so that you are sure that this is a valid testsuite.

15.0 HOW TO CREATE A PROJECT

A Warrior Project is an XML file written using specific tags that have been pre-defined by Warrior Framework. To correctly write a Warrior Project, understanding what each tag means and the correct structure of the Project is very important. To understand the significance each tag, please refer to section 8.0 of this User Guide.

You can use the Warrior Tool – Katana to create a new project or, you can create a project directly. To create a create case without Katana, you should:

1. Open an xml editor – like Notepad++, Gedit, or any other that is installed on your system

2. Start creating tags – for reference, you can refer to the final figure in Katana User Guide: Section 5 which shows a complete project, or you can follow the Katana steps (Katana User Guide: Section 5) and go on creating the tags and adding their respective values in your editor. Be sure that you close every tag that you open and that every attribute value is enclosed in double quotation marks. You can refer to -

http://www.xmlobjective.com/category/guide/

to better understand how to create a valid xml file.

3. Once you are satisfied with the testsuite that you have created, it is recommended that you run this file through Ironclaw (Section 3.3) so that you are sure that this is a valid project.

16.0 HOW TO CREATE AN INPUT DATA FILE

A Warrior Input Data File is an XML file written in the system based format that acts as a data store for the Warrior Testcase. To correctly write a Warrior Input Data File, understanding structure of the Input Data File is very important. To understand this, please refer to section 9.0 of this User Guide.

You can use the Warrior Tool – Katana to create a new input data file or, you can create it directly using any available XML editor. To create a create case without Katana, you should:

1. Open an xml editor – like Notepad++, Gedit, or any other that is installed on your system

2. Start creating tags – for reference, you can refer to the final figure in Katana User Guide: Section 6 which shows a complete project, or you can follow the Katana steps (Katana User Guide: Section 6) and go on creating the tags and adding their respective values in your editor. Be sure that you close every tag that you open and that every attribute value is enclosed in double quotation marks. You can refer to -

http://www.xmlobjective.com/category/guide/

to better understand how to create a valid xml file.

17.0 EXECUTING THE TESTCASE

17.1 THROUGH CLI

Warrior test cases can be executed in multiple ways. Below is a list of CLI execution options:

17.1.1 RUN A TESTCASE

```
Warrior /path/testcase.xml
```

17.1.2 RUN MULTIPLE TESTCASES

```
Warrior /path/testcase1.xml /path/testcase2.xml
```

17.1.3 RUN A TESTSUITE

```
Warrior /path/testsuitename.xml
```

17.1.4 RUN MULTIPLE TESTSUITES

```
Warrior /path/testsuite1.xml /path/testsuite2.xml
```

17.1.5 RUN A PROJECT

```
Warrior /path/projectname.xml
```

17.1.6 RUN A COMBINATION OF TESTCASE, TESTSUITE, AND PROJECT

```
Warrior /path/testcase.xml /path/testsuite1.xml /path/projectname.xml
```

17.1.7 SCHEDULE EXECUTION

```
Warrior -scheduletime datetime /path/to/testcase1.xml /path/to/testcase2.xml
/path/to/testcase3.xml
```

17.1.8 RUN KEYWORDS IN A TESTCASE IN PARALLEL, TESTCASES IN SEQUENCE

```
Warrior -kwparallel -tcsequential /path/to/testcase1.xml
/path/to/testcase2.xml /path/to/testcase3.xml
```

17.1.9 RUN KEYWORDS IN A TESTCASE IN SEQUENCE, TESTCASES IN PARALLEL

```
Warrior -tcparallel /path/to/testcase1.xml /path/to/testcase2.xml
/path/to/testcase3.xml
```

17.1.10 RUN MULTIPLE TIMES

```
Warrior -RMT <numeric value> /path/to/testcase1.xml /path/to/testcase2.xml
/path/to/testcase3.xml
```

17.1.11 RUN UNTIL FAILURE

```
Warrior -RUF <numeric value> /path/to/testcase1.xml /path/to/testcase2.xml
/path/to/testcase3.xml
```

17.1.12 RUN A PROJECT

```
Warrior -RUP <numeric value> /path/to/testcase1.xml /path/to/testcase2.xml
/path/to/testcase3.xml
```

17.1.13 EXECUTION BASED ON CATEGORY

Warrior CLI supports execution of test cases by category. The category field within the testcases will be read to determine if a test case is a match. By providing a category and a test directory, Warrior will search the test cases in the directory for test cases that match the give category and they allow the user to execute the test cases.

Arguments:

`-runcat` (mandatory): provide a list of test case categories to be searched for and executed.

`-tcdir` (optional): list of directories to search for testcase xml files, default=cwd.

`-suitename` (optional): name of test suite xml file to be created.

`-suitelocn` (optional): path to location where the test suite xml file will be created, default=cwd

Format:

```
./Warrior -runcat cat1 cat2 cat3 -tcdir path/to/dir1 path/to/dir2 path/to/dir3
-suitename testsuite_file_name -suitelocn location_to_create_suite_xml_file
```

Commands:

Search the cwd for testcases that matches at least one of the provided categories and executes them as individual testcases. Note: The test cases will not be grouped into a test suite using this command.

```
./Warrior -runcat cat1 cat2 cat3
```

Search the provided directories path/to/dir1 path/to/dir2 path/to/dir3 for testcases that matches atleast one of the provided categories cat1 cat2 cat3, and executes them as individual testcases.
Note: No testsuite created automatically in this case.

```
./Warrior -runcat cat1 cat2 cat3 -tcdir path/to/dir1 path/to/dir2 path/to/dir3
```

Search the cwd for testcases that matches at least one of the matching categories cat1 cat2 cat3
Creates a testsuite xml with the provided suite name in the cwd and executes the testsuite.

```
./Warrior -runcat cat1 cat2 cat3 -suitename testsuite_file_name
```

Search the cwd for testcases that matches at least one of the provided categories cat1 cat2 cat3
Creates a testsuite xml with the provided suite name in the provided suite locationn and executes the testsuite.

```
./Warrior -runcat cat1 cat2 cat3 -suitename testsuite_file_name -suitelocn
location_to_create_test_suite_ xml_ file
```

Search the provided directories path/to/dir1 path/to/dir2 path/to/dir3 for testcases that matches at least one of the provided categories cat1 cat2 cat3. Creates a testsuite xml with the provided suite name in the cwd and executes the testsuite.

```
./Warrior -runcat cat1 cat2 cat3 -tcdir path/to/dir1 path/to/dir2 path/to/dir3
-suitename testsuite_file_name
```

Search the provided directories path/to/dir1 path/to/dir2 path/to/dir3 for testcases that matches at least one of the provided categories cat1 cat2 cat3. Creates a testsuite xml with the provided suite name in the provided suite location and executes the testsuite.

```
./Warrior -runcat cat1 cat2 cat3 -tcdir path/to/dir1 path/to/dir2 path/to/dir3
-suitename suite_file_name -suitelocn location_to_create_suite_xml_file
```

17.1.14 JIRA BUG REPORTING

For JIRA bug reporting, configure `Warrior/ Tools/Jira/jira_config.xml` file

Default JIRA project definition:

Warrior will use the first project it finds in the JIRA config file marked `default="true"`. Do not have multiple default projects. 2. If no projects are marked as `default="true"`, then it is the first project in the `jira_config.xml`

Commands:

Automatically creates JIRA defects against the default JIRA project from JIRA config file.

```
./Warrior -ad path/to/tc1.xml path/to/tc2.xml path/to/ts1.xml path/to/ts2.xml
path/to/proj1.xml path/to/proj2.xml
```

Automatically creates JIRA defects against the provided JIRA project from JIRA config file.

```
./Warrior -ad -jiraproj jiraproj name path/to/tc1.xml path/to/tc2.xml
path/to/ts1.xml path/to/ts2.xml path/to/proj1.xml path/to/proj2.xml
```

Creates defects in default JIRA project for all the defects JSON files present in defects_directory1, defects_directory2, defects_directory3

```
./Warrior -ujd -ddir path/to/defects_directory1 path/to/defects_directory2 path/to/defects_directory3
```

Creates defects in default JIRA project for the defects JSON files path/to/defects_json1 path/to/defects_json2 path/to/defects_json3

```
./Warrior -ujd -djson path/to/defects_json1 path/to/defects_json2 path/to/defects_json3
```

Creates defects in provided JIRA project for all the defects JSON files present in defects_directory1, defects_directory2, defects_directory3

```
./Warrior -ujd -jiraproj jiraproj_name -ddir path/to/defects_directory1 path/to/defects_directory2 path/to/defects_directory3
```

Creates defects in provided JIRA project for the defects JSON files path/to/defects_json1 path/to/defects_json2 path/to/defects_json3

```
./Warrior -ujd -jiraproj jiraproj_name -djson path/to/defects_json1 path/to/defects_json2 path/to/defects_json3
```

17.1.15 CREATING SUITES

Warrior offers CLI support for creation of a default test suite from a list of testcases.

Arguments:

`-cs` (mandatory): used to setup suite creation from command line, takes no values

`-suitename` (mandatory): name of test suite xml file to be created.

`-suitelocn` (optional): path to location where the test suite xml file will be created, default=cwd

`-cat` (optional): to add testcases by Category

`-tcdir` (optional): The testcase directory

`filepaths`: list of testcase xml files, not required when used to create suite with –category.

Format:

```
./Warrior -createsuite -suitename testsuite_file_name -suitelocn location_to_create_test_suite_xml_file path/to/tc1.xml path/to/tc2.xml path/to/tc2.xml
```

```
./Warrior -createsuite -cat cat1 cat2 cat3 -suitename testsuite_file_name -suitelocn location_to_create_test_suite_xml file -tcdir path/to/dir1 path/to/dir2 path/to/dir3
```

Commands:

Creates a testsuite with provided name and with the povided list of testcases in the cwd, with default values for other suite parameters.

```
./Warrior -createsuite -suitename testsuite_file_name path/to/tc1.xml path/to/tc2.xml path/to/tc2.xml
```

Creates a testsuite with provided name and provided list of testcases in the provided suitelocn, with default values for other suite parameters.

```
./Warrior -createsuite -suitename testsuite_file_name-suitelocn location_to_create_test_suite_xml_file path/to/tc1.xml path/to/tc2.xml path/to/tc2.xml
```

Create a testsuite with provided name in the cwd. Search the cwd for testcases matching atleast one of the provided categories cat1 cat2 cat3.

```
./Warrior -createsuite -suitename testsuite_file_name -cat cat1 cat2 cat3
```

Search the cwd for testcases matching at least one of the provided categories cat1 cat2 cat3.

Create a testsuite with provided name in the provided suitelocn.

```
./Warrior -createsuite -suitename testsuite_file_name -suitelocn location_to_create_test_suite_xml_file -cat cat1 cat2 cat3
```

Searches the provided list of test case directories [<path/to/dir1> <path/to/dir2> <path/to/dir3>] for testcases matching at least one of the provided categories cat1 cat2 cat3. Create a testsuite with provided name in the provided suite location.

```
./Warrior -createsuite -suitename testsuite_file_name -suitelocn location_to_create_test_suite_xml_file -cat cat1 cat2 cat3 -tcdir path/to/dir1 path/to/dir2 path/to/dir3
```

17.1.16 ENCRYPT PASSWORDS

Run this command to encrypt a password:

```
Path/to/Warrior –encrypt plain_text_password
```

You can add 'sudo' in front of the command, if you want run it as an admin. This command will give you an encoded password.

```
The encrypted text for plain_text_password is:
25298e188e659bba82672dcb317215847c68db0f39998c7f92deeb455e86648c
```

Encrypt a password with a specific Secret Key

To encrypt a plain text password using a specific key, run the following command:

```
Path/to/Warrior –encrypt plain_text_password –secretkey IamaNinjaWarrior
```

This would encrypt plain_text_password by using "IamaNinjaWarrior" as the secret key. The secret key should be 16 letters – counting special characters and spaces – in length.

17.2 THROUGH KATANA

Warrior can also be executed through Katana. Please refer to Katana User Guide: Section 4 for all the details.

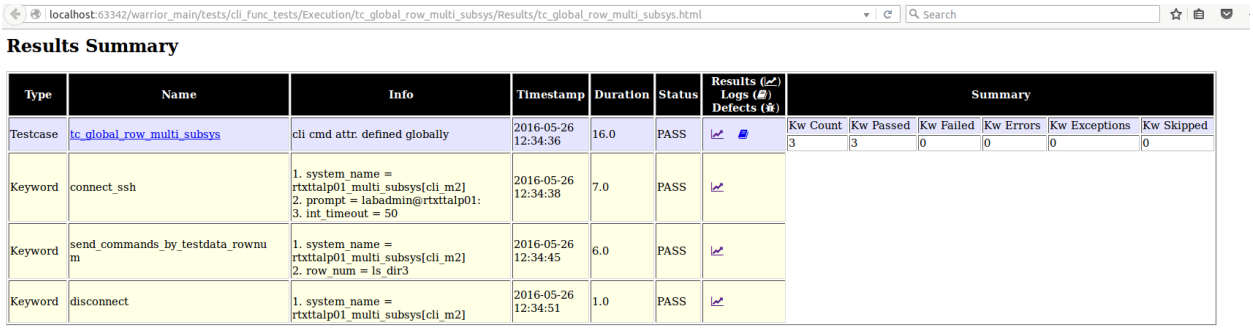
18.0 UNDERSTANDING THE RESULTS

When a testcase, suite, or a project is run, it creates a result directory. Being able to understand this result directory is a big asset in debugging and understanding your keywords and/or the testcases, suites, and projects. The results directory would of the same name that the testcase, suite, or project that was run and it would be time stamped. This directory would be created either in the path mentioned in the w_settings file - if that is left empty, then in the path mentioned in the testcase, suite, or project - if that is left empty, then in the default directory – the Execution folder in Warrior space.

If the results directory is the first one in the destination for that particular testcase, suite, or project, then the time stamp would not be added. This results directory will have three subfolders underneath in – Defects, Logs, and Results. If the testcase (or suite, or project) passed with no defects, error, or exceptions, then the defects folder would be empty.

The Logs folder would contains logs from all the system that were used by the testcase and also the consoleLogs. The consoleLogs contain all the activity that went on, on the console when the testcase (or suite or project) was running. These consoleLogs contain all the information you need to know what happened during the execution – but, consoleLogs can be tedious to read, so Warrior has the HTML result file inside the Results directory.

The Results directory has one subfolder – Keyword Results – and three sub-files inside it. Out of those three sub-files, one is an HTML file which can opened in a browser. It would look something like this:


localhost:63342/warrior_main/tests/cli_func_tests/Execution/tc_global_row_multi_subsys/Results/tc_global_row_multi_subsys.html

Results Summary

Type	Name	Info	Timestamp	Duration	Status	Results (📈) Logs (🗐) Defects (🐞)	Summary					
							Kw Count	Kw Passed	Kw Failed	Kw Errors	Kw Exceptions	Kw Skipped
Testcase	tc_global_row_multi_subsys	cli cmd attr. defined globally	2016-05-26 12:34:36	16.0	PASS	📈 🗐	3	3	0	0	0	0
Keyword	connect_ssh	1. system_name = rtxttalp01_multi_subsys[cli_m2] 2. prompt = labadmin@rtxttalp01: 3. int_timeout = 50	2016-05-26 12:34:38	7.0	PASS	📈						
Keyword	send_commands_by_testdata_rownum	1. system_name = rtxttalp01_multi_subsys[cli_m2] 2. row_num = ls_dir3	2016-05-26 12:34:45	6.0	PASS	📈						
Keyword	disconnect	1. system_name = rtxttalp01_multi_subsys[cli_m2]	2016-05-26 12:34:51	1.0	PASS	📈						

This HTML result file gives a visual appeal to the result files. Since this is a screenshot of an HTML result file of a testcase, only the testcase and the keywords are showing in it, but if it were a suite or a project, all the testcases inside the suite, and all the suites inside the project would be displayed.

Here you can see, that just by glancing that the result file, you can determine which keywords passed and which did not. The results are also segregate the keywords based on the final execution state of that keyword, so you know exactly how many passed, failed, threw an Error or an Exception.

Clicking on the Result Icon in the Results, Logs, and Defects column redirects you to the corresponding file containing the result xml file of that particular entity. Clicking of the Logs icon would direct you to logs, and clicking on the defects icon would direct you to the corresponding defects file.

Then, there is the JUnit result file – this file is generated for the benefit of external tools such as Jenkins which can read the JUnit files generated by testcase executions to display the results graphically.

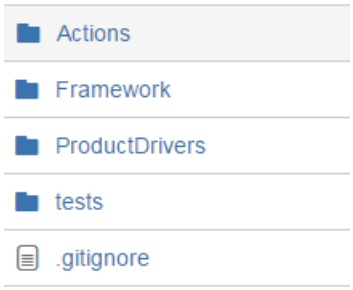
This JUnit file is in the correct JUnit format. This file provides a very high level view of the execution and its result.

The XML result file is the third file found under the Results directory. This result file is internal to Warrior that has more details about the execution than the JUnit result file. This file is similar to the JUnit file but but in a format that has been decided by Warrior.

19.0 GUIDELINES TO CREATE A WARRIOR-KEYWORDS REPOSITORY

Any product specific keywords that are developed should have a particular file structure for Warrior to work as expected. To adhere to this file structure, it is important that any Warrior Keywords repository be structured correctly. The following are a few guidelines that would help you create a brand new Warrior-Keywords repository:

1. The repository name must be the name of the product for which you are developing the Warrior Keywords. This would be the root of the repository.
2. Inside the root of the repository, a directory structure like this must be maintained:



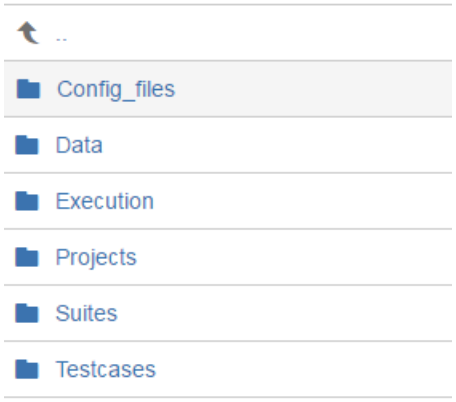
3. The Actions folder must contain all the keywords, the Framework folder must contain all the associated utils, and the ProductDrivers must contain all the product drivers.
4. The tests directory should be used only for uploading the unit tests related to this keyword repository. Do NOT upload Warrior tests in this directory.

20.0 GUIDELINES TO CREATE A WARRIORSPACE REPOSITORY

Any product specific warriorspaces that are developed are recommended to have a file structure so that writing and managing tests is simplified. For this, it is important that the warriorspace repository be structured well. The following are a few guidelines that would help you create a brand new Warriorspace repository:

1. The repository name should be the name of the product for which you are developing the Warriorspace. This would be the root of the repository.
2. Inside the root of the repository, a directory structure like this should be maintained:

3. Inside this Warriorspace directory, this file structure is recommended:



4. The Config_files folder should contain all the Testdata files and the Variable Config files, the Data folder should contain all the Input Data files, the Projects folder should contain all the Project files, the Suites should contain all the Suite files, and the Testcases should contain all the Testcase files.
5. The Execution directory is not necessary, and ca be skipped completely while creating the repository.

KATANA

USER GUIDE

1. KATANA: AN INTRODUCTION

Katana is Warrior's web based testcase creation and execution tool. From Katana, users can create testcases, suites, projects, and input data files. Testcases, suites, projects, and input data files in Katana are web based forms that once saved, create the appropriate xml files. Each field in the form has a tooltip explaining its functionality. Katana also allows you to execute testcases, suites, and projects from the execute tab.

1.0 WHERE IS KATANA LOCATED?

Katana is currently located at:

http://rtx-swtl-git.fnc.net.local/projects/WAR/repos/katana/browse

1.1 THE PREREQUISITES

Katana requires git to be installed – this is required solely to clone Katana into your machine.

If you are working on Windows, you will need to install python27 before you can run the Katana server.

If you are working on any Linux based machines, you'll need xterm available on your machine. Xterm is required only when you are executing Warrior through Katana. If you don't intend to execute Warrior through Katana, but only need Katana to build your Testcases, Suites, and Projects for you, then you don't need xterm installed.

1.2 HOW TO INSTALL KATANA?

Katana can be cloned into any directory as desired. Open your terminal, 'cd' into the directory of your choice and then type in this:

```
git clone http://USERNAME@rtx-swtl-git.fnc.net.local/scm/war/katana.git
```

Don't forget to replace the USERNAME with your actual username.

1.3 HOW TO RUN KATANA?

Katana is easy to run. Open your terminal and cd into the directory where you have cloned Katana. Then, type in:

```
cd katana/Katana
```

Once you are inside the Katana directory, type in:

```
python katana.py
```

This starts up Katana on the localhost at the port 5000. You can access Katana by opening any browser and typing in:

```
localhost:5000
```

If you wish to start up Katana on a different port, you can use the following command:

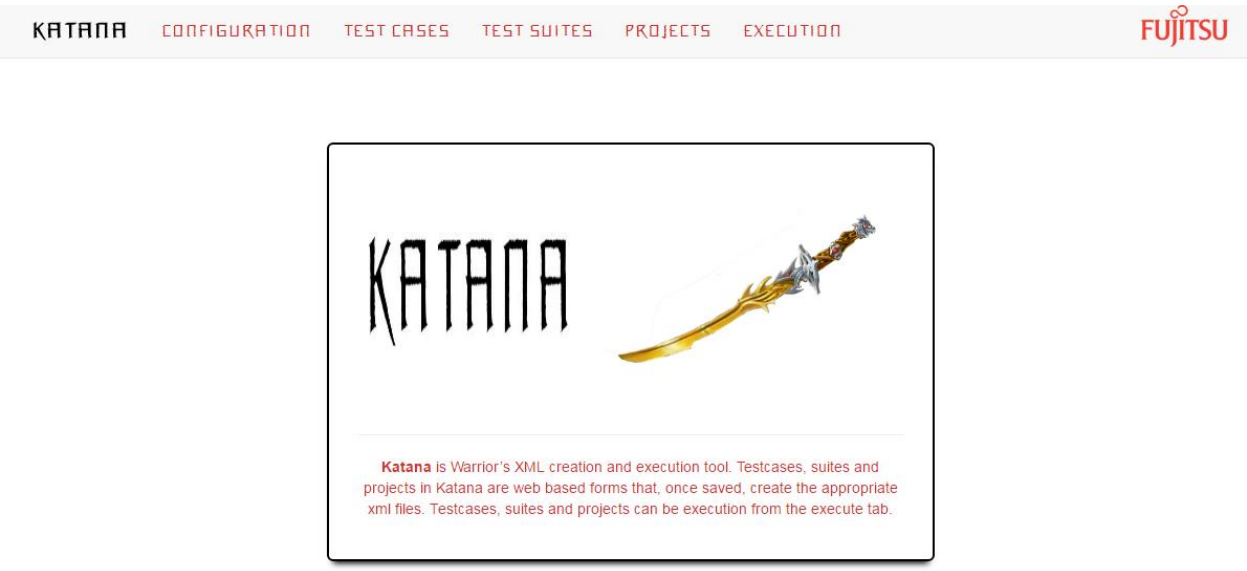
```
python katana.py -p 5001
```

This starts up Katana on the localhost at the port 5001. Now, you can access Katana by opening any browser and typing in:

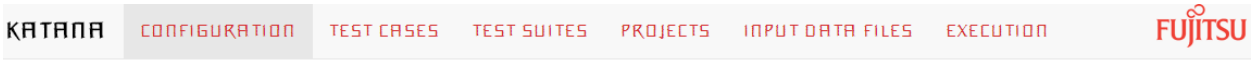
```
localhost:5001
```

2. GETTING STARTED WITH KATANA

Katana starts up on the localhost and this page shows up



Katana has 7 tabs on the top in total. To configure Katana to your system, you will need to click on the second tab – the configuration tab – and this page would show up.



CONFIGURATION

On this page you may define the following Katana configuration parameters.

Name of the Engineer
All testcase created/modified in this session will be tagged with this Engineer's name.

Engineer name

Current location of the Warrior Framework Directory
The directory where the Python Action & Driver files will be read from.

Warrior Framework Directory

Test Case Directory
The test case directory is where the XML files created by this application are stored.

Test case destination directory

Test Suite Directory
The test suite directory is where the XML files created by this application are stored.

Test Suite destination directory

Project Directory
The project directory is where the project XML files created by this application are stored.

Project destination directory

Input Data File Directory
The input data file directory is where the data files created by this application will be stored.

Input Data File destination directory

Python Path
Enter the path to the Python that you want to use while executing Warrior - if left empty, the default Python will be used.

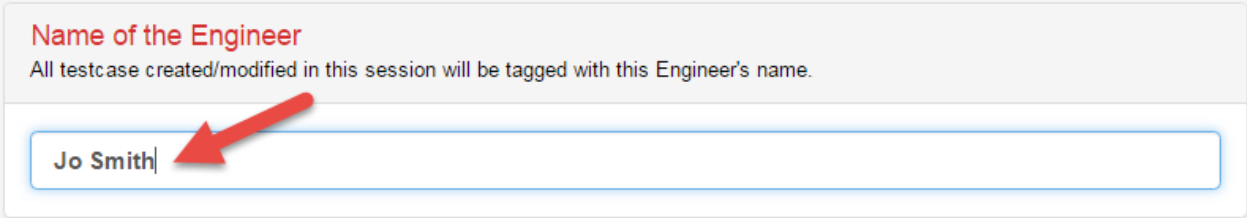
Path to Python

Save

Now, the information on this page is supposed to be filled in once by the User when Katana is fired up for the first time. Each field is explained below:

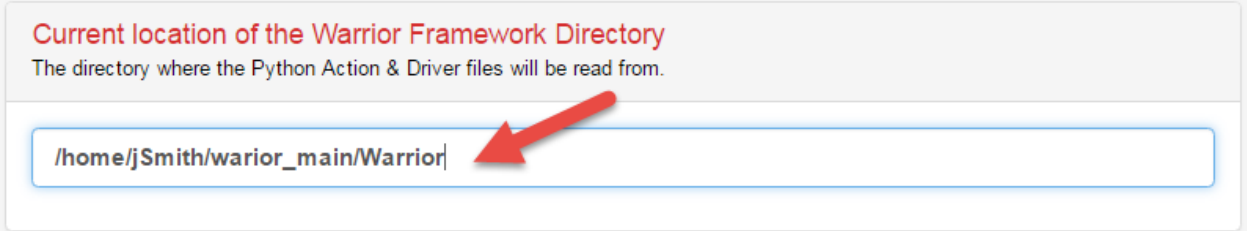
Name of the Engineer

This field should contain your name.



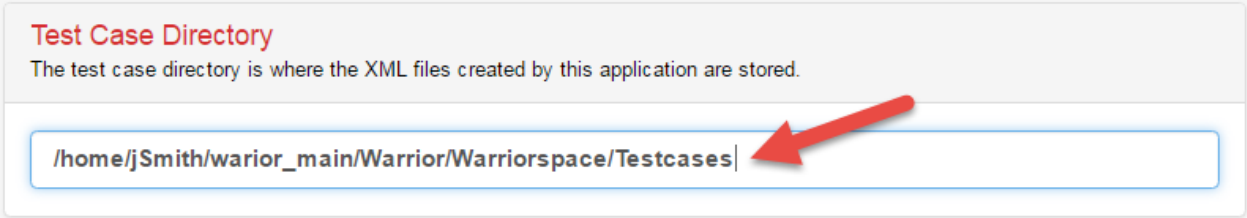
Current Location of the Warrior Framework Directory.

This field should contain the path to the Warrior directory in your machine.



Testcase Directory

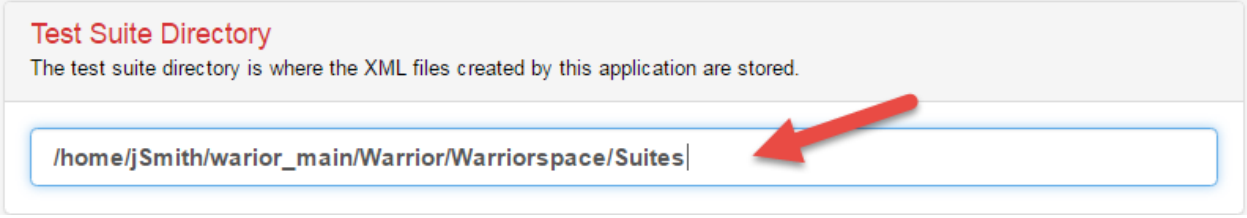
This field should contain the path to the Testcase directory in your machine. This directory may or may not be inside your Warrior directory



To understand what Warrior Testcases are, refer to Warrior User Guide: Section 6.0. To create testcases in Katana, refer to Katana User Guide: Section 3

Suite Directory

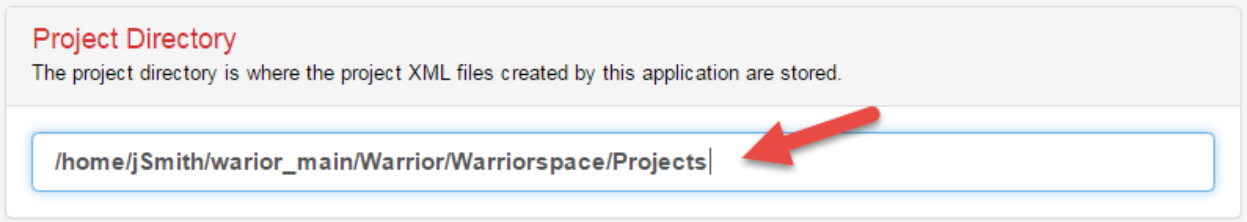
This field should contain the path to the Testsuite directory in your machine. This directory may or may not be inside your Warrior directory



To understand what Warrior Suites are, refer to Warrior User Guide: Section 7.0. To create suites in Katana, refer to Katana User Guide: Section 4

Project Directory

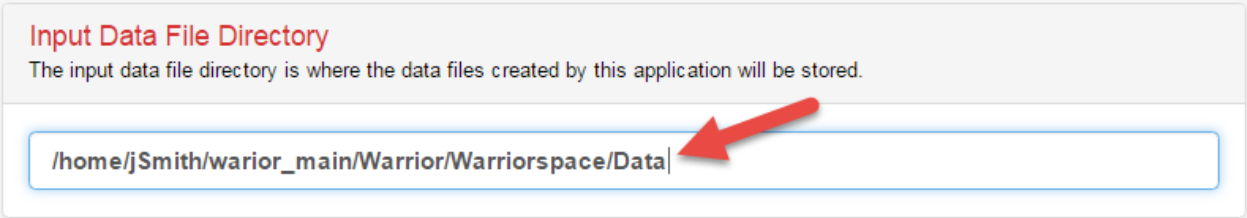
This field should contain the path to the Project directory in your machine. This directory may or may not be inside your Warrior directory



To understand what Warrior Projects are, refer to Warrior User Guide: Section 8.0. To create projects in Katana, refer to Katana User Guide: Section 5

Input Data Directory

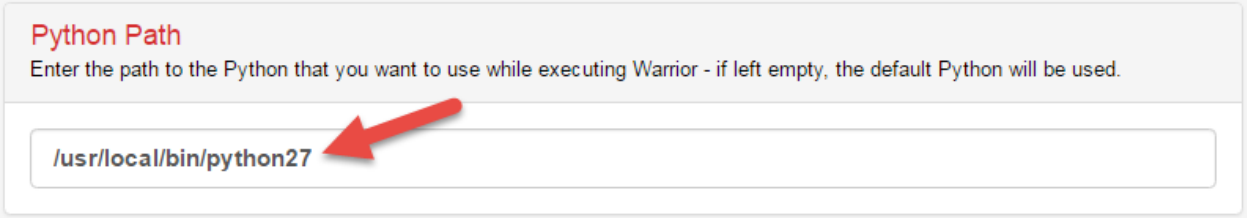
This field should contain the path to the Data directory in your machine. This directory may or may not be inside your Warrior directory



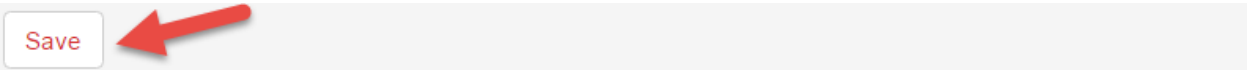
To understand what Warrior Input Data Files are, refer to Warrior User Guide: Section 9.0. To create input data files in Katana, refer to Katana User Guide: Section 6

Python Path

Here, a path to an alternate Python that you may want to use for executing Warrior can be added here. If this field is left empty, then the default Python on your system would be used.



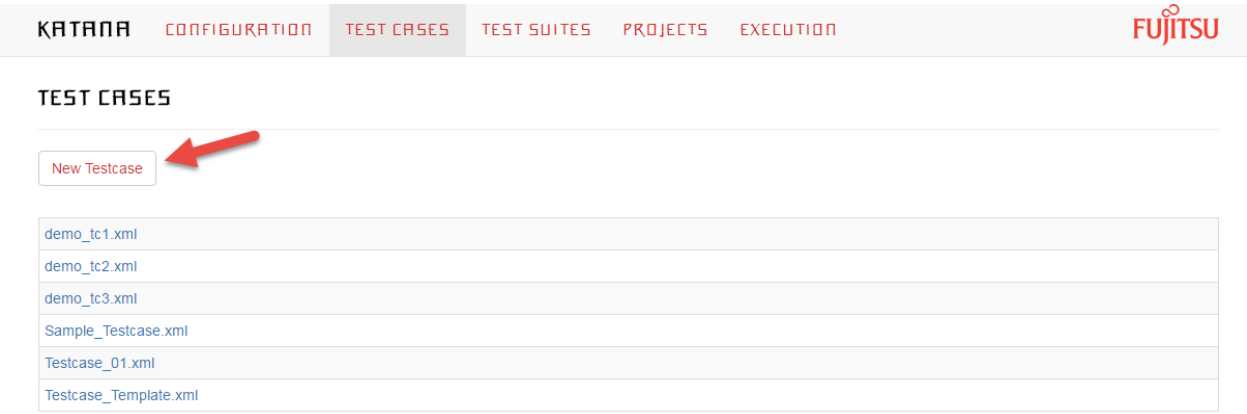
Finally, hit save to save the configuration.



Katana is now ready to be used.

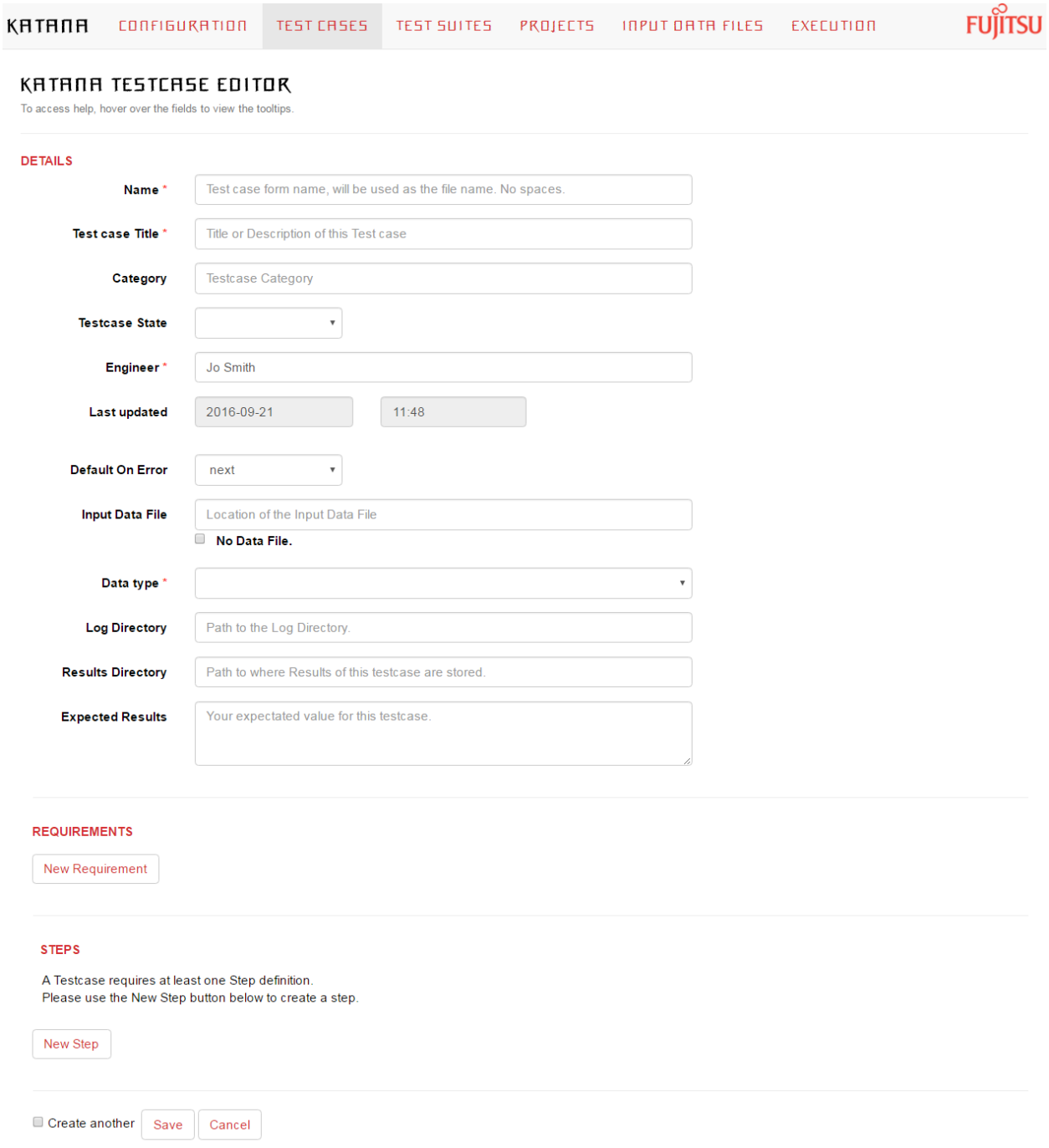
3. CREATING TESTCASES WITH KATANA

Katana allows you to create Testcases through its user interface. You have to start up Katana and go to its Testcases tab. There, you will see a “New Testcase” button:



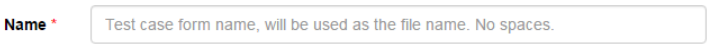
Note: The list of testcases below the “New Testcase” button may differ as Katana would show the Testcases saved in your testcase directory.

On clicking that button, you will be directed to a new page:



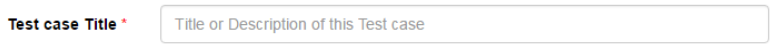
A step by step guide to create a testcase is Katana is given below

1. Name



Here enter the name of the testcase. A name can be anything that you recognize. This is a mandatory field.

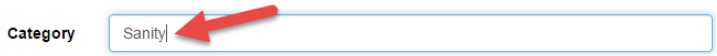
2. Testcase Title



A testcase Title (or description) is a field where you can add a descriptive title to identify what this testcase does. This is a mandatory field.

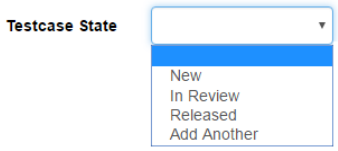
3. Category

In this field, you can classify this testcase into a particular "category". Categorization of testcases is important when you have to run all the testcases in the same category together. Please refer Warrior User Guide: Section 17.1.13 to better understand how a category works. This is an optional field, although, it is recommended that you categorize your testcases.

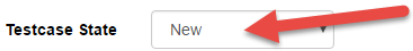


4. Testcase State

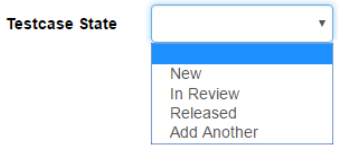
A testcase state is informative when you have to keep track of which testcases are new, released, or in review. These are the in-build categories that are available to you. This is an optional field, but it is recommended that you fill out this field so as to keep track of which testcase is in which state of development.



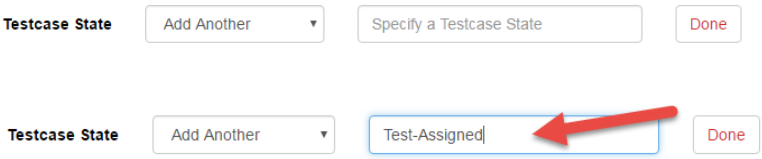
You can select any state of your choice, like “New”. This would mark your testcase as “New” meaning that this is a newly created testcase.



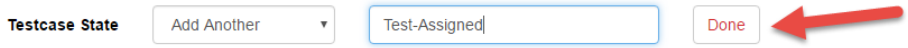
Furthermore, if the available categories do not really fit your testcase state, you can go ahead and select “Add Another”.



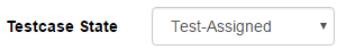
This opens up another field which lets you add a State of your own.



Here you can add in any testcase state that suits the state in which your testcase is, such as “Test-Assigned”, and click “Done”.



Your testcase now has a newly created “Test-Assigned” state. You can also use this newly created states for any other Testcase that you may create after this.

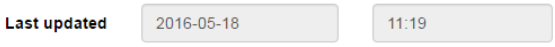


5. Engineer

This field indicates the name of the Engineer who created this Testcase. This is a prefilled field but, you can change it if you want to. This is a mandatory field.

6. Last Updated (Date-Time stamp)

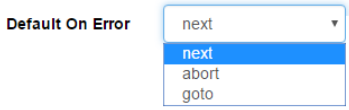
This field indicates the Date and Time on which this Testcase was last updated. This is a prefilled field, but the contents cannot be changed.



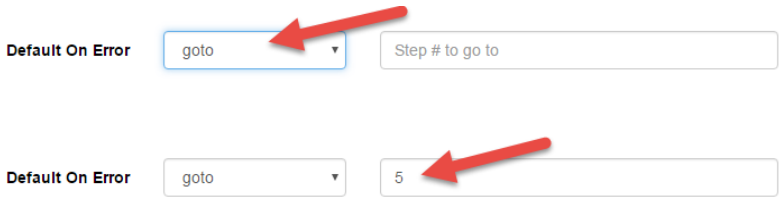
7. Default on Error

This is field where you can specify what the Testcase should do it case it errors out. By default, it is "next", that is, if you don't make a change, the Testcase will proceed to the next Testcase available for execution if the current testcase throws an error.

Clicking on the dropdown shows all the options that are available to you to tell Warrior what to do in the case where the current Testcase errors out. You have the ability to choose any one of them.



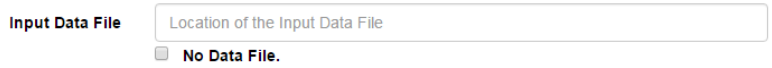
On selecting "goto", another pops up and lets you add the Testcase number that Warrior should go to in case of an error.



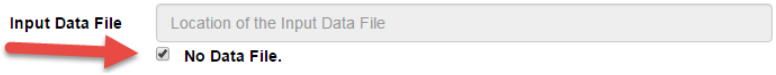
For further information about what Default on Error does, please go through Warrior User Guide: Section 6

8. Input Data File

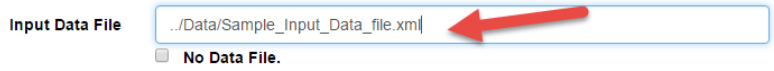
This is field wherein you can add the path to a datafile (Warrior User Guide: Section 9) that you want this testcase to use. This field is mandatory only if the "No Data File" checkbox is left unchecked.



Once the "No Data File" checkbox is checked, the field grays out and then you don't have to enter a path to the datafile.

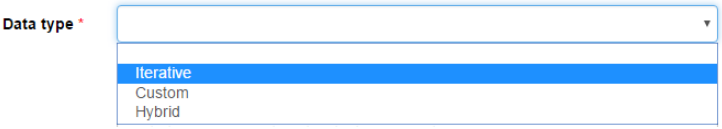


In the case where you want a datafile included in your Testcase, uncheck the "No Data File" checkbox and add either an absolute path to your datafile or a path relative from your testcase. It is recommended that you add a relative path here.



9. Data type

This dropdown field lets you indicate what kind how a testcase should interact with the datafile (Warrior User Guide: Section 9.0). This is a mandatory field.

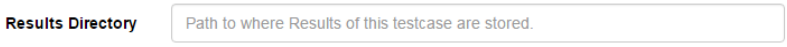


10. Log Directory

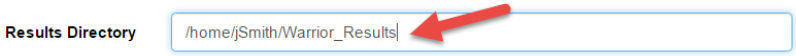
This field lets you specify the path to the Logs directory – this directory is where all the logs returned by Warrior would be stored. This is an optional field.

If left empty, the logs would be directed and stored in the default location.

11. Results Directory



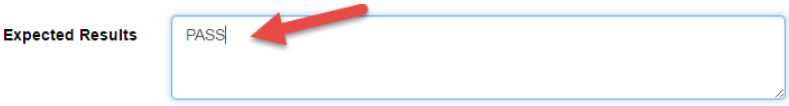
This field lets you specify the path to the Results directory – this directory is where all the results returned by Warrior would be stored. This is an optional field.



If left empty, the results would be directed and stored in the default location.

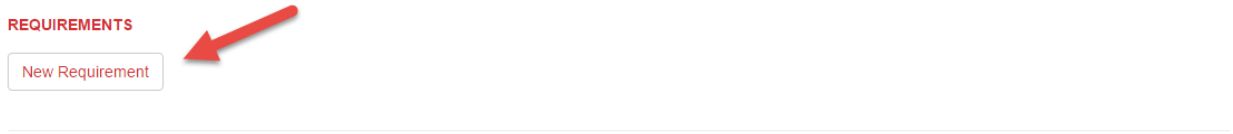
12. Expected Results

This field is where you can specify what kind of a result is expected from the testcase. This is an optional field as it is an informative field for you.

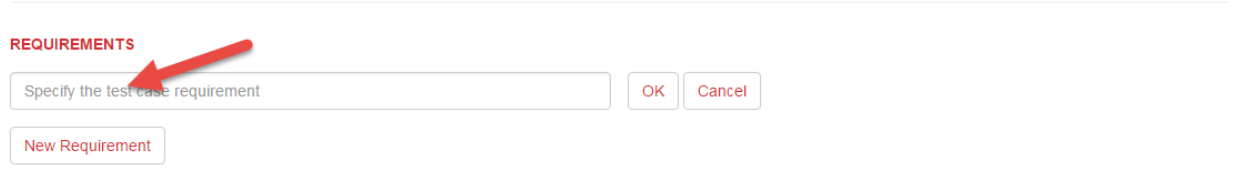


13. Requirements

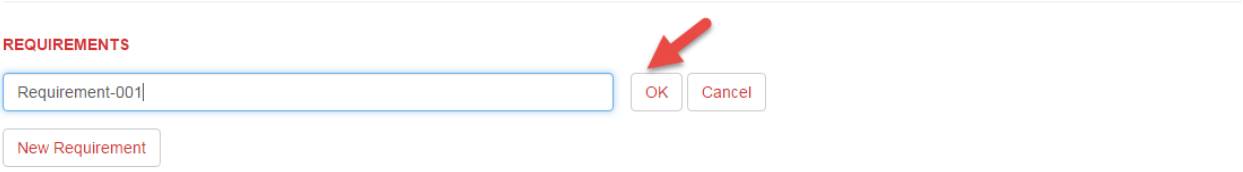
This field is for you add any requirements that this testcase may have. This is an optional field. You can add a new requirement by clicking on the “New Requirement” button.

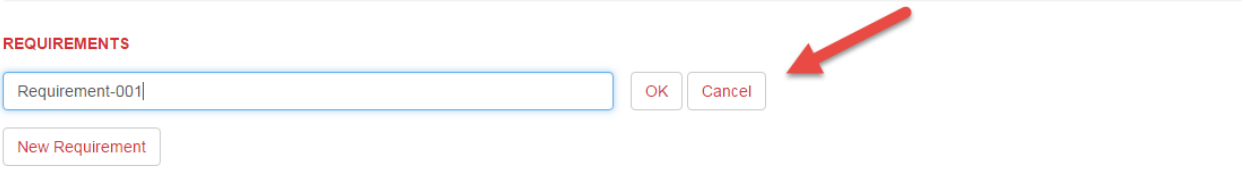


This opens up a new field in which you can specify the requirement.

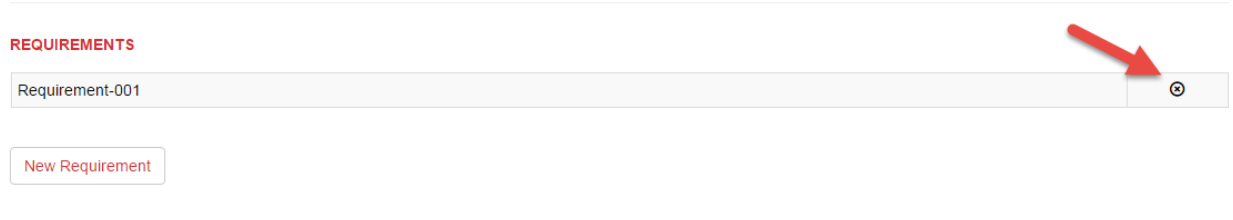


Once typed in, you can hit “OK” to save it or “Cancel” to discard all changes.



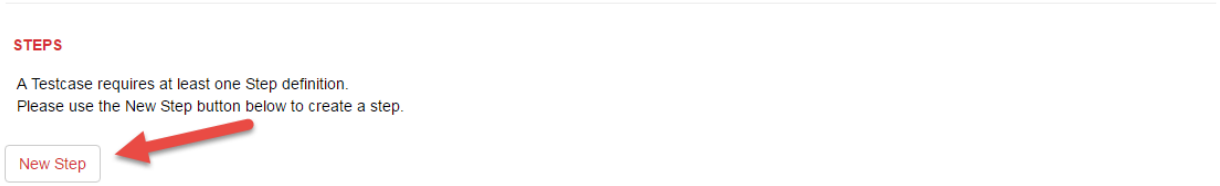


You can also delete an added requirement by clicking the minute “x” against the Requirement that you want to delete.

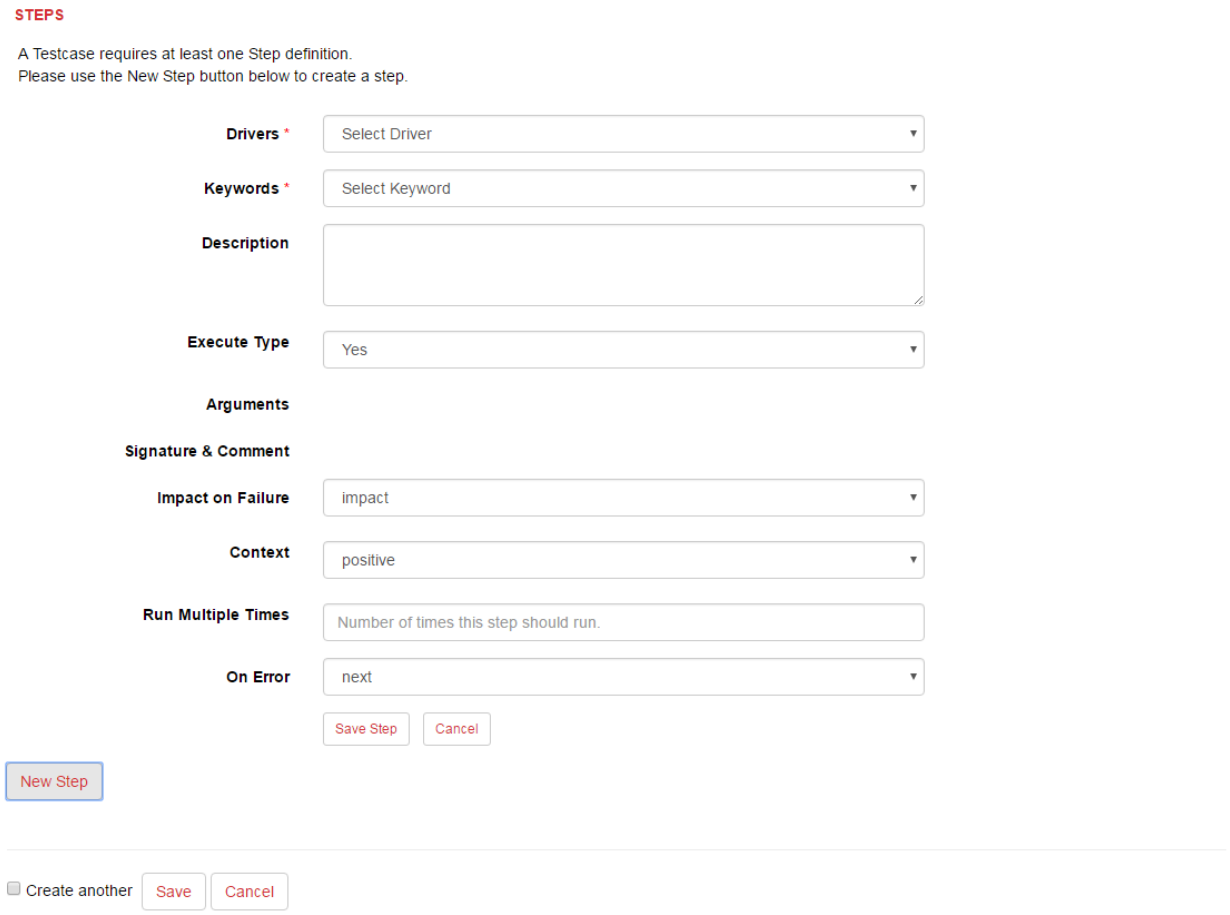


14. Step

Every Testcase requires at least one step. A new step can be added to the Testcase by clicking the “New Step” button. The first entire step is mandatory. Every step after that is optional.



On clicking the “New Step” button, a new set of fields drop down for you to be filled. All the newly appeared fields are a part of this new “step”.



15. Drivers

Every step must have an associated driver to it. This is a mandatory field.

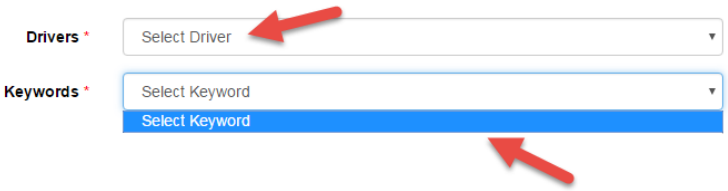
Katana lets you select a driver by showing all the drivers that you have in your Warrior directory in the dropdown. You can select any driver out of those. You would not be able to select any driver which lies outside of the ProductDrivers directory. This is because Warrior enforces a strict directory structure for the Actions and ProductDrivers.

The list of drivers on your Katana may differ since you may have different drivers in your ProductDrivers directory.

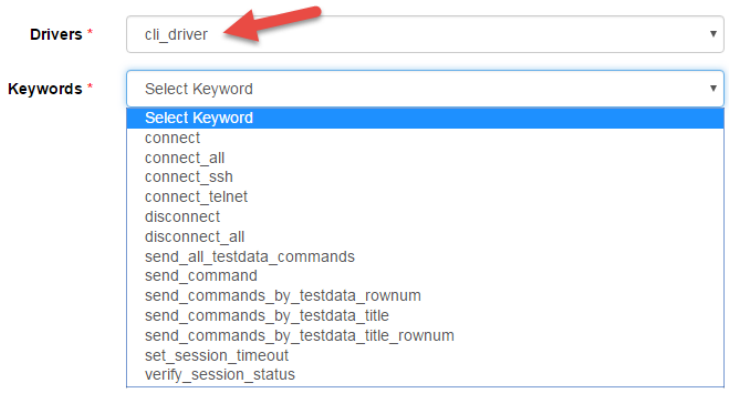
16. Keywords

Every step must have an associated keyword to it. This is a mandatory field

If no driver is selected, no keywords will show up in the dropdown. You have to choose the driver before you can choose the Keyword.

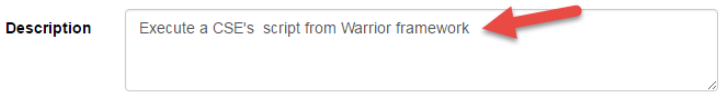


If the driver is chosen, a list of Keywords would show up. You can select any Keyword out of those.



17. Description

This field is populated when the Keyword is chosen. The description is for your reference, so that, you know exactly what this keyword does. This is an optional field.

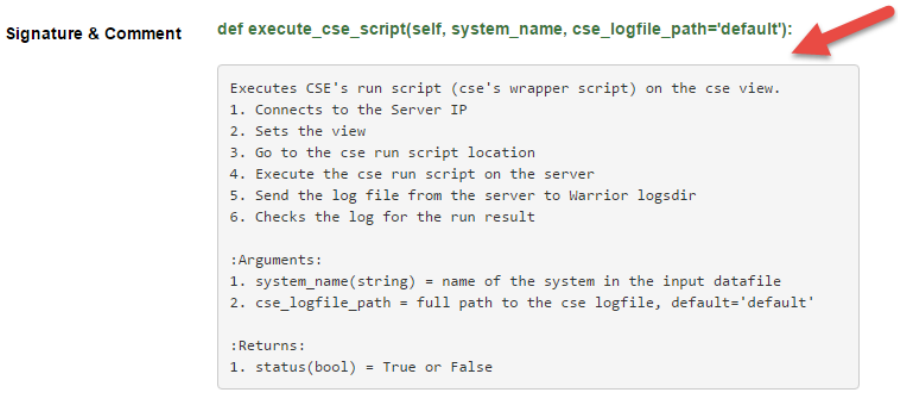


18. Arguments

This space gets populated when the keyword is selected. All the arguments accepted by the Keyword show up here. You can type in the values associated with each value against that argument name. To identify which are the mandatory arguments, you can go through the next section.

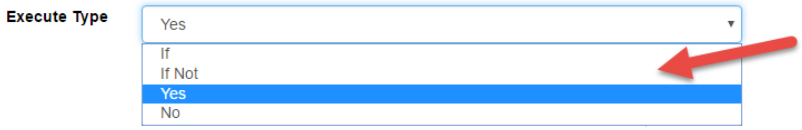
19. Signature & Comments

This field also appears prepopulated when the Keyword is selected. This is basically for you benefit, so that you understand what the keyword does when it run, what kind of arguments it accepts, and what kind of values it returns.

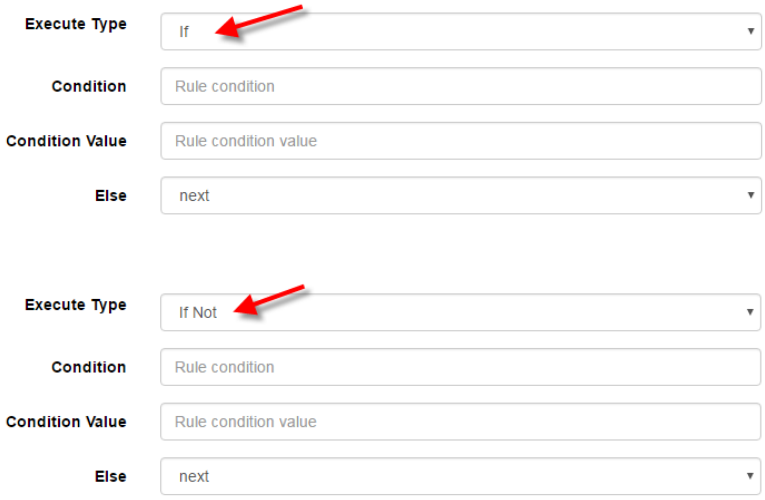


20. Execute Type

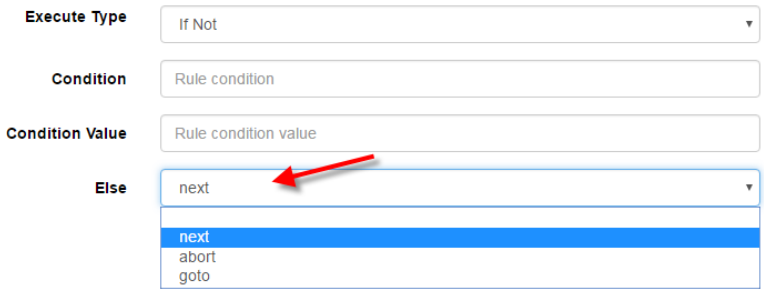
The Execute Type is by default set to 'Yes'. You can change it by clicking on the dropdown. This reveals all the options you have available to you.



On clicking on 'If' and 'If Not', two other input boxes show up that let you add your conditions.

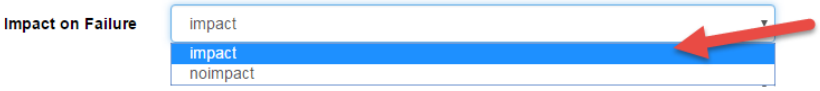


The third input box is a dropdown consisting of the condition that handles the scenario where the condition in 'If'/'If Not' is not met and hence Warrior would need to know what to do in such a case.



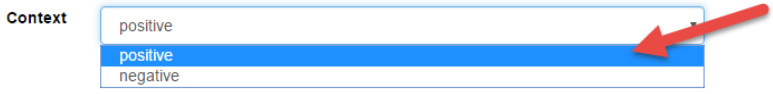
21. Impact on Failure.

On clicking the dropdown you would be able to select from one of the options presented to you. The default is 'impact'.



22. Context

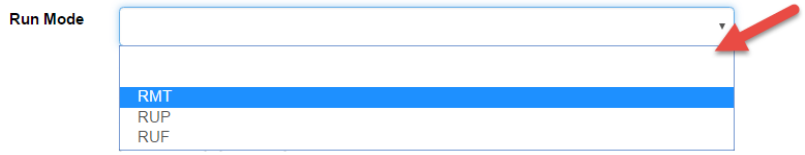
On clicking the dropdown you would be able to select from one of the options presented to you. The default is 'positive'.



23. Run Mode

This field lets you select if you want this keyword to run until pass (RUP), run until failure (RUF), or run multiple times (RMT). This is an optional field.

The Run Mode field is dropdown consisting of values “RUP”, “RUF”, and “RMT”.



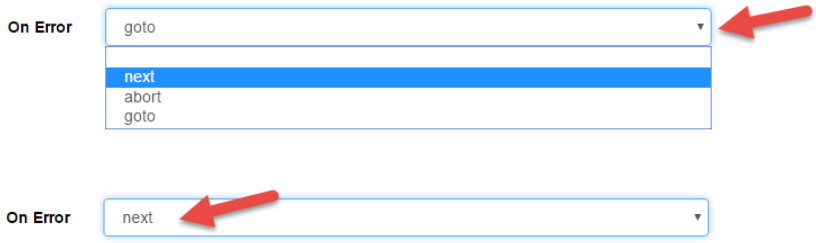
Whenever a value from the dropdown is selected, and new field “Value” pops up that lets you fill in the maximum number of times you want to run this step – if RUP or RUF is selected or the number of times you want to run this step if RMT is selected.

You can input the number of times you want this particular step to run here.

24. On Error

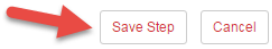
This tag handles the Error condition. You can click on the dropdown to choose an option. The default is whatever the testcase onError value is.

You can click on the dropdown to assign a different error handling for this step.

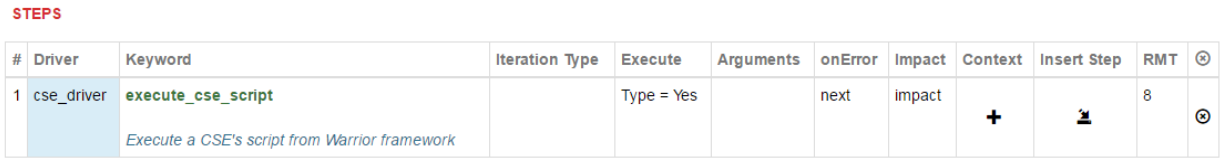


25. Save Step

Clicking on the 'Save Step' button saves this step. This DOES NOT save the testcase – only that particular step. A step must be saved

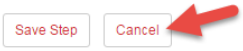


Once the step is saved, a summary of it would appear on that page. This would look something like this:



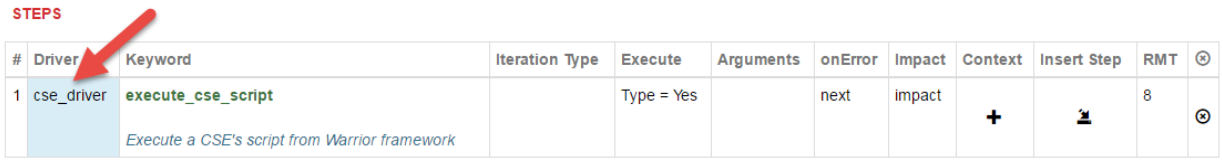
26. Cancel

On the other hand, clicking on 'Cancel' discards all the changes made to this step.



26. Edit step

Clicking on the driver name lets you edit an already created step.



This opens up the step editor once again and you can make the desired changes.

STEPS

#	Driver	Keyword	Iteration Type	Execute	Arguments	onError	Impact	Context	Run Mode	Insert Step	
1	cse_driver	**execute_cse_script** *Execute a CSE's script from Warrior framework*		Type = Yes		next	impact	+	RMT = 8		⊗

Drivers * cse_driver

Keywords * execute_cse_script

Description Execute a CSE's script from Warrior framework

Execute Type Yes

Arguments system_name

cse_logfile_path

Signature & Comment def execute_cse_script(self, system_name, cse_logfile_path='default'):

```
Executes CSE's run script (cse's wrapper script) on the cse view.
1. Connects to the Server IP
2. Sets the view
3. Go to the cse run script location
4. Execute the cse run script on the server
5. Send the log file from the server to Warrior logsdir
6. Checks the log for the run result

:Arguments:
1. system_name(string) = name of the system in the input datafile
2. cse_logfile_path = full path to the cse logfile, default='default'

:Returns:
1. status(bool) = True or False
```

Impact on Failure impact

Context positive

Run Mode RMT

Value * 8

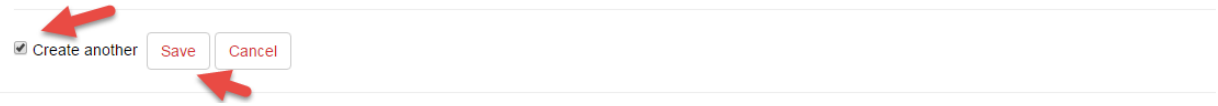
On Error next

Save Step Cancel

27. Save

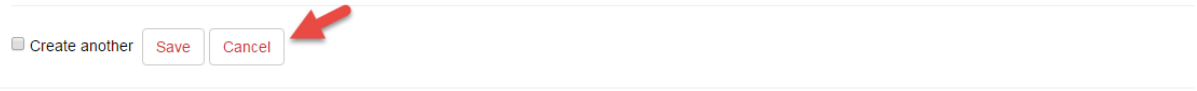
Clicking on ‘Save’ saves the entire Testcase. If any unsaved step is present when this button is clicked, hat unsaved step will get saved.

Checking the box against “Create Another” and then hitting Save, lets you save the testcase and directly opens up a page where you can start creating a new Testcase.



28. Cancel

Clicking on ‘Cancel’ discards all the changes performed.

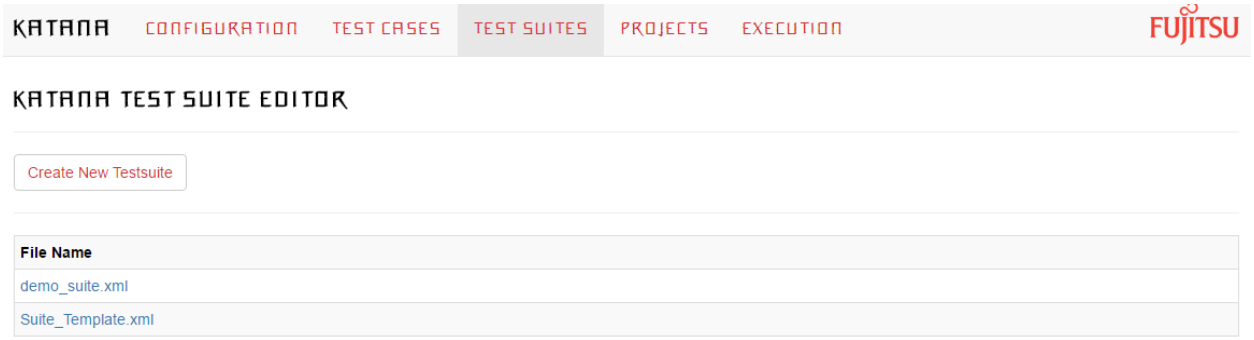


The testcase thus created would look something like this:

```
▼<Testcase>
  ▼<Details>
     <Name>Testcase_01</Name>
     <Title>Testcase_01_Title</Title>
     <Engineer>Jo Smith</Engineer>
     <Date>2016-05-18</Date>
     <Time>13:25</Time>
     <State>Test-Assigned</State>
     <InputDataFile>../Data/Sample_Input_Data_file.xml</InputDataFile>
     <Datatype>Iterative</Datatype>
     <default_onError action="goto" value="5"/>
     <Logsdir>/home/jSmith/Warrior_Logs</Logsdir>
     <Resultsdir>/home/jSmith/Warrior_Results</Resultsdir>
     <ExpectedResults>PASS</ExpectedResults>
   </Details>
  ▼<Steps>
   ▼<step Driver="cse_driver" Keyword="execute_cse_script" TS="1">
    ▼<Arguments>
       <argument/>
     </Arguments>
     <onError action="next"/>
     <Description>Execute a CSE's script from Warrior framework</Description>
     <iteration_type type=""/>
     <Execute ExecType="Yes"/>
     <context>positive</context>
     <impact>impact</impact>
     <runmode type="RMT" value="8"/>
   </step>
  </Steps>
  ▼<Requirements>
     <Requirement>Requirement-001</Requirement>
   </Requirements>
</Testcase>
```

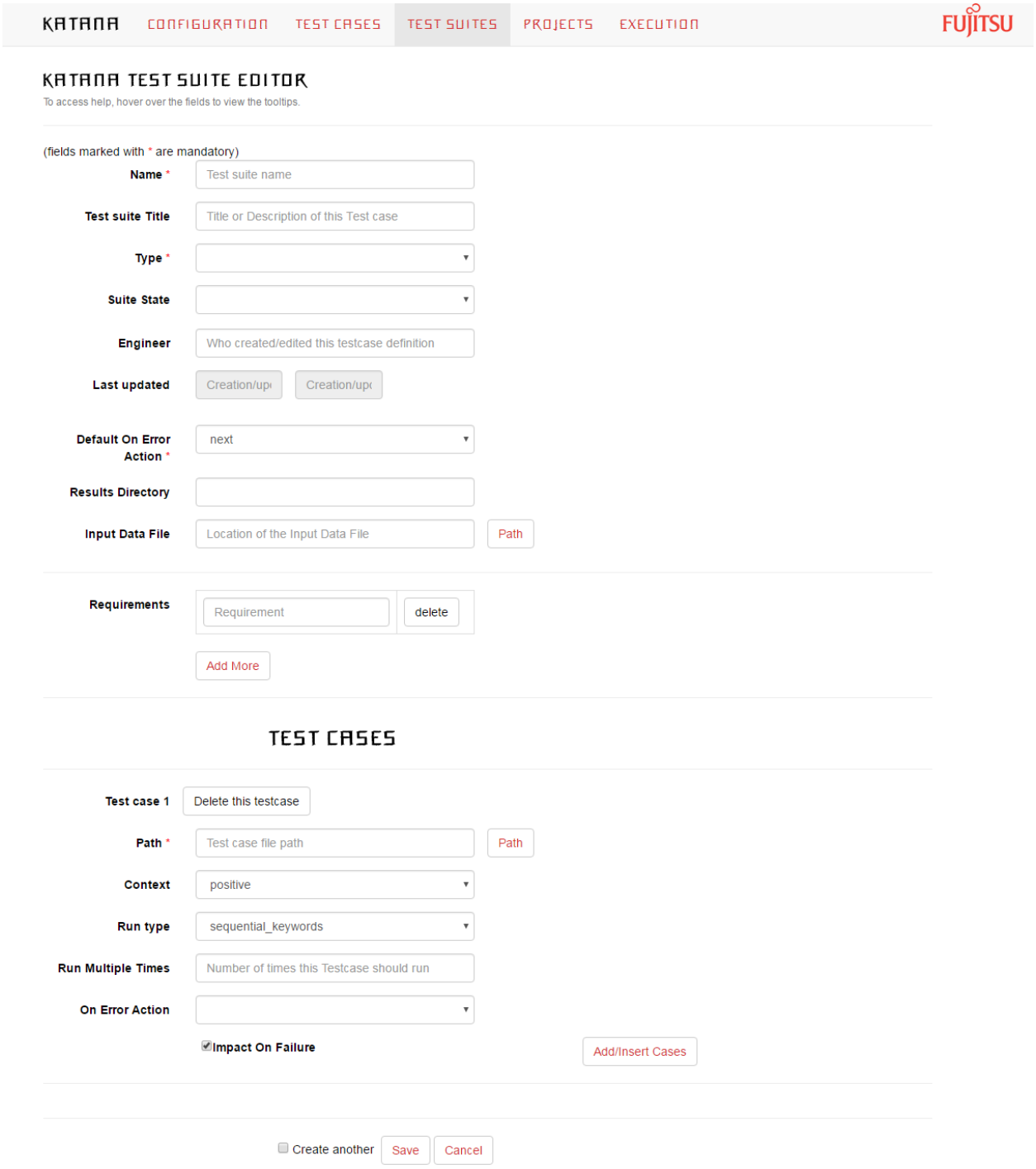
4. CREATING TESTSUITES WITH KATANA

Katana allows you to create Testsuites through its user interface. You have to start up Katana and go to its Testsuites tab. There, you will see a “New Testsuite” button:



Note: The list of testsuites below the “New Testsuite” button may differ as Katana would show the Testsuites saved in your testsuite directory.

On clicking that button, you will be directed to a new page:

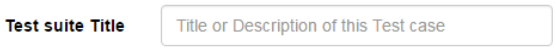


A step by step guide to create a testsuite is Katana is given below:

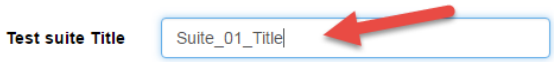
1. Name

Here enter the name of the testsuite. A name can be anything that you recognize. This is a mandatory field.

2. Testsuite Title

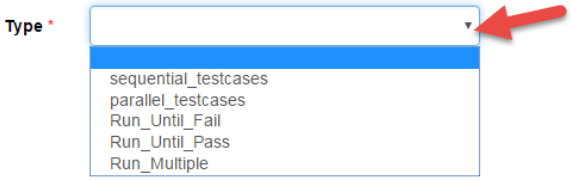


A testsuite Title (or description) is a field where you can add a descriptive title to identify what this testsuite does. This is a mandatory field.



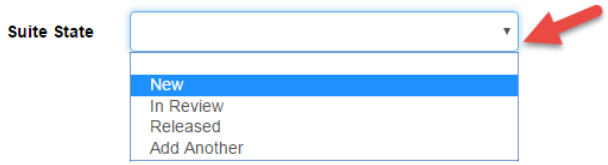
3. Type

This is a mandatory field that lets you determine how the suite should run. You can click on the dropdown and select the 'type' of the testsuite.



4. Suite State

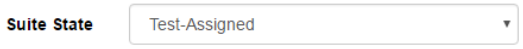
A suite state is informative when you have to keep track of which suites are new, released, or in review. These are the in-build categories that are available to you. This is an optional field, but it is recommended that you fill out this field so as to keep track of which suite is in which state of development.



You can select any state of your choice, like “New”. This would mark your suite as “New” meaning that this is a newly created suite. Furthermore, if the available categories do not really fit your suite state, you can go ahead and select “Add Another”. This opens up another field which lets you add a State of your own.

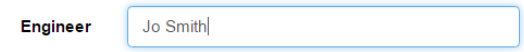
Here you can add in any testcase state that suits the state in which your testcase is, such as “Test-Assigned”, and click “Done”.

Your testcase now has a newly created “Test-Assigned” state. You can also use this newly created states for any other Testcase that you may create after this.



5. Engineer

This field indicates the name of the Engineer who created this Testcase. This is a prefilled field but, you can change it if you want to. This is a mandatory field.

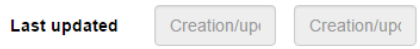


6. Results Directory

This field lets you specify the path to the Results directory – this directory is where all the results returned by Warrior would be stored. This is an optional field.

7. Last Updated (Date-Time stamp)

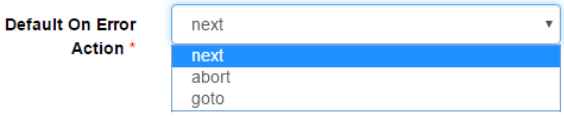
This field indicates the Date and Time on which this Testcase was last updated. This is a prefilled field, but the contents cannot be changed.



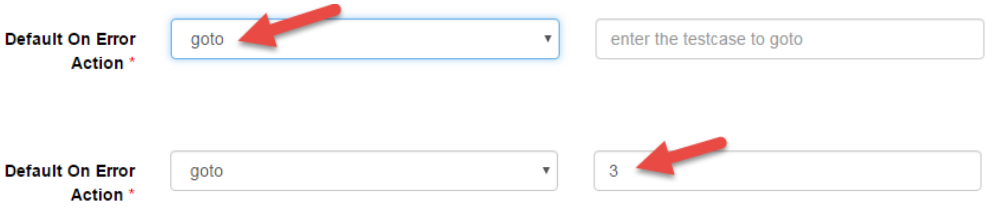
8. Default on Error

This is field where you can specify what the Testsuite should do it case it errors out. By default, it is “next”, that is, if you don’t make a change, the Testsuite will proceed to the next Testsuite available for execution if the current testsuite throws an error.

Clicking on the dropdown shows all the options that are available to you to tell Warrior what to do in the case where the current Testcase errors out. You have the ability to choose any one of them.

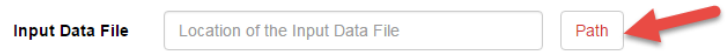


On selecting “goto”, another pops up and lets you add the Testcase number that Warrior should go to in case of an error.

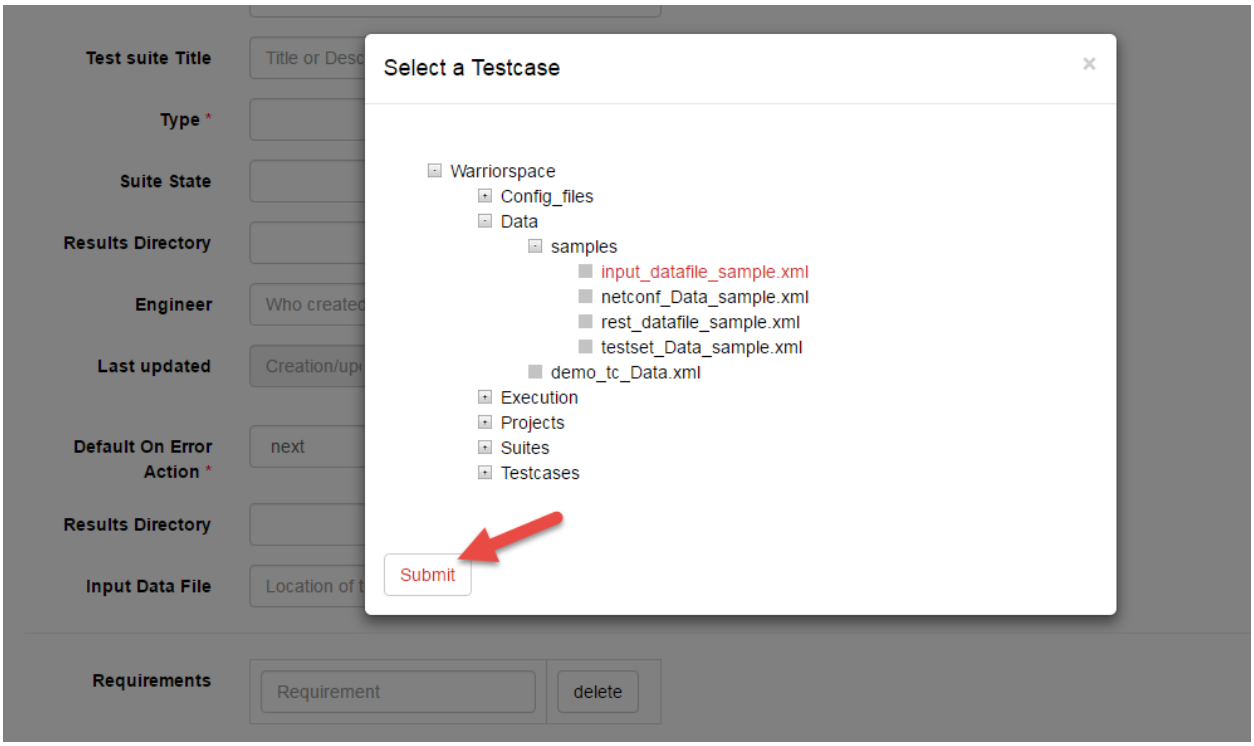


9. Input Data File

This is field wherein you can add the path to a datafile (Warrior User Guide: Section 9.0) that you want this testcase to use. You can click on the 'Path' button to choose a path

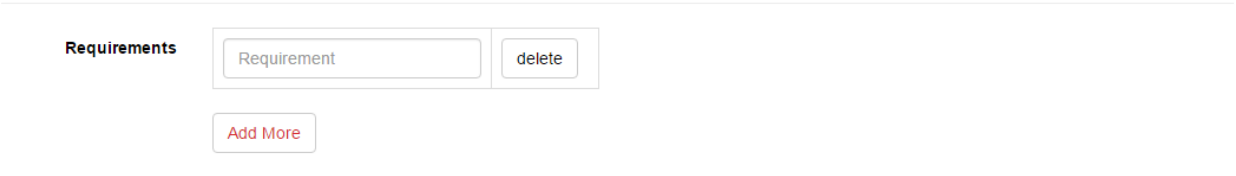


Once you choose the Datafile, you can click on 'Submit' to enter that datafile path.

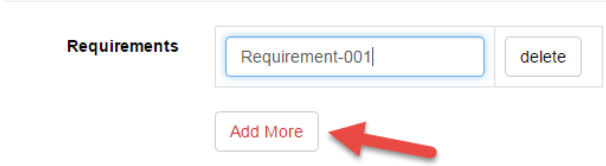


Or you can simply type in a path to your datafile.

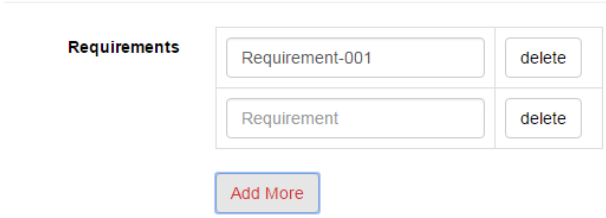
10. Requirements



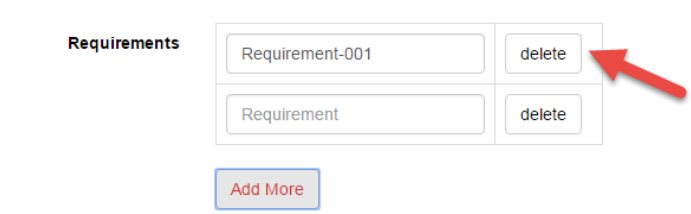
This field is for you add any requirements that this testsuite may have. This is an optional field. You can add a new requirement by clicking on the “Add More” button.



This opens up a new field in which you can specify the requirement.



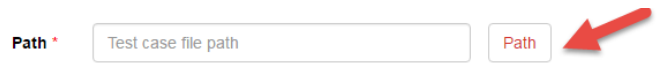
You can also delete an added requirement by clicking the ‘delete’ button against the Requirement that you want to delete.

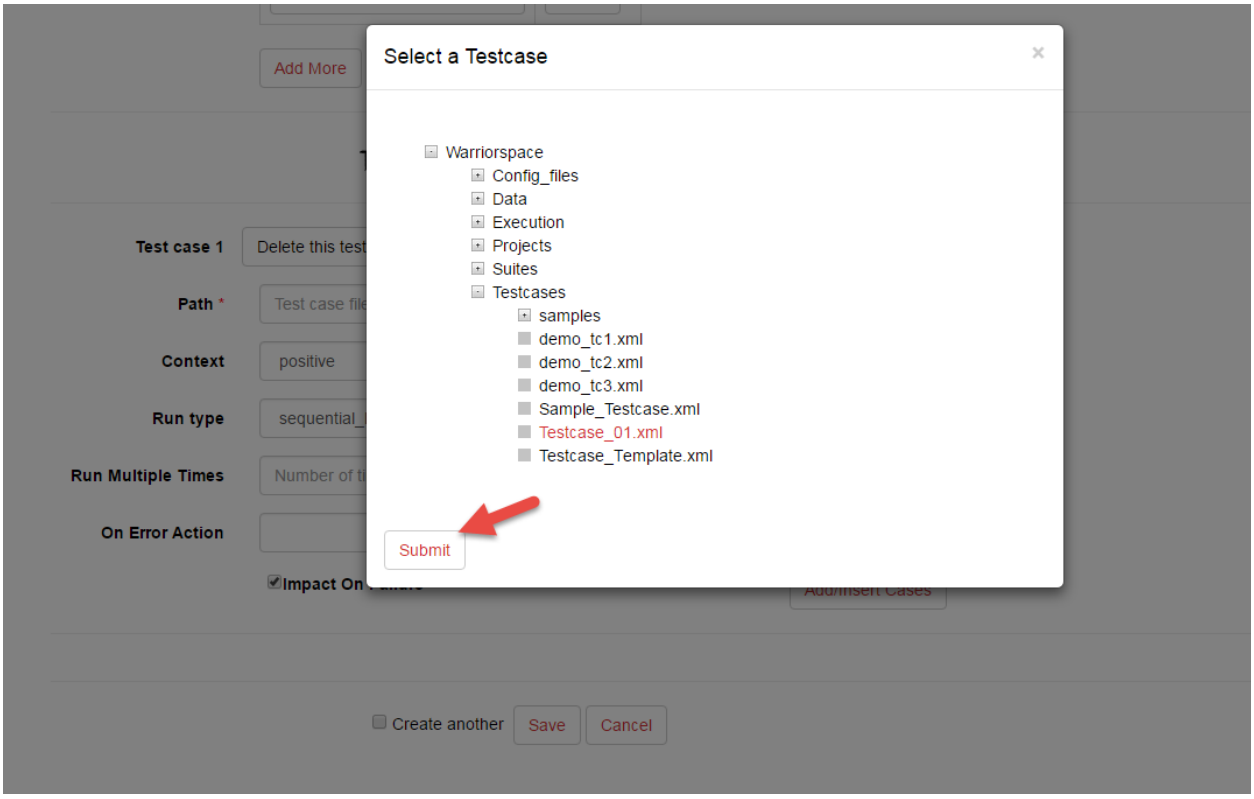


After this point, every input is related to the testcase:

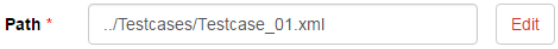
11. Path

This is the first field in testcase part of the Testsuite creation. You can add a path to the testcase by clicking on the ‘Path’ button.





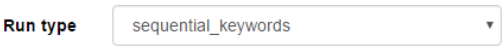
Then, you will be able to select a path to your testcase, and to enter this path you can hit 'Submit'.

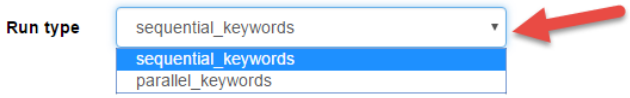


You can also manually add a path to your testcase. This is a mandatory field.

12. Context

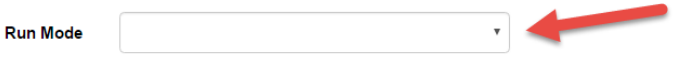
13. Run type



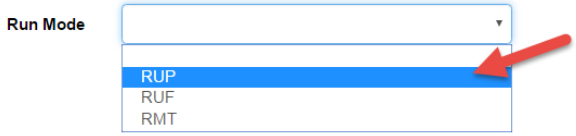


14. Run Mode

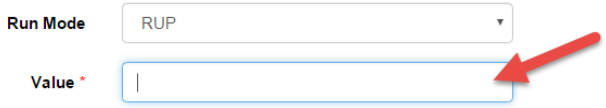
This field lets you select if you want this testcase to run until pass (RUP), run until failure (RUF), or run multiple times (RMT). This is an optional field.



The Run Mode field is dropdown consisting of values “RUP”, “RUF”, and “RMT”.



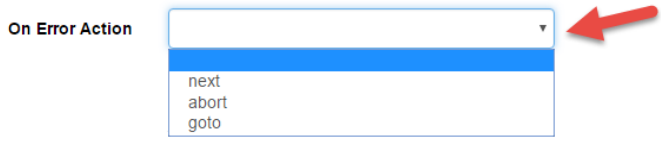
Whenever a value from the dropdown is selected, and new field “Value” pops up that lets you fill in the maximum number of times you want to run this testcase – if RUP or RUF is selected or the number of times you want to run this testcase if RMT is selected.



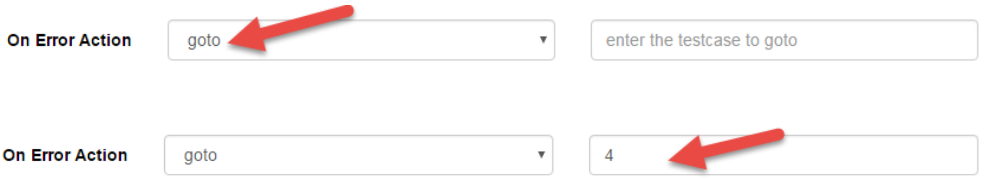
You can input the number of times you want this particular testcase to run here.

15. On Error Action

This tag handles the Error condition. You can click on the dropdown to choose an option.

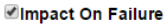


If goto is selected, then, another input box appears that lets you add the number of testcase to which you want to jump in case of failure.

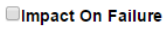


16. Impact on Failure

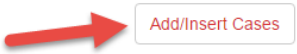
The Impact on Failure checkbox is checked by default.



You can uncheck it if you want the result of this particular testcase to not affect the entire suite.

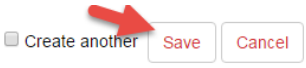


17. Add/Insert Cases

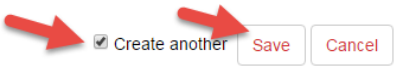


18. Save

Clicking on the 'Save' button saves this testsuite.

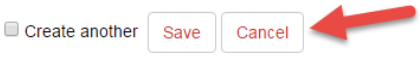


Checking the 'Create Another' checkbox, lets you save the testsuite and then open a new fresh testsuite page for you to start creating a new testsuite.



19. Cancel

On the other hand, clicking on 'Cancel' discards all the changes made to this testsuite.

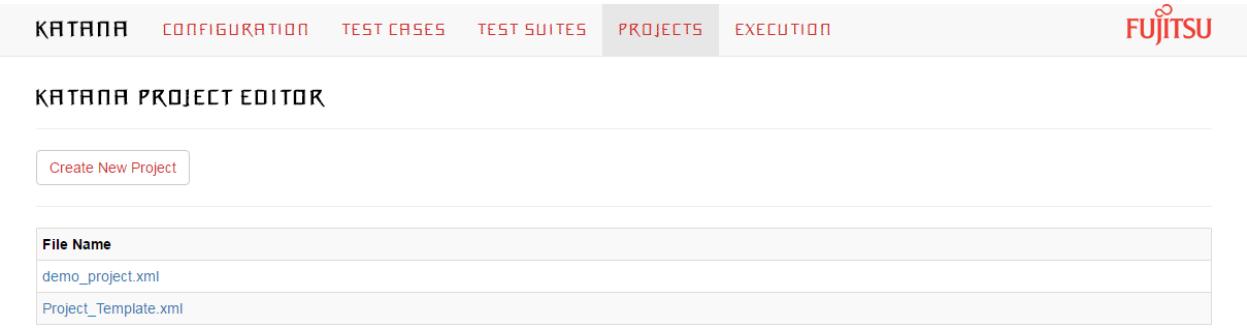


The testsuite thus created looks something like this:

```
▼<TestSuite>
  ▼<Details>
     <Name>Suite_01</Name>
     <Title>Suite_01_Title</Title>
     <Engineer>Jo Smith</Engineer>
     <Date>05/18/2016</Date>
     <Time>15:33:57</Time>
     <type exectype="sequential_testcases"/>
     <State>Test-Assigned</State>
     <default_onError action="goto" value="3"/>
     <Resultsdir>/home/jSmith/Warrior_Results</Resultsdir>
     <IDF>../Data/samples/input_datafile_sample.xml</IDF>
   </Details>
  ▼<Requirements>
     <Requirement>Requirement-001</Requirement>
     <Requirement/>
   </Requirements>
  ▼<Testcases>
    ▼<Testcase>
       <path>../Testcases/Testcase_01.xml</path>
       <context>positive</context>
       <runtype>sequential_keywords</runtype>
       <onError action="goto" value="4"/>
       <runmode type="RUP" value="5"/>
       <impact>impact</impact>
     </Testcase>
   </Testcases>
 </TestSuite>
```

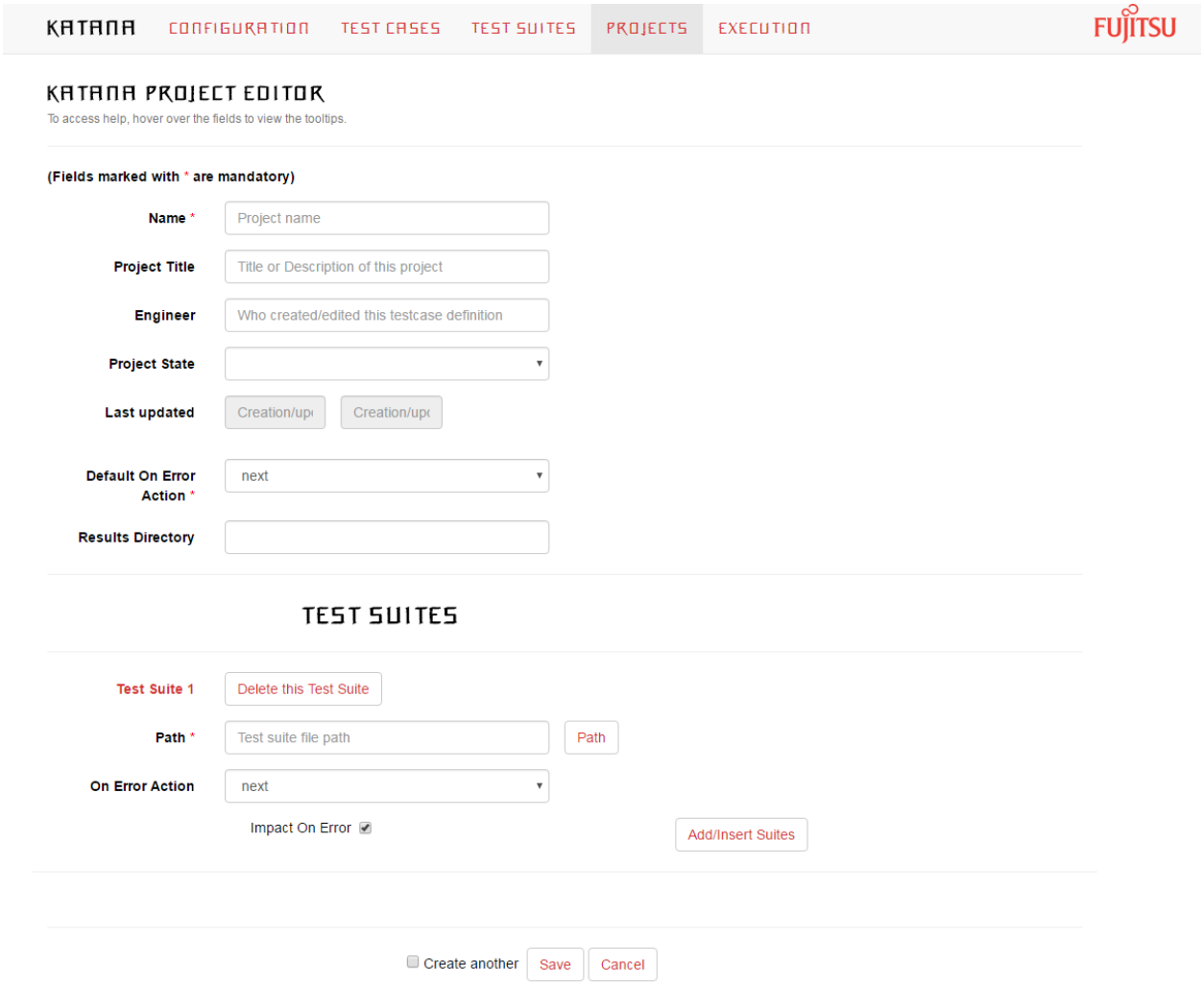
5. CREATING PROJECTS WITH KATANA

Katana allows you to create Projects through its user interface. You have to start up Katana and go to its Projects tab. There, you will see a “Create New Project” button:



Note: The list of projects below the “Create New Project” button may differ as Katana would show the Projects saved in your project directory.

On clicking that button, you will be directed to a new page:

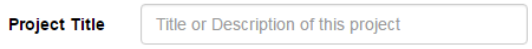


A step by step guide to create a project in Katana is given below:

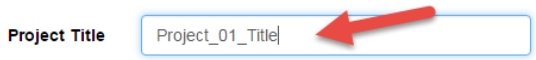
1. Name

Here enter the name of the project. A name can be anything that you recognize. This is a mandatory field.

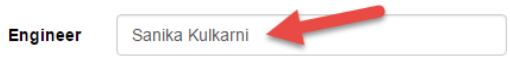
2. Project Title



A testsuite Title (or description) is a field where you can add a descriptive title to identify what this testsuite does. This is a mandatory field.



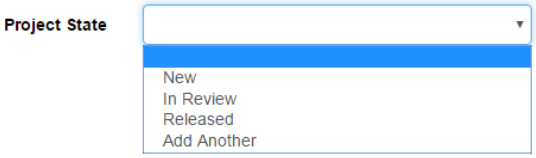
3. Engineer



This field indicates the name of the Engineer who created this Testcase. This is a prefilled field but, you can change it if you want to. This is a mandatory field.

4. Project State

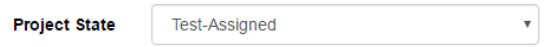
A project state is informative when you have to keep track of which projects are new, released, or in review. These are the in-build categories that are available to you. This is an optional field, but it is recommended that you fill out this field so as to keep track of which project is in which state of development.



You can select any state of your choice, like “New”. This would mark your suite as “New” meaning that this is a newly created project. Furthermore, if the available categories do not really fit your project state, you can go ahead and select “Add Another”. This opens up another field which lets you add a State of your own.

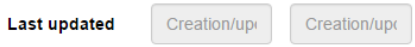
Here you can add in any project state that suits the state in which your project is, such as “Test-Assigned”, and click “Done”.

Your project now has a newly created “Test-Assigned” state. You can also use this newly created states for any other Project that you may create after this.



5. Last Updated (Date-Time stamp)

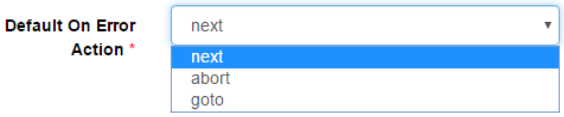
This field indicates the Date and Time on which this Testcase was last updated. This is a prefilled field, but the contents cannot be changed.



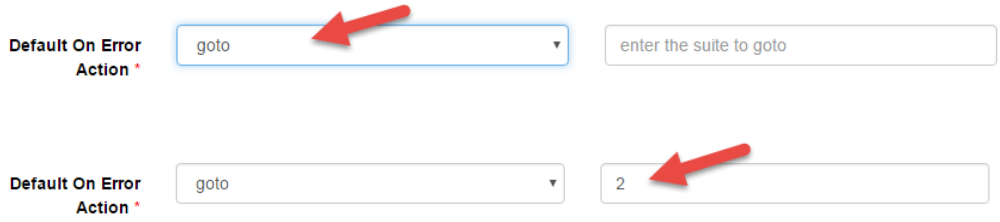
6. Default on Error

This is field where you can specify what the Project should do it case it errors out. By default, it is “next”, that is, if you don’t make a change, the Project will proceed to the next Project available for execution if the current testsuite throws an error.

Clicking on the dropdown shows all the options that are available to you to tell Warrior what to do in the case where the current Project errors out. You have the ability to choose any one of them.



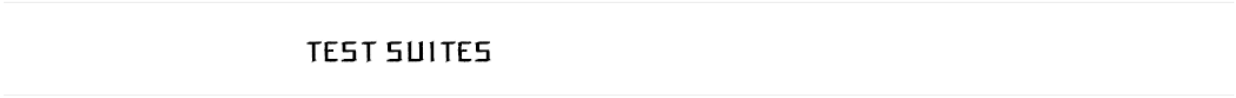
On selecting “goto”, another pops up and lets you add the Project number that Warrior should go to in case of an error.



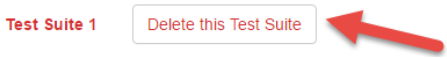
7. Results Directory

This field lets you specify the path to the Results directory – this directory is where all the results returned by Warrior would be stored. This is an optional field.

Beyond this, all the inputs refer to the testsuites that would be included in the project.



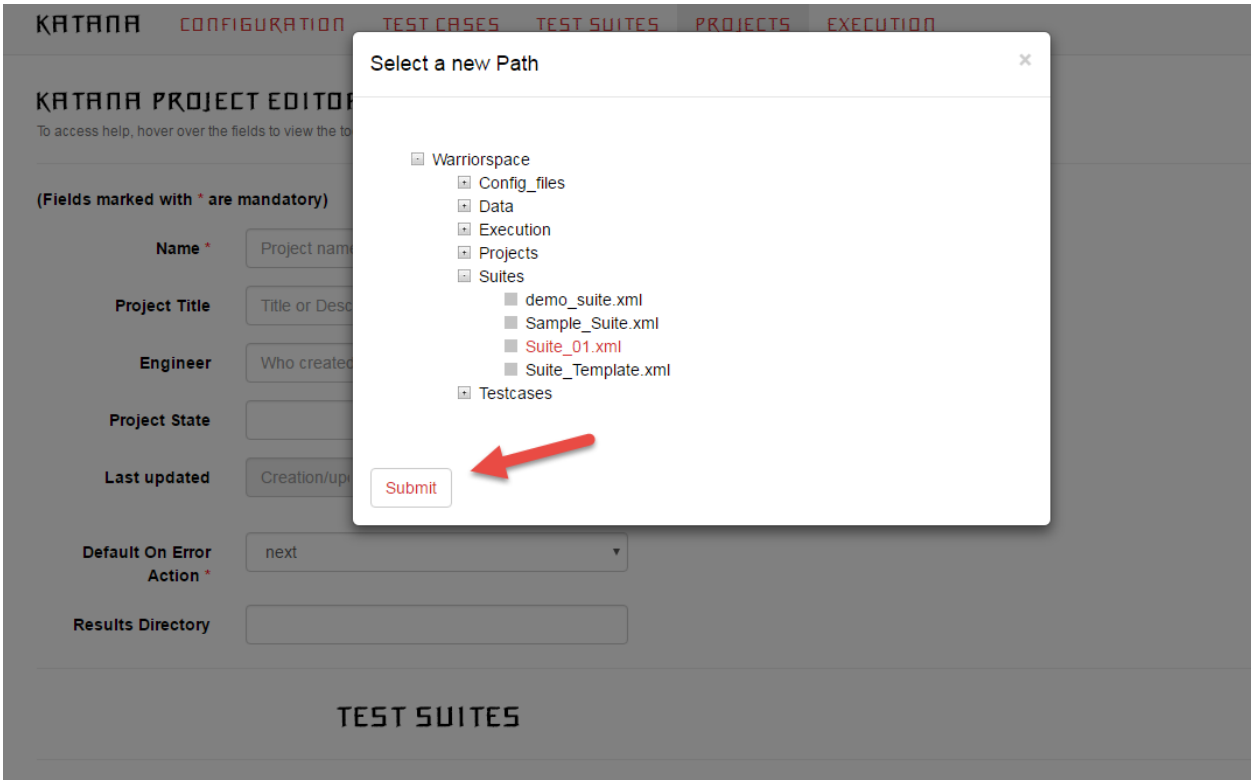
8. Delete this Test Suite



This button lets you delete a Test suite that has been added to the project. But the Project needs at least one testsuite in it.

9. Path

This is the first field in testcase part of the Testsuite creation. You can add a path to the testcase by clicking on the 'Path' button.

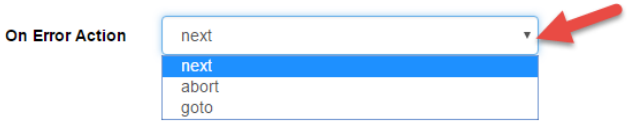


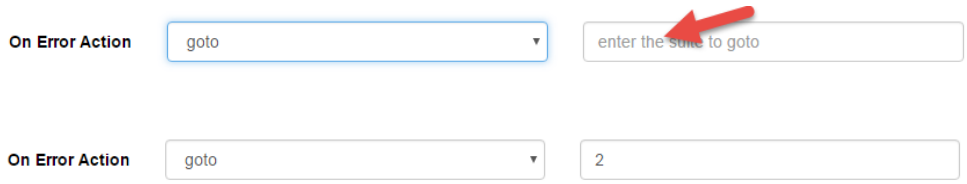
Then, you will be able to select a path to your testcase, and to enter this path you can hit 'Submit'.

You can also manually add a path to your testcase. This is a mandatory field.

10. On Error Action

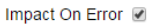
This tag handles the Error condition. You can click on the dropdown to choose an option. If goto is selected, then, another input box appears that lets you add the number of testsuite to which you want to jump in case of failure.



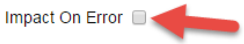


11. Impact on Failure

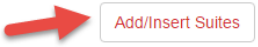
The Impact on Failure checkbox is checked by default.



You can uncheck it if you want the result of this particular testcase to not affect the entire suite.



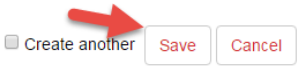
12. Add/Insert Suites



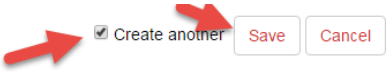
Clicking on this button lets you add another testsuite to this project.

12. Save

13. Clicking on the 'Save' button saves this project.

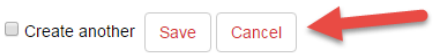


Checking the 'Create Another' checkbox, lets you save the project and then open a new fresh project page for you to start creating a new project.



14. Cancel

15. On the other hand, clicking on 'Cancel' discards all the changes made to this project.

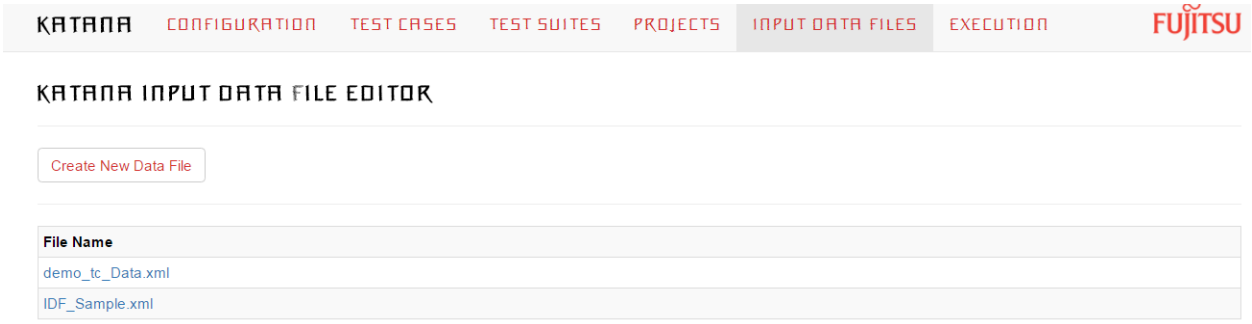


The Project thus created looks something like this:

```
▼<Project>
  ▼<Details>
     <Name>Project_01</Name>
     <Title>Project_01_Title</Title>
     <Engineer>Jo Smith</Engineer>
     <State>Test-Assigned</State>
     <Date>05/31/2016</Date>
     <Time>18:03:53</Time>
     <default_onError action="next"/>
   </Details>
  ▼<Testsuites>
    ▼<Testsuite>
       <path>../Suites/Suite_01.xml</path>
      ▼<Execute ExecType="Yes">
         <Rule Condition="" Condvalue="" Else="next" Elsevalue=""/>
       </Execute>
       <onError action="goto" value="5"/>
       <impact>impact</impact>
     </Testsuite>
   </Testsuites>
 </Project>
```

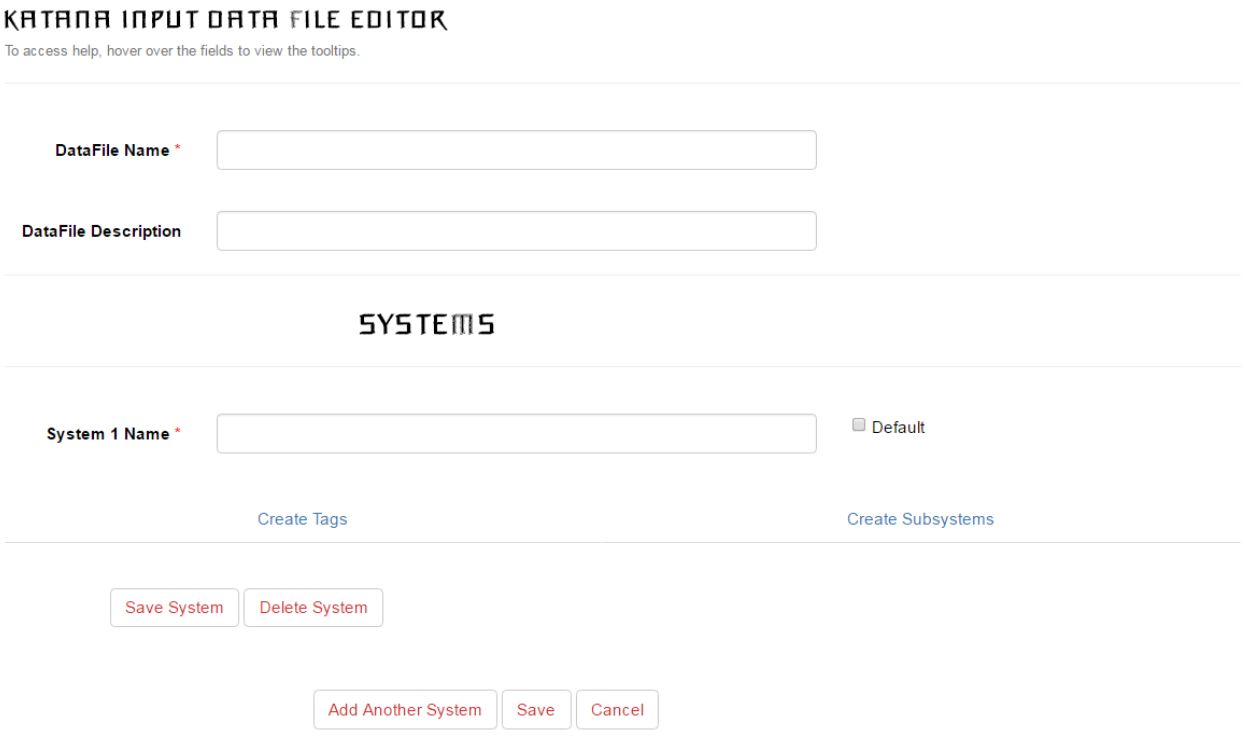
6. CREATING INPUT DATA FILES WITH KATANA

Katana allows you to create Input Data Files through its user interface. You have to start up Katana and go to its Input Data Files tab. There, you will see a “Create New Data File” button:



Note: The list of input data files below the “Create New Data Files” button may differ as Katana would show the input data files saved in your Data directory.

On clicking that button, you will be directed to a new page:



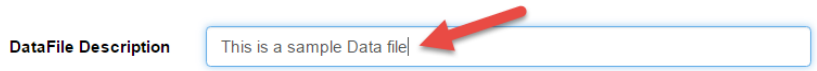
A step by step guide for creating an input data file in Katana is given below:

1. DataFile Name

This field represents the name of the file that you will be creating. Type in a name for the input data file here

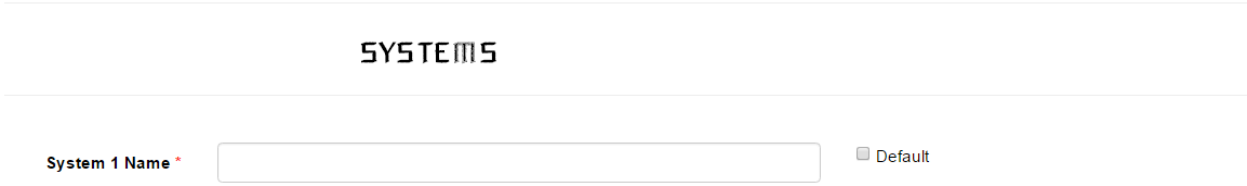
2. DataFile Description

You can add a description for the data file here.



3. Systems

You can start creating systems inside this Data file



3.1. System Name

You can add the name of the system in this input field

3.2. Default System

You can set any system as the default system by checking the “Default” checkbox.

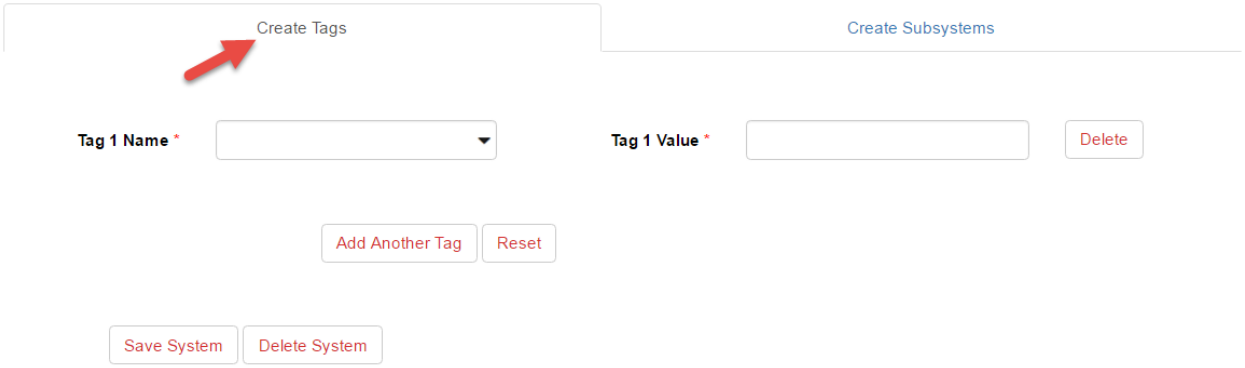
If no system has been set as default then the first system would be treated as the default.

3.3. Create Tags and Subsystems

For any system, you can either create Tags for it or subsystems, but not both.

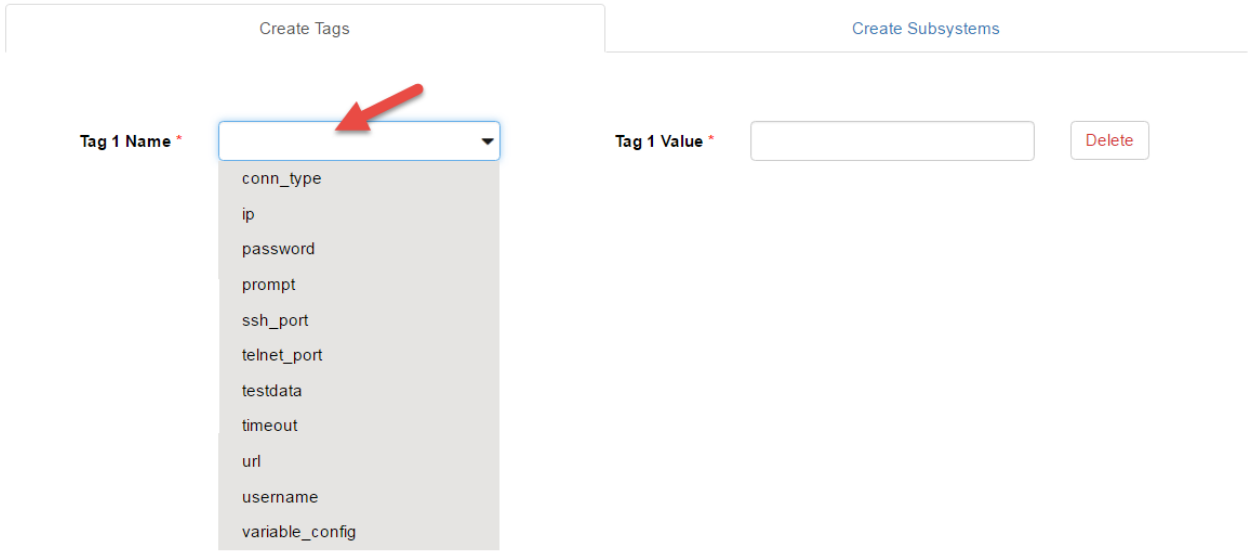
3.3.1. Creating Tags

Clicking on the “Create Tags” would display input boxes and other options to create the tags.

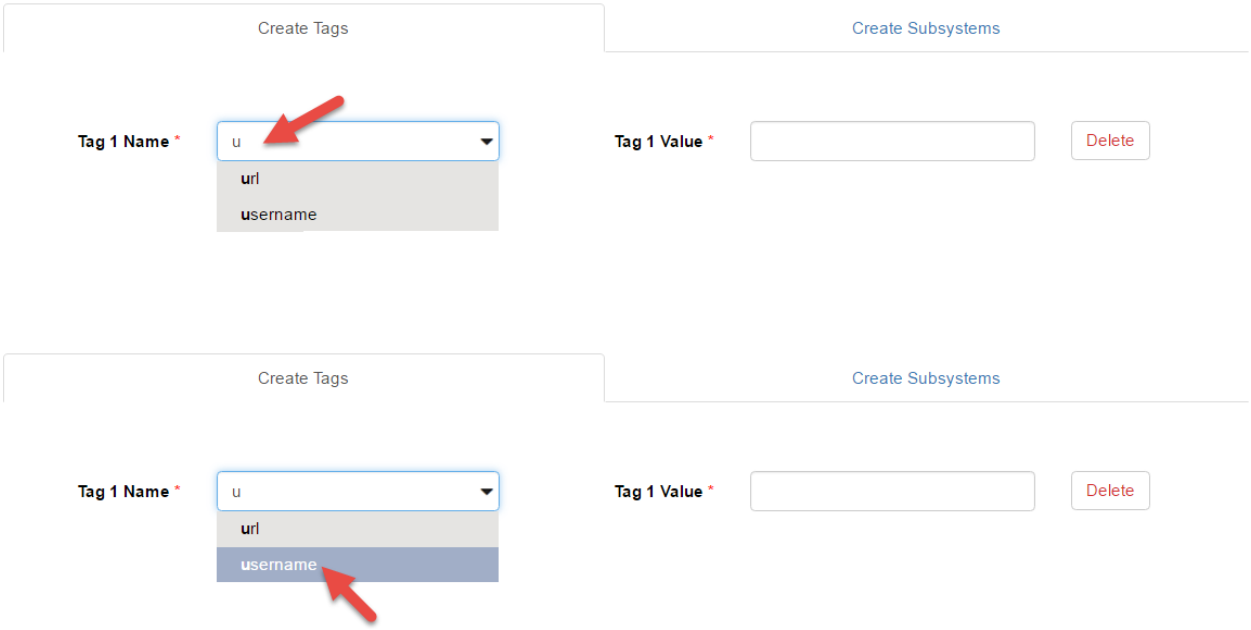


3.3.1.1. Tag Name

Since, the input data file does not dictate as to which tags should be created for each system, you have to add the tag names that you want. Immediately, on clicking inside the Tag Name input box, you would be able to view all the tag names that are already available to choose from.



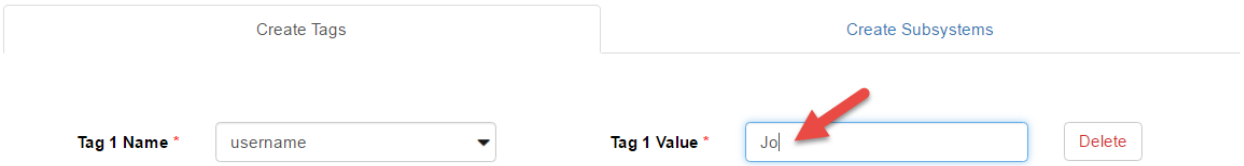
If the tag name already exists in that list, you can select it from the dropdown list; or else you can add a tag name that you want.



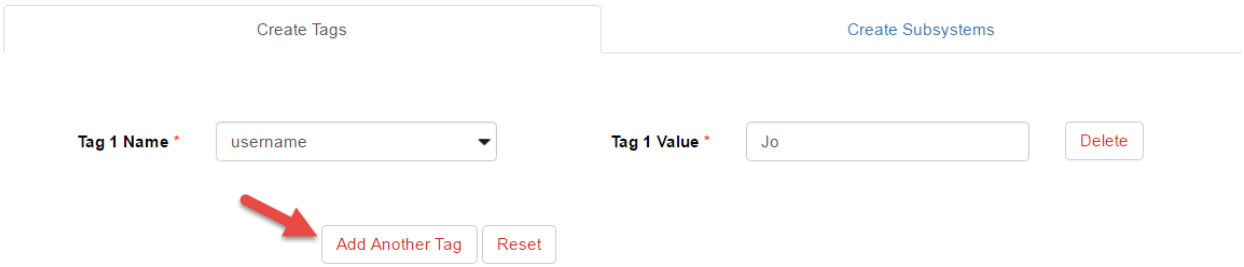
This is a mandatory field.

3.3.1.2. Tag Value

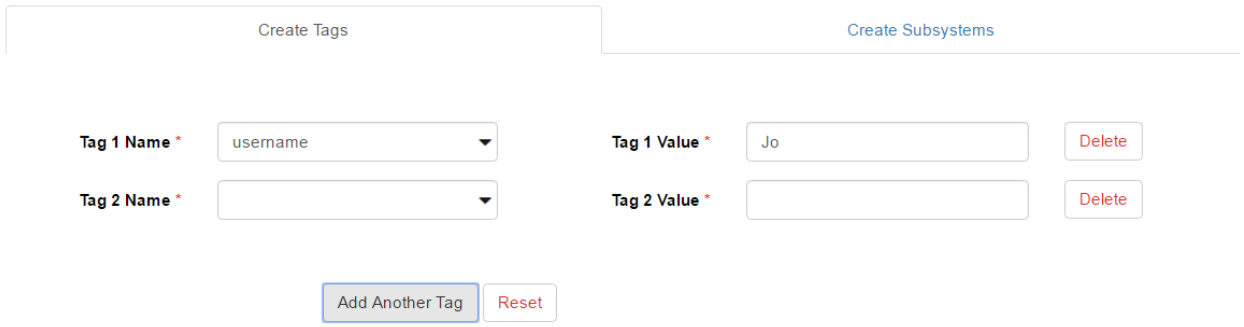
A corresponding value to the tag created above can be added here.



3.3.1.3. Add Another Tag



Clicking the "Add Another Tag" lets you create another tag for the same system.

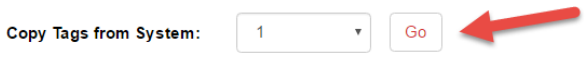


3.3.1.4. Copy Tags from System

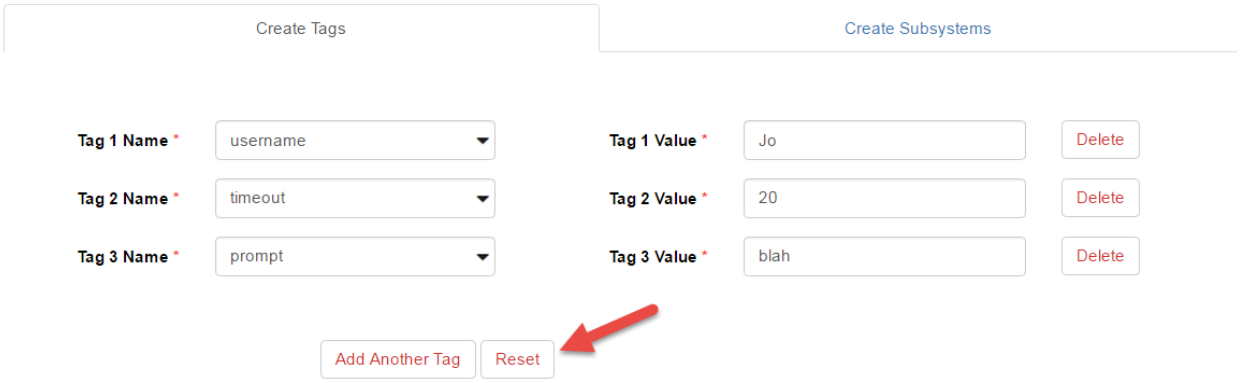
This functionality lets you copy tags over from another system into the current system. This functionality is visible only when there are more than two systems containing tags in that data file.

You can select the system number that you want to copy the tags from.

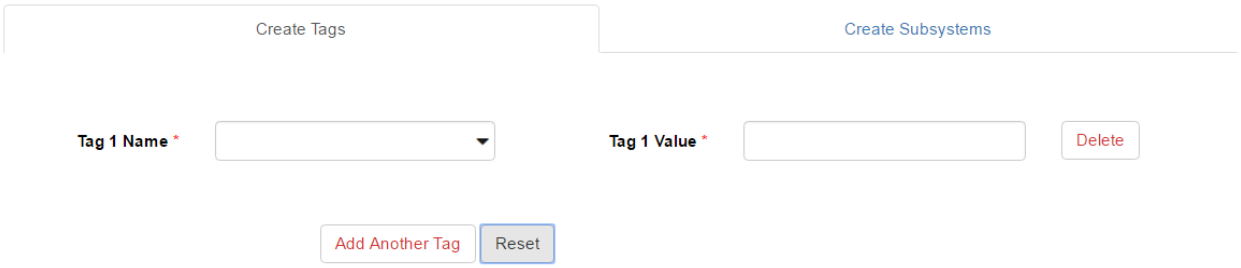
And then hit "Go" to copy over the tags.



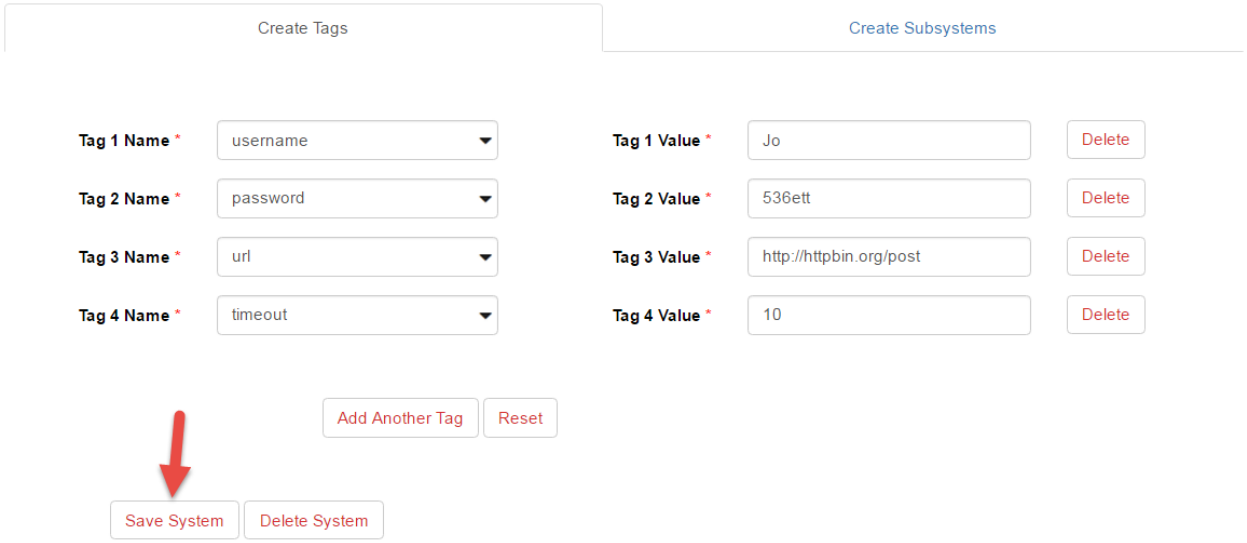
3.3.1.5. Reset



The “Reset” button resets the entire system tags for you.



3.3.1.6. Save System



Clicking on Save System, saves the system for you.

SYSTEMS

SYSTEMS

#	System Name	System Child-Tags/Attributes	Subsystem Details	Default	⊗
1	system1	username = Jo password = 536ett url = http://httpbin.org/post timeout = 10		Yes	⊗

3.3.1.7. Edit System

SYSTEMS

#	System Name	System Child-Tags/Attributes	Subsystem Details	Default	⊗
1	system1	username = Jo password = 536ett url = http://httpbin.org/post timeout = 10		Yes	⊗

You can click on a saved system name to edit that system.

System 1 Name * system1 Default

Create Tags | Create Subsystems

Tag 1 Name *	username	Tag 1 Value *	Jo	Delete
Tag 2 Name *	password	Tag 2 Value *	536ett	Delete
Tag 3 Name *	url	Tag 3 Value *	http://httpbin.org/post	Delete
Tag 4 Name *	timeout	Tag 4 Value *	10	Delete

Add Another Tag | Reset

Save System | Delete System

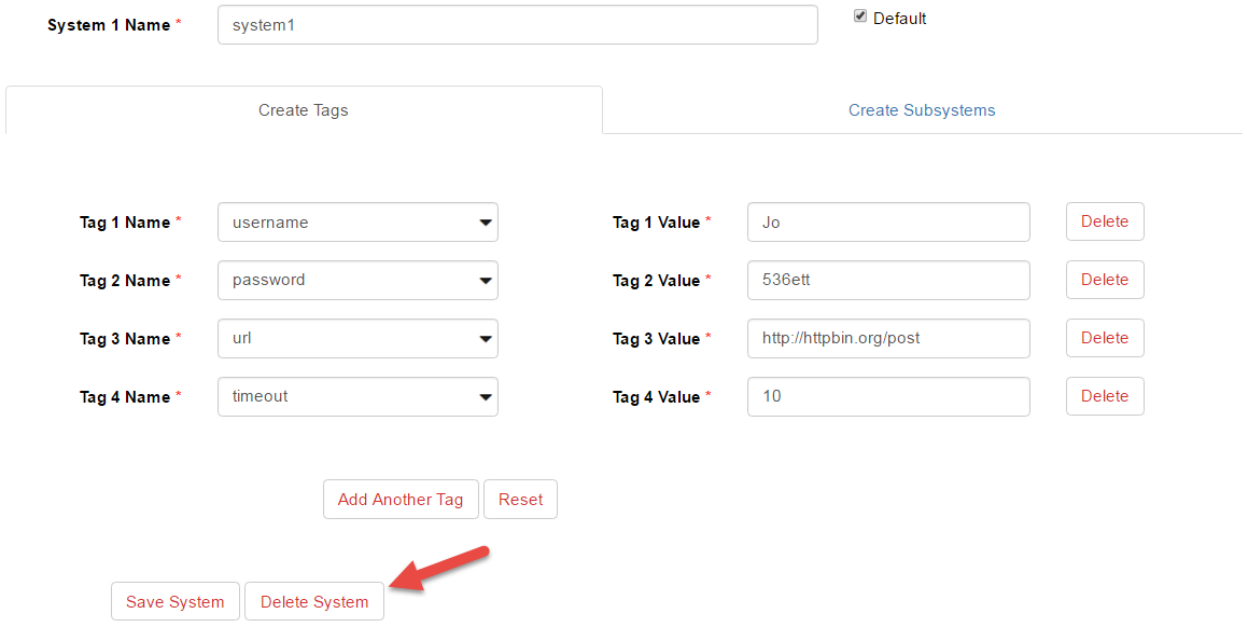
3.3.1.8. Delete System

You can delete a saved system by clicking on the delete icon next to it.

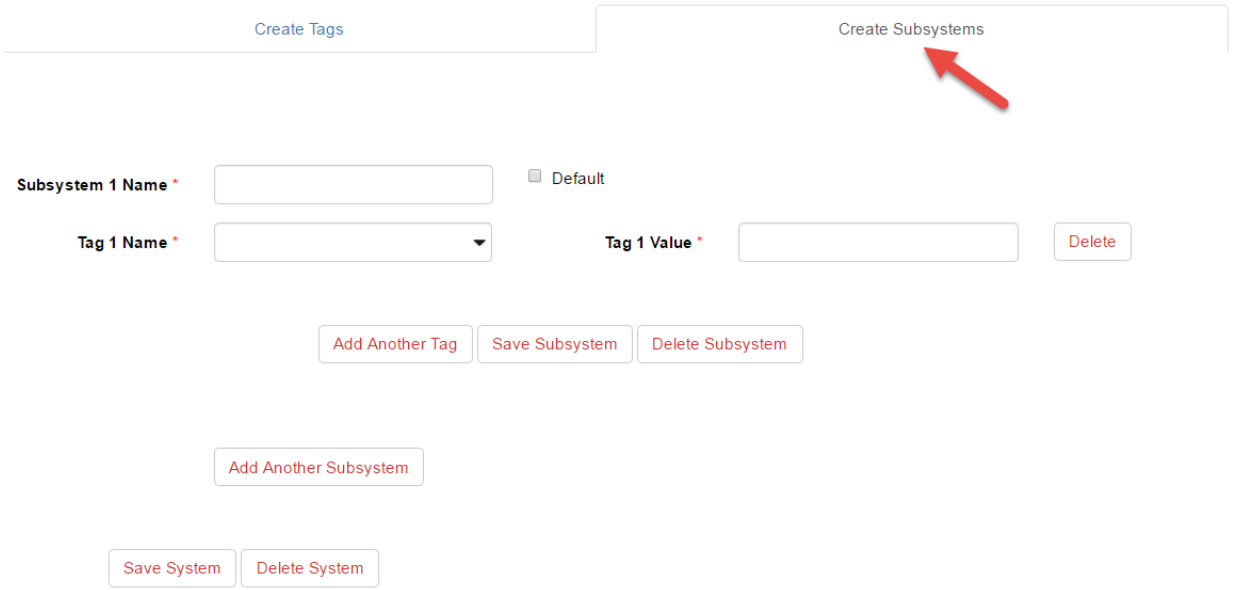
SYSTEMS

#	System Name	System Child-Tags/Attributes	Subsystem Details	Default	⊗
1	system1	username = Jo password = 536ett url = http://httpbin.org/post timeout = 10		Yes	⊗

You can delete an unsaved system by clicking on the “Delete System” button.



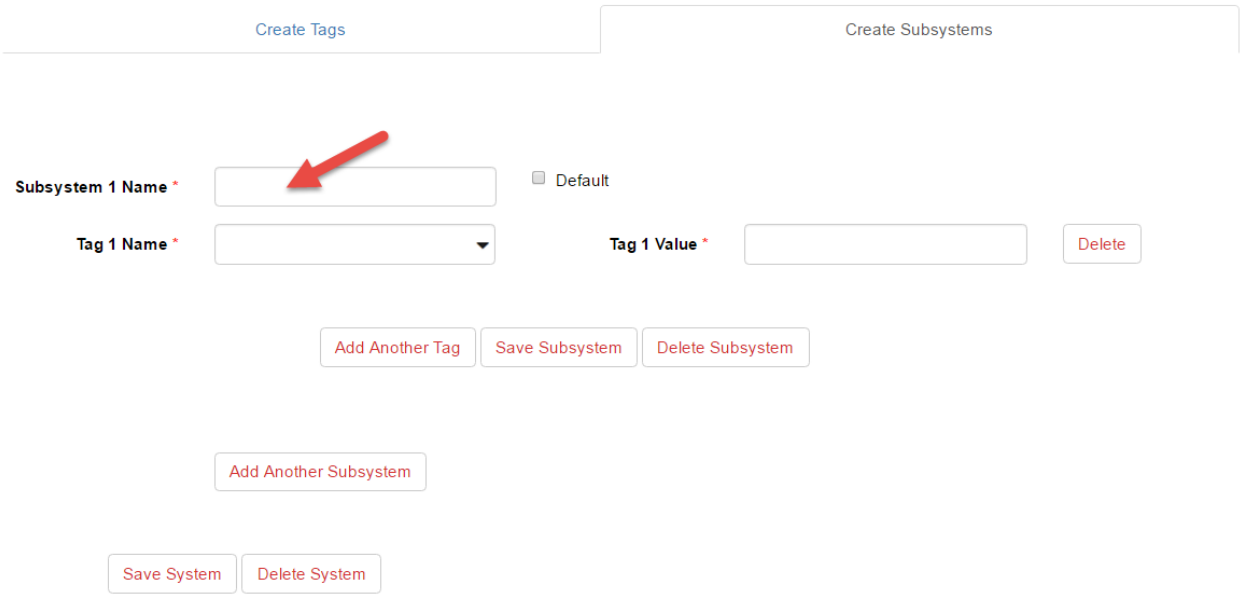
3.3.2. Create Subsystems



Clicking on the “Create Subsystems” tab lets you create subsystems for that System.

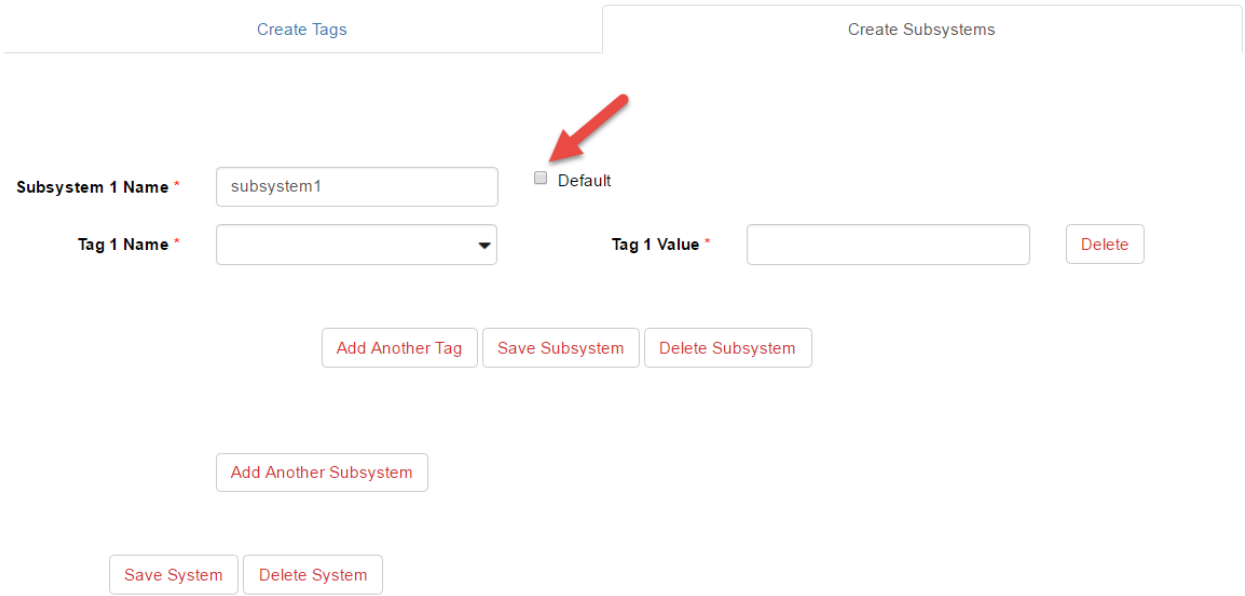
3.3.2.1. Subsystem Name

You can add a name to the subsystem in the “Subsystem Name” field. This is a mandatory field.



3.3.2.2. Default Subsystem

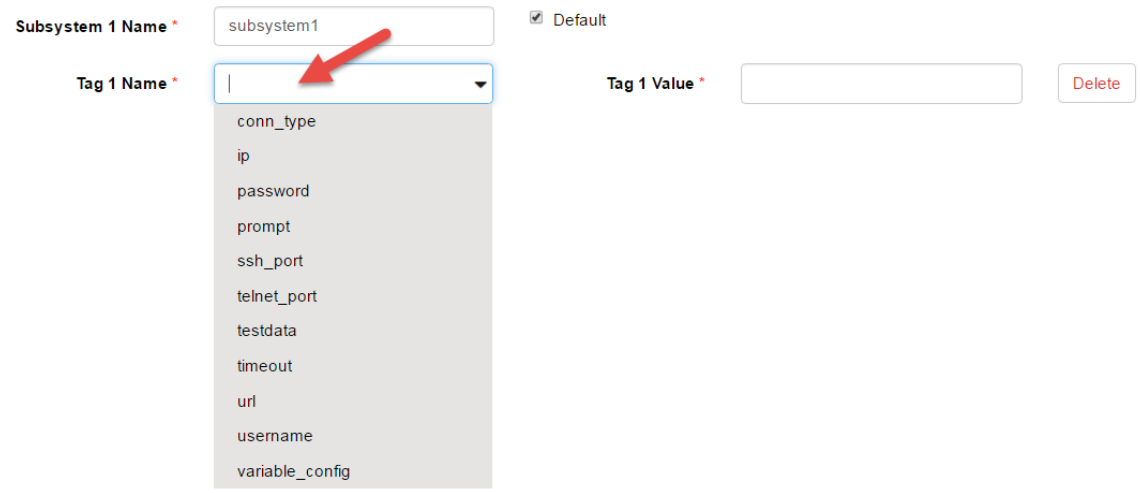
Each subsystem has an option to be set as Default. A subsystem can be set as “Default” by checking the Default checkbox against it.



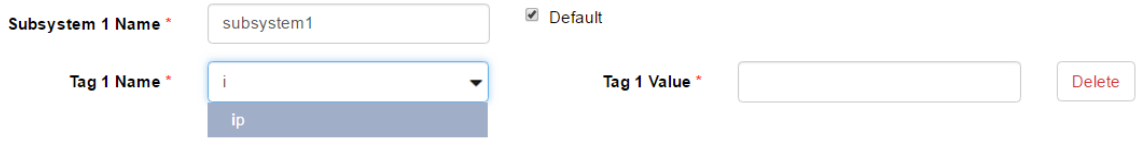
By default, the subsystem is always the default subsystem.

3.3.2.3. Tag Name

Since, the input data file does not dictate as to which tags should be created for each subsystem, you have to add the tag names that you want. Immediately, on clicking inside the Tag Name input box, you would be able to view all the tag names that are already available to choose from.



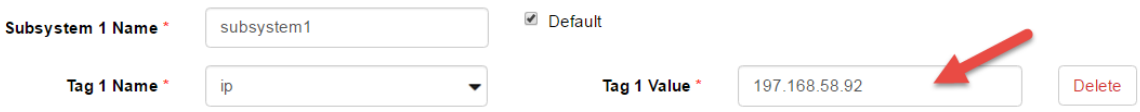
If the tag name already exists in that list, you can select it from the dropdown list; or else you can add a tag name that you want.



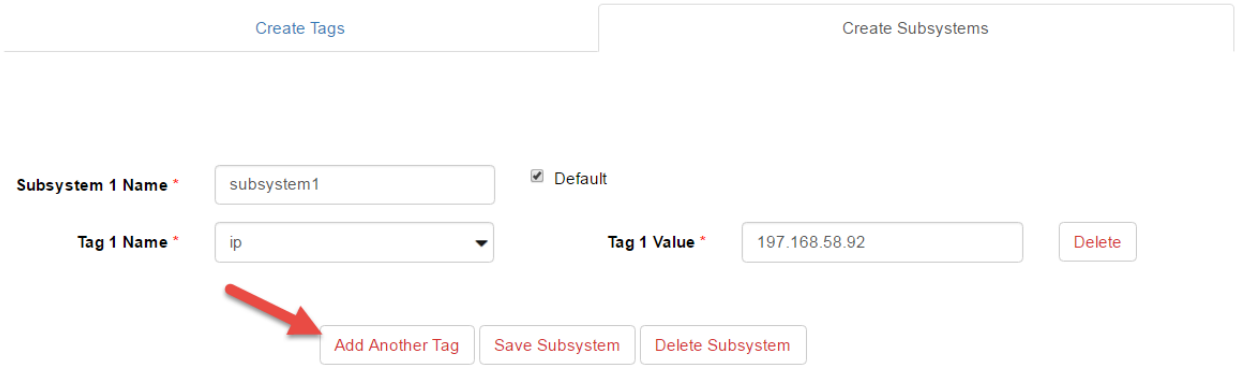
This is a mandatory field.

3.3.2.4. Tag Value

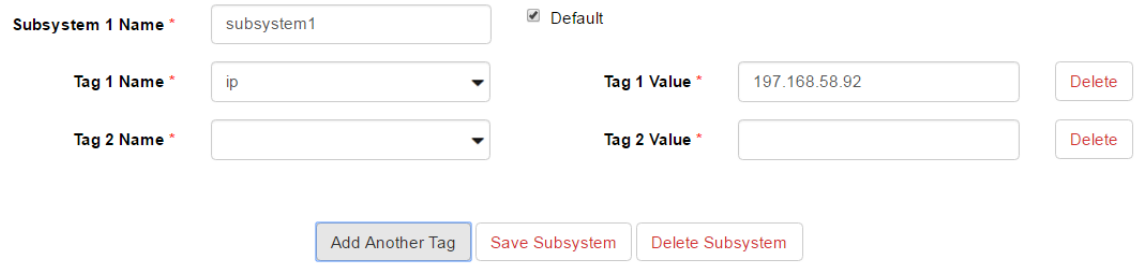
A corresponding value to the tag created above can be added here.



3.3.2.5. Add Another Tag



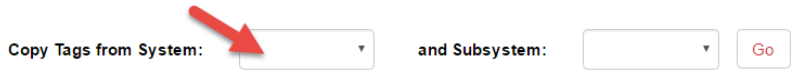
Clicking this button, lets you add another tag to the same subsystem.



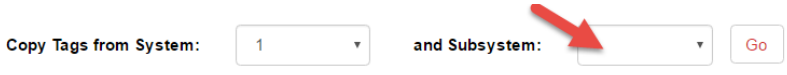
3.3.2.6. Copy Tags from Subsystem

This functionality lets you copy tags over from another subsystem into the current subsystem. This functionality is visible only when there are more than two systems containing subsystems in that data file.

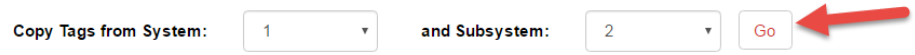
You can select the system number that contains the subsystem that you want.



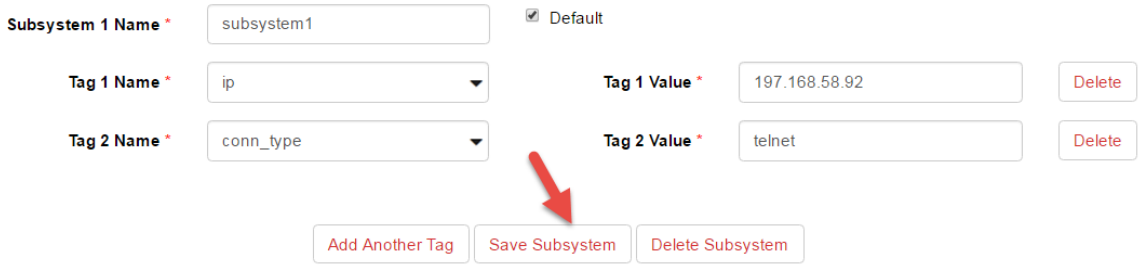
You can select the subsystem number that you want to copy the tags from.



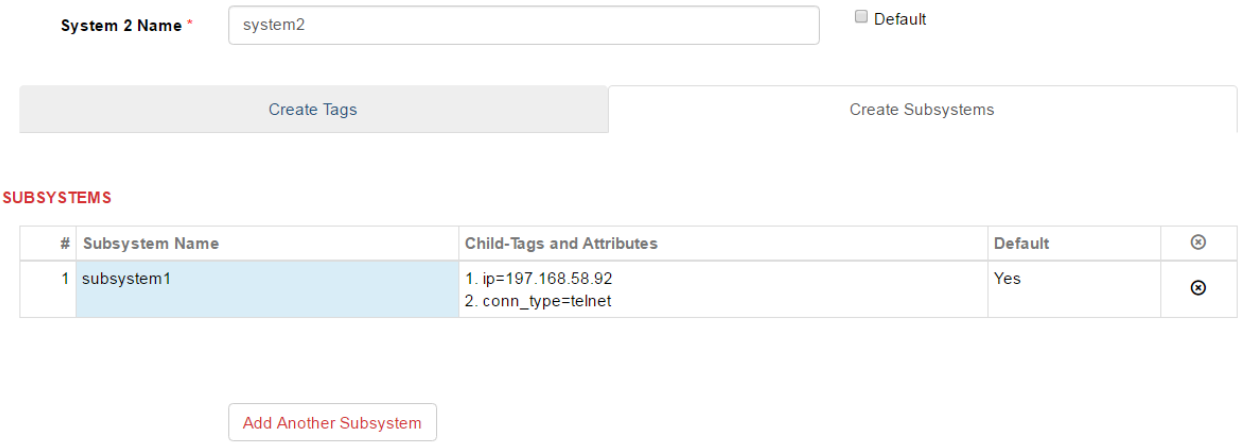
And then hit “Go” to copy over the tags.



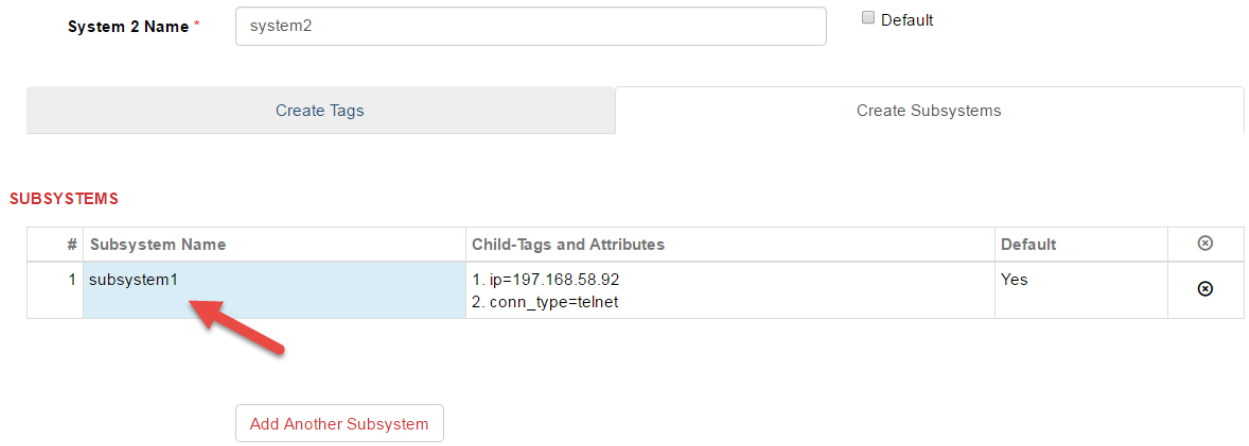
3.3.2.7. Save Subsystem



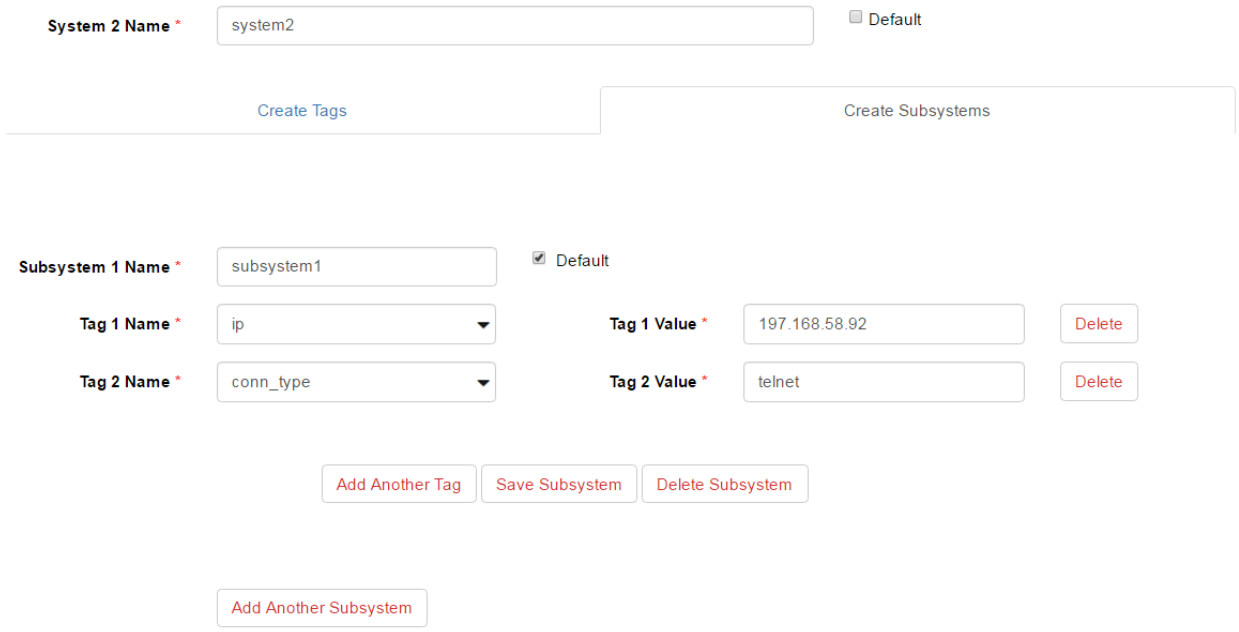
"Save Subsystem" saves the subsystem in Katana



3.3.2.8. Edit Subsystem

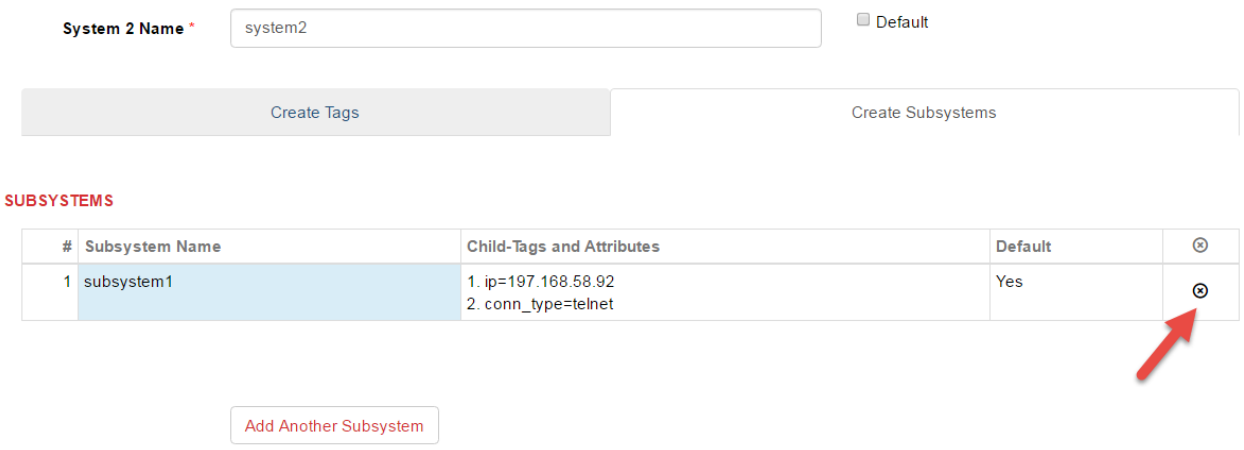


A saved subsystem can be edited by clicking on the subsystem name.

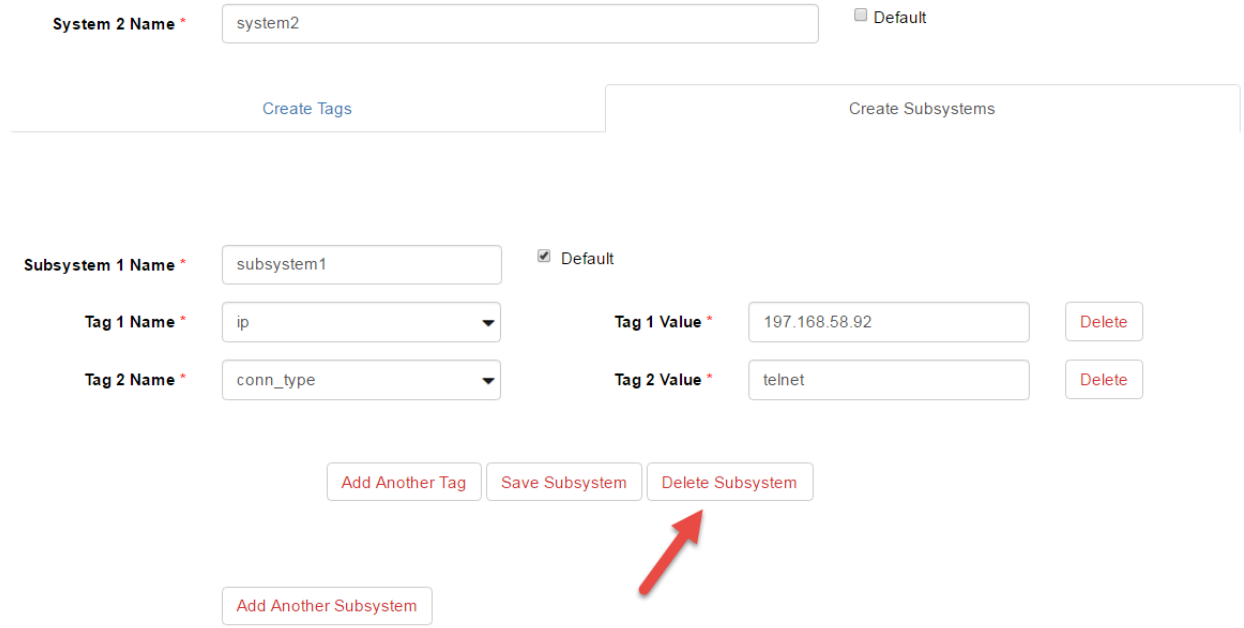


3.3.2.9. Delete Subsystem

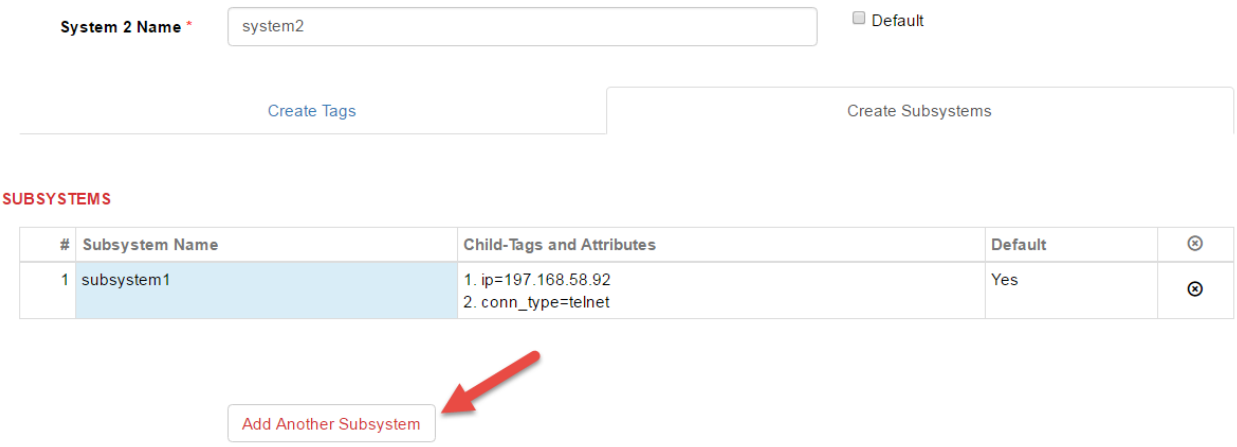
A saved subsystem can be deleted by clicking on the delete icon against it.



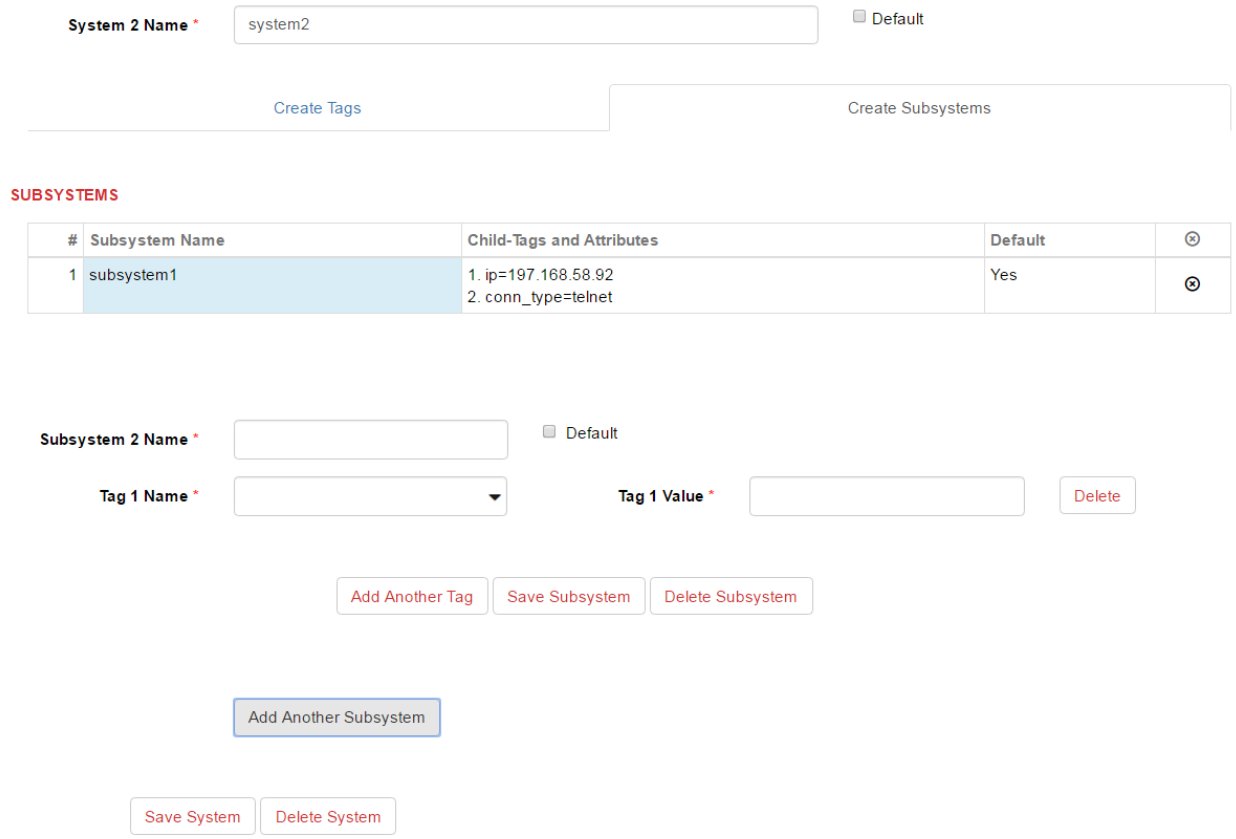
An unsaved subsystem can be deleted by clicking the “Delete subsystem” button.



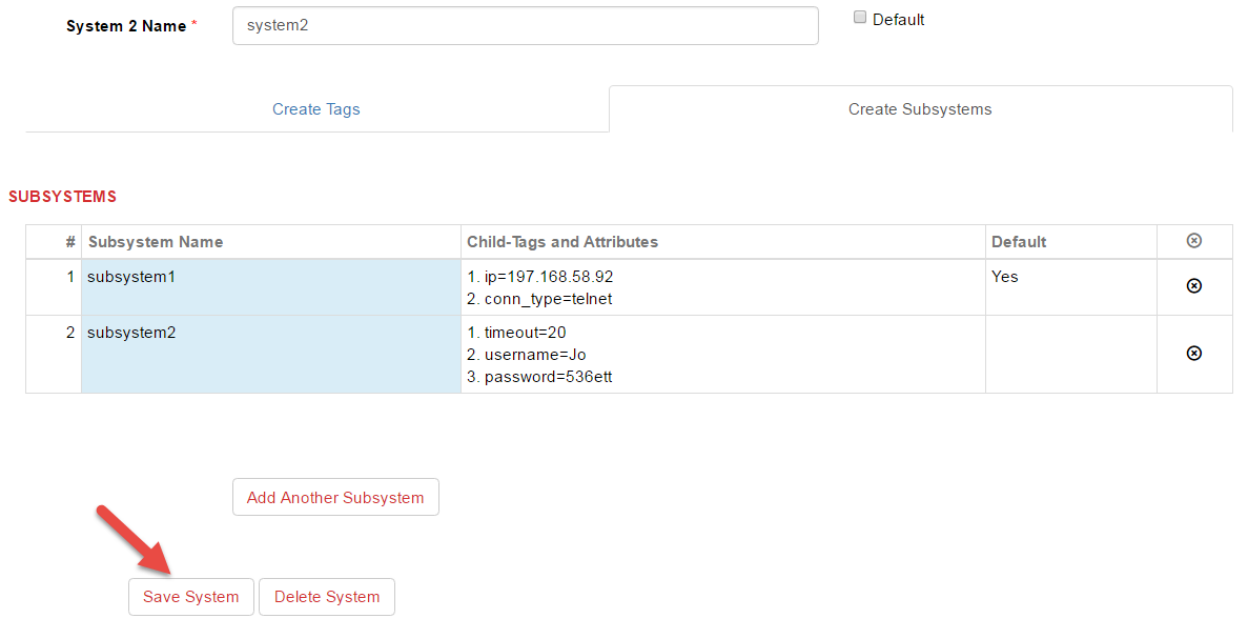
3.3.2.10. Add another subsystem



To add another subsystem to the same system, you can click the “Add Another Subsystem” button.



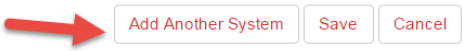
3.3.2.11. Save System



A system with subsystems can also be saved by clicking the “Save System” button

3.3.3. Add Another System

Another system can be added to this datafile by clicking the “Add Another System” button.

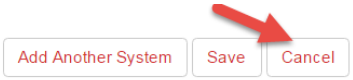


3.3.4. Save Data File

To save the data file, you can click on the “Save” button.

3.3.5. Cancel Data file creation

To exit data file creation without saving it, click “Cancel”.

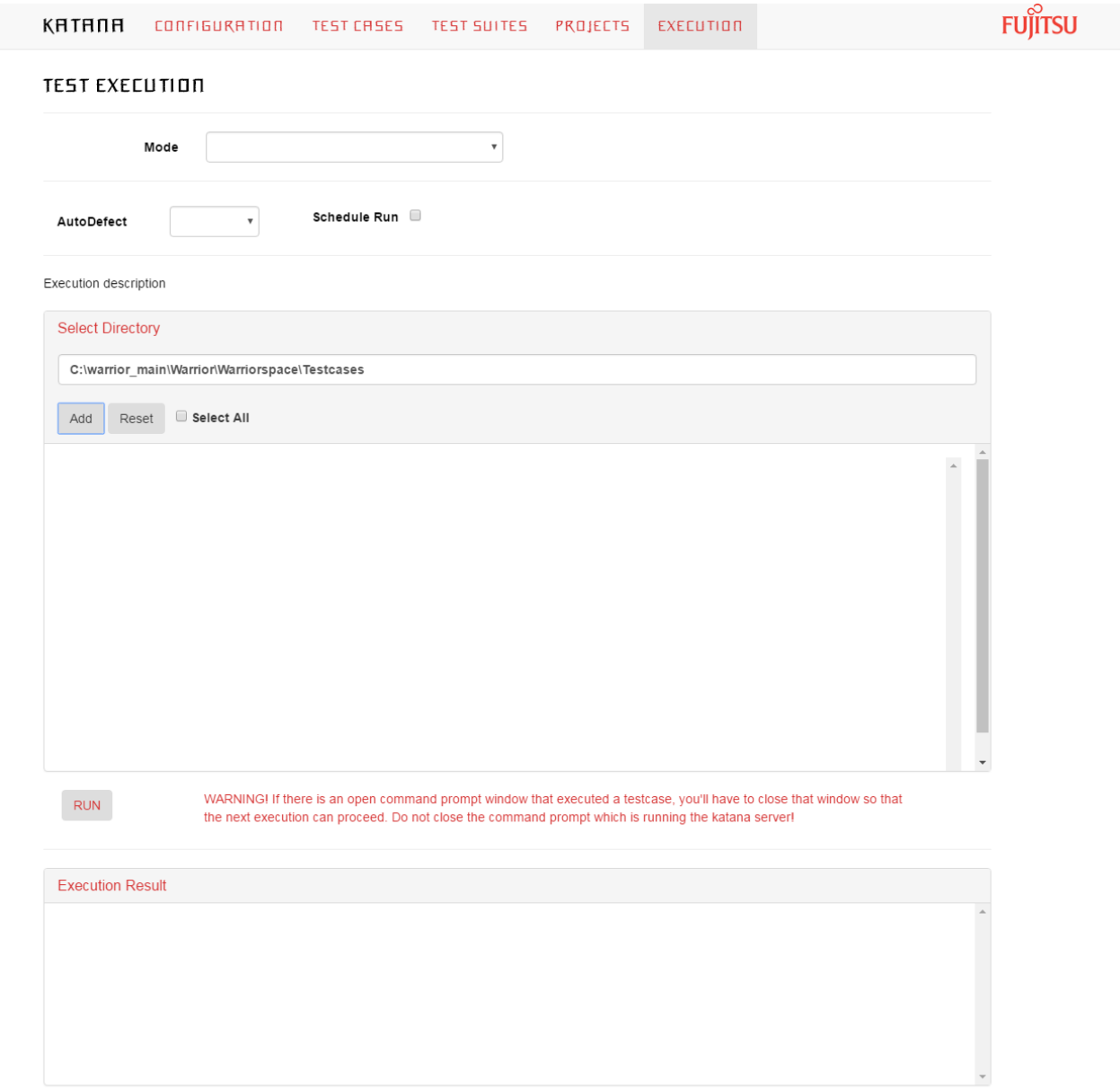


The final Data File thus created looks something like this:

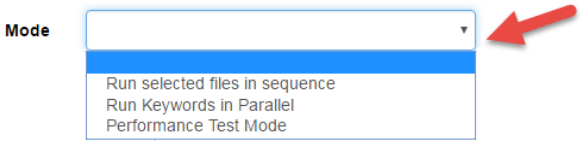
```
▼<systems>
   <description>This is a sample Data file</description>
  ▼<system default="yes" name="system1">
     <username>Jo</username>
     <password>536ett</password>
     <url>http://httpbin.org/post</url>
     <timeout>10</timeout>
   </system>
  ▼<system name="system2">
   ▼<subsystem default="yes" name="subsystem1">
      <ip>197.168.58.92</ip>
      <conn_type>telnet</conn_type>
    </subsystem>
   ▼<subsystem name="subsystem2">
      <timeout>20</timeout>
      <username>Jo</username>
      <password>536ett</password>
    </subsystem>
  </system>
</systems>
```

7. EXECUTING WARRIOR THROUGH KATANA

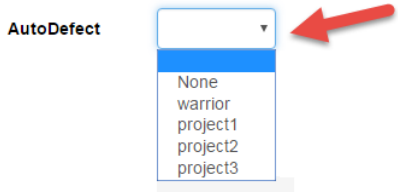
Warrior can also be executed through Katana. On clciking the 'Execution' tab, you will arrive at this page:



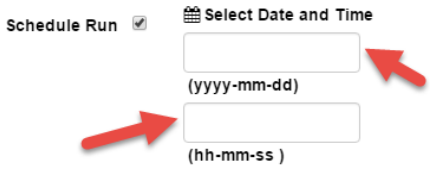
You will be able to select the 'Mode' by clicking on the dropdown:



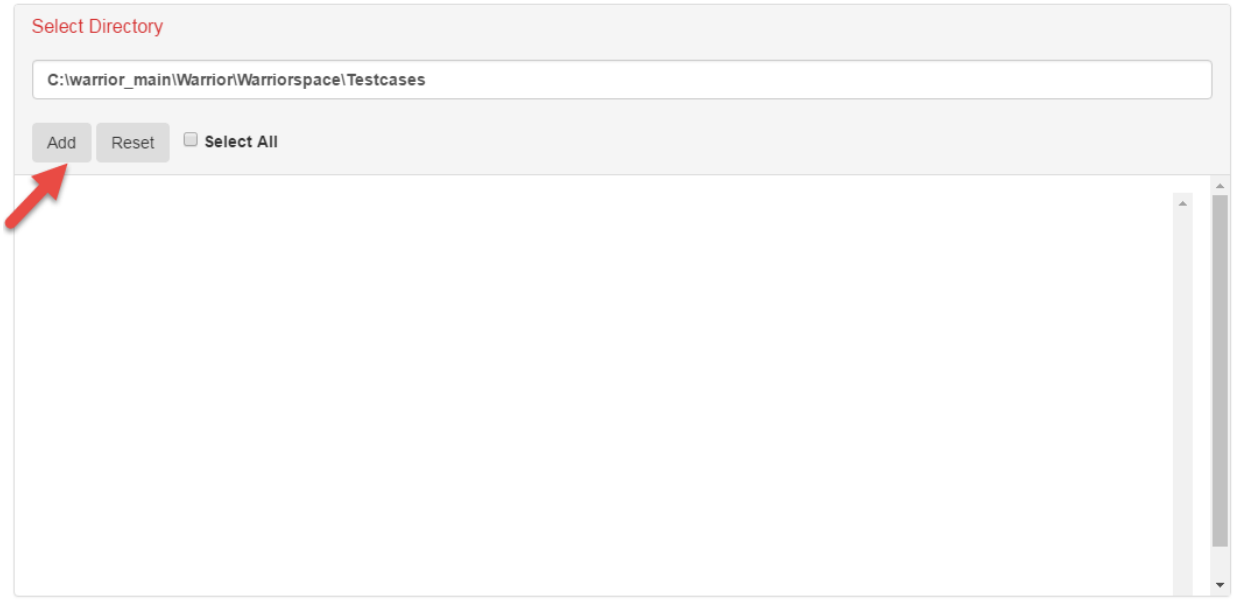
For JIRA auto defect creation, you can select your project from the 'AutoDefect' dropdown:



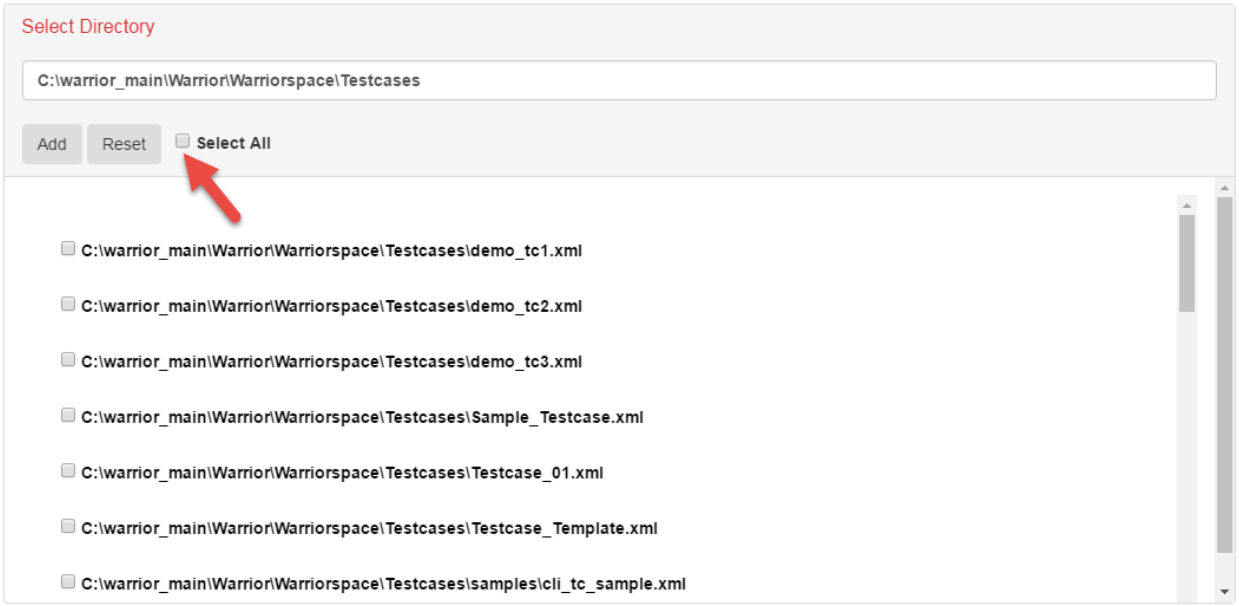
You can schedule the Testcase run by checking the 'Schedule Run' dropdown and then adding the date and time



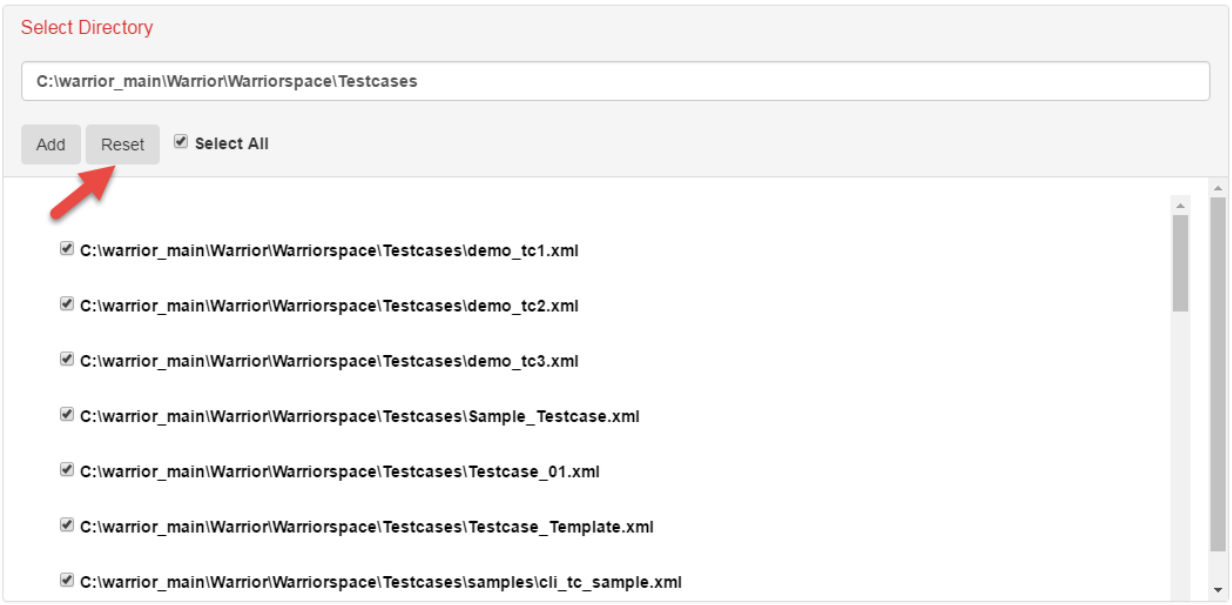
Then, You can click on add, to add all the testcases available in the directory showing up in the input box beneath the 'Select Directory', You can also edit that directory if you want to run Testcases from a different directory.

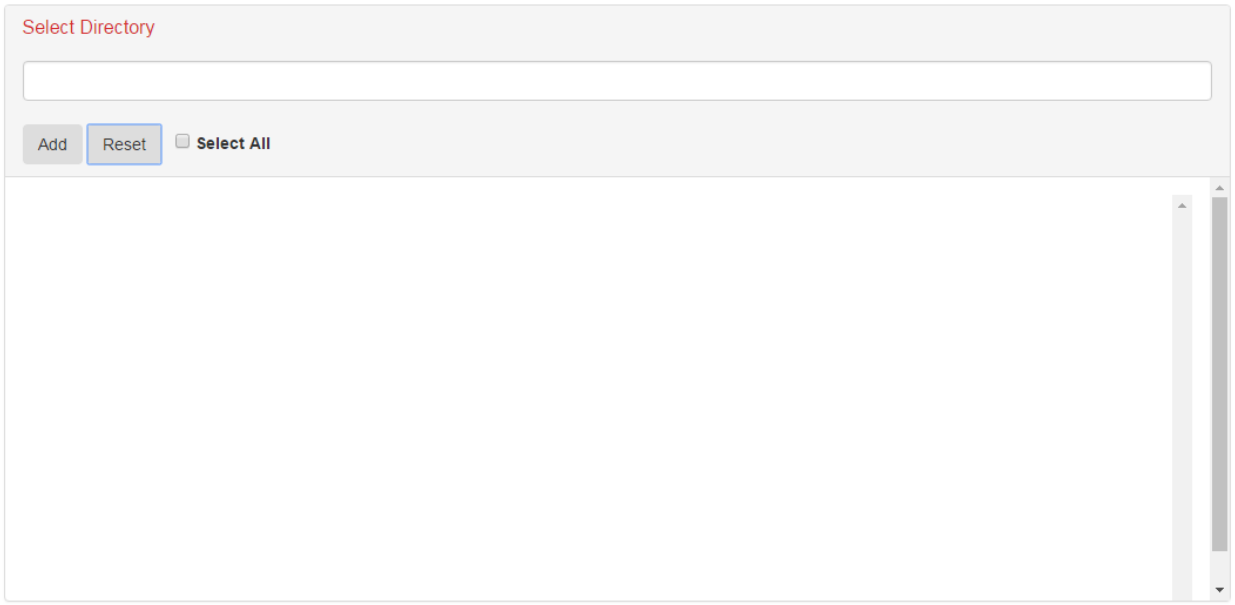


Checking 'Select All' lets you select all testcases.

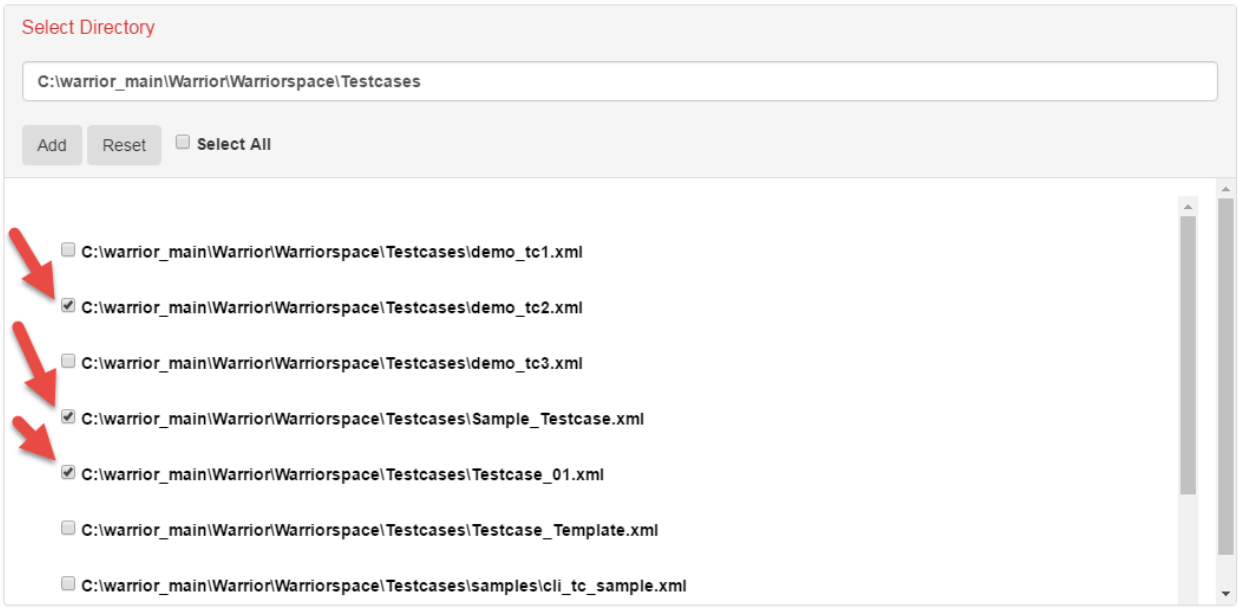


Clicking 'Reset', resets the entire page.



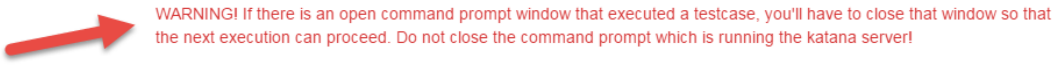


If you want to run a few, particular testcases, you can check the checkboxes against those testcases. The testcases would run in the order in which to check the checkboxes.

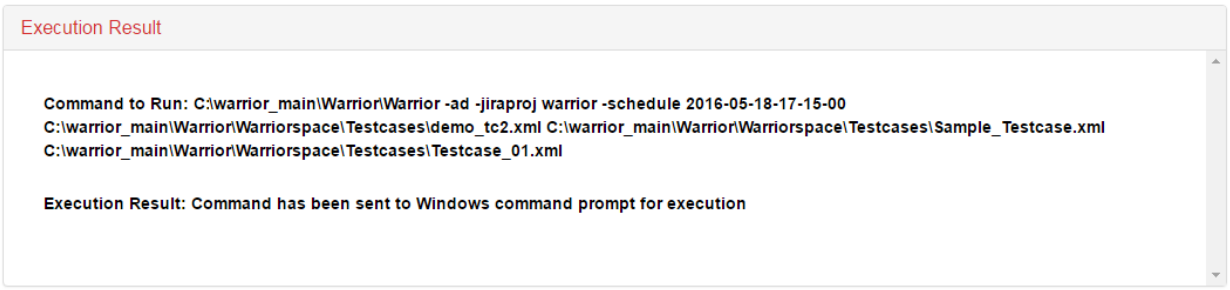


Clicking 'Run' would run the command in xterm – if you running Katana on Linux, or the Command Prompt – if you are running Katana on Windows.

This Warning appears only if Katana is being run on Windows



You would be able to see the Command that was sent to the xterm or Command Prompt here.



You would be able to view the execution on the console

WARHORN

USER GUIDE

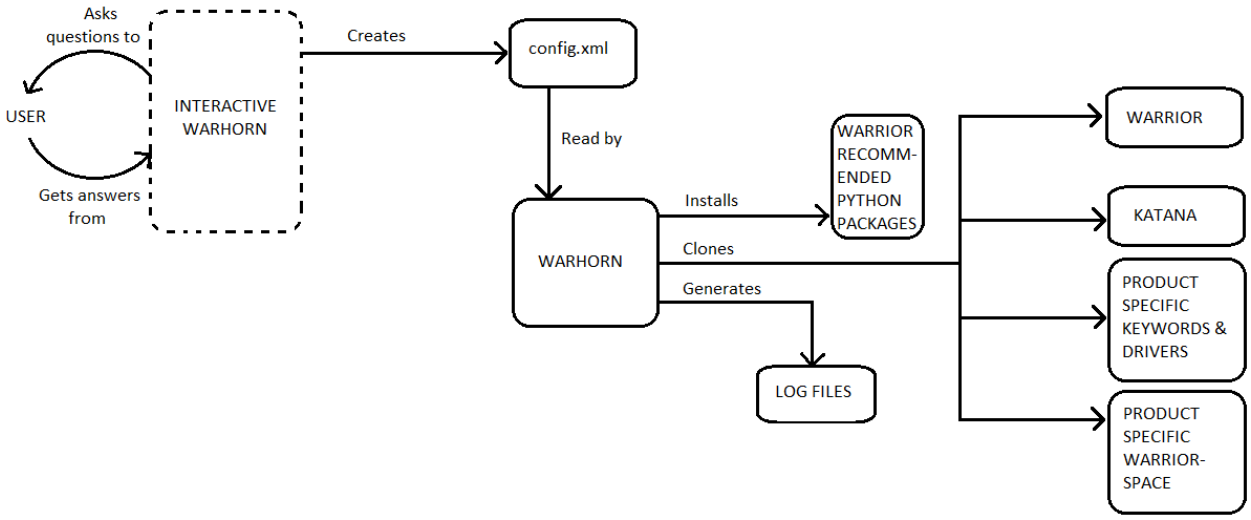
1. WHAT IS WARHORN?

Warhorn is Warrior's installation tool. Warhorn has the capability to install the python packages that are recommended by Warrior, Warrior, Katana – another Warrior tool, Product Specific Keywords and Drivers, and Warriorspace for you in your system.

With Warhorn, you can customize your Warrior environment to suit your needs and specifications (This is explained in more detail in Warhorn User Guide: Section 6.

2. HOW DOES WARHORN WORK?

Warhorn's premise is quite simple. Diagrammatically, it can be represented this way:



There are basically just three main components in Warhorn – the Interactive Warhorn, the Configuration file in XML format, and Warhorn.

The Interactive Warhorn is an automated way to generate a configuration file. Warhorn can be run in the interactive mode by this command:

```
python warhorn.py -interactive
```

The interactive mode in the diagram above has been drawn with a dotted line – this indicates that it is not necessary to use the interactive mode to create the configuration file. You can create the configuration file (see Warhorn User Guide: Section 6 for additional information) manually, in an editor - Gedit, Notepad++, or if you are feeling adventurous, vim - but the only caveat is that the configuration file has to be in a specific format as defined by Warhorn. Hence, it is recommended that you use the interactive mode.

The next component is the configuration file itself. It is essentially an XML file that acts as a data store for Warhorn. All the information provided by you in the configuration file is used by Warrior to install Warrior recommended dependencies and repositories in your system.

The third component is Warhorn – it holds all the intelligence necessary for cloning and installing stuff on your system. Warhorn produces log files (Warhorn User Guide: Section 8) which stores information and details about what happened during the Warhorn execution.

3. PREREQUISITES

Warhorn requires the following to run successfully:

1. A Linux system
2. Python – version 2.7.6+ till the latest in the 2.7 family
3. Git

Warhorn by default would install the dependencies recommended by Warrior. For this, warhorn may require you to have privileges to install packages on your system.

4. HOW TO INSTALL WARHORN?

The Warhorn Installation Tool is located at:

```
http://rtx-swtl-git.fnc.net.local/scm/wartools/warhorn.git
```

If you want to clone Warhorn into a particular location, type in:

```
cd path_to_the_desired_directory
```

Now that the system has git installed on it, open the terminal and type in to clone Warhorn:

```
git clone http://rtx-swtl-git.fnc.net.local/scm/wartools/warhorn.git
```

If you want to checkout a particular, branch, commit-id, or tag, type in:

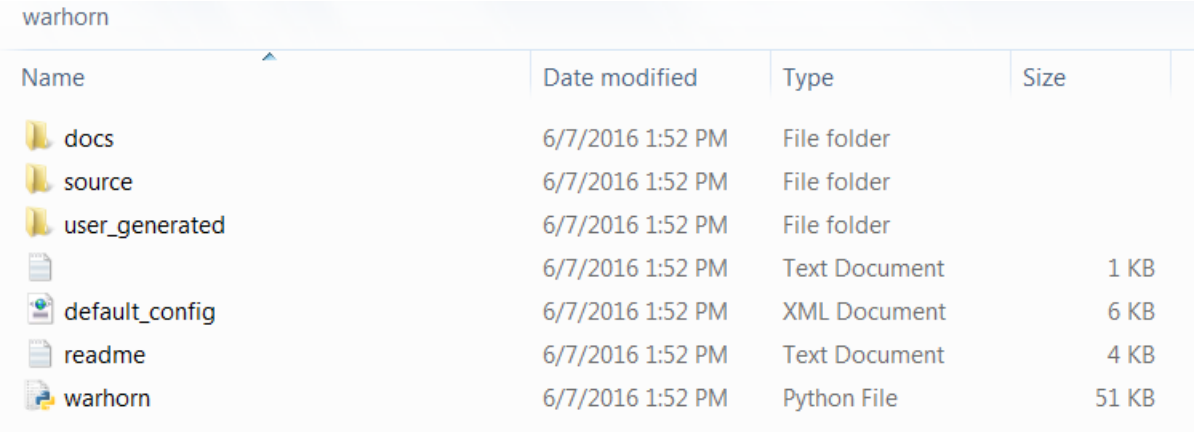
```
cd warhorn
```

This command gets you into the newly cloned warhorn directory. Then type in:

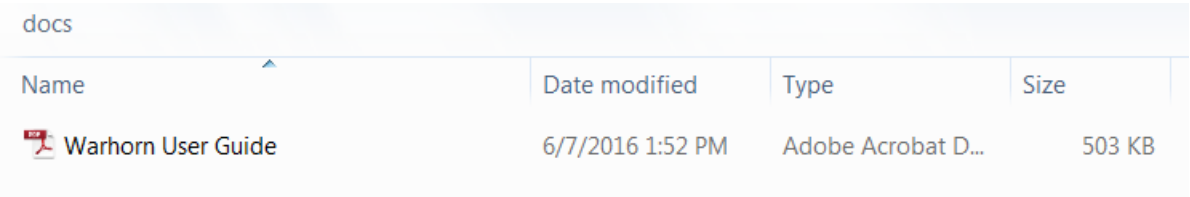
```
git checkout branch name/tag name/commit-id
```

5. THE DIRECTORY STRUCTURE

The current Warhorn directory structure looks something like this:

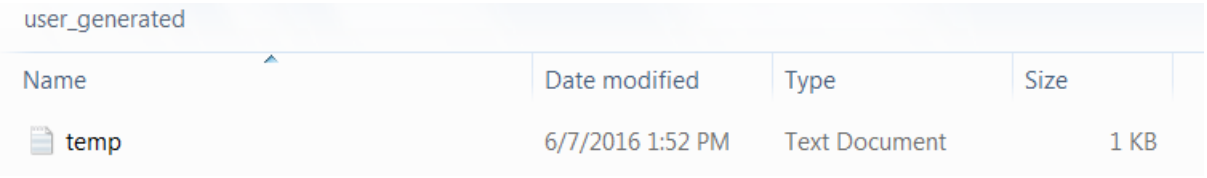


There are three folders and four files inside. The first is the Docs directory which stores the Warhorn User Guide.



The second directory – source – contains all the files that Warhorn uses to work. You should not change any contents of this directory.

Then comes the user_generated directory. This is the default directory for storing files created via the interactive mode.



The four files in the directory are – the .gitignore, which is just a file specifies intentionally untracked files that Git should ignore. The readme.txt – this file details the brief introduction to Warhorn, the default_config.xml which you are encouraged to open, read through, and edit, and finally the warhorn.py is the Warhorn executable.

6. HOW TO CONFIGURE THE DEFAULT CONFIGURATION FILE?

The configuration files are an integral part of running the Warhorn tool. Warhorn, when freshly cloned, comes with a default_config.xml file with it.

This configuration file feeds data to warhorn.py and tells it what to install and clone on your machine. The default_config.xml can be edited but it is strongly recommended that it not be moved from its original location and that its name not be changed as then, the command for running Warhorn would change (Warhorn User Guide: Section 7)

Any configuration file, including the default_config.xml file, should contain these five tags - `<warhorn>`, `<warrior>`, `<katana>`, `<drivers>`, and `<warriorspace>`. Out of these tags, only the <warrior> tag is the mandatory one.

6.1 THE <WARHORN> TAG

The `<warhorn>` tag is the first tag in the configuration file.

```
▼<warhorn name="Warhorn">
```

This tag contains information about the dependencies and their version that Warhorn would install by default. The comment below notes all that information

```
▼<!--
    The required versions for each of the dependencies given below are:
                jira: 1.0.3 - Not needed for Warrior version 2.1.1 and above
                lxml: 3.5
                ncclient: 0.4.6  - Not needed for Warrior version 1.9 and above
                paramiko: 1.16.0
                pexpect: 3.1
                pysnmp: 4.3.1
                requests: 2.9.1
                selenium: 2.48.0
            Your system may or may not have the correct version installed.
            ** Set the 'install' attribute to yes if you want to install the
            dependency in your system.
            ** Set the 'correct_version' attribute to yes if you want to upgrade the
            existing package to the required version

  -->
```

Since, Warhorn by default, clones the latest version of Warrior, unless specified otherwise, Jira and ncclient, have been turned “off” in the default_config.xml. So dependency section in the default_config.xml looks something like:

```
<!--
    Necessary for Jira module to log defects to Jira automatically
-->
<dependency name="jira" install="no"/>
<!--  Used by IronClaw tool  -->
<dependency name="lxml" install="yes"/>
<!--  Used for netconf operations  -->
<dependency name="ncclient" install="no"/>
<!--  Used by ncclient  -->
<dependency name="paramiko" install="yes"/>
<!--  Used by cli utilities  -->
<dependency name="pexpect" install="yes"/>
<!--  Used for snmp operations  -->
<dependency name="pysnmp" install="yes"/>
<!--  Used for rest operations  -->
<dependency name="requests" install="yes"/>
<!--  Used for web based testing  -->
<dependency name="selenium" install="yes"/>
```

Here all the dependencies recommended by Warrior have their own <dependency></dependency> tags. The dependency name is given as value to the name attribute and whether you want to install that dependency – 'yes' or 'no' is given as value to the install attribute. The dependency corresponding to the install attribute marked as 'yes' or 'no' will only get installed if the install attribute is set to 'yes'.

The `</warhorn>` closing tag marks the end of this section

```
</warhorn>
```

6.2 THE <WARRIOR> TAG

This tag carries with it information regarding the cloning of Warrior.

```
<warrior url="http://rtx-swtl-git.fnc.net.local/scm/war/warrior_main.git" destination=""
         label="" clean_install_warrior="">
</warrior>
```

This is a mandatory tag. So, please make sure every configuration file contains this tag and that it has been filled out with the correct information.

The attributes in the `<warrior></warrior>` tag:

Attribute	Description
url	Holds the URL of the warrior repository. The url tag has already been prepopulated in the default_config.xml. You can change that if you want to, but make sure that it is a valid url.

destination	Contains the path to the directory in which you want to clone Warrior. If it is left empty, then Warrior would be cloned in the same directory as Warhorn.
label	Indicates the branch name, tag name, or commit-id that you may want to checkout. If it is left empty, then the latest version of Warrior will be cloned.
clean_install	This tag lets you indicate whether you want to delete the existing Warrior in your system or not. If the tag value is set to 'yes', the existing Warrior will get deleted. If the tag is set to 'no', or is left empty, or is taken out altogether, existing Warrior will not get deleted.

6.3 THE <KATANA> TAG

This tag carries with it information regarding the cloning of the Warrior Tool - Katana.

```
<katana url="http://rtx-swtl-git.fnc.net.local/scm/war/katana.git"
        destination="" label="" clean_install="" clone="yes"></katana>
```

The attributes in the `<katana></katana>` tag:

Attribute	Description
url	Holds the URL of the Katana repository. The url tag has already been prepopulated in the default_config.xml. You can change that if you want to, but make sure that it is a valid url.
destination	Contains the path to the directory in which you want to clone Katana. If it is left empty, then Katana would be cloned in the same directory as Warhorn.
label	Indicates the branch name, tag name, or commit-id that you may want to checkout. If it is left empty, then the latest version of Katana will be cloned.
clean_install	This tag lets you indicate whether you want to delete the existing Katana in your system or not before cloning a fresh one. If the tag value is set to 'yes', the existing Katana will get deleted. If the tag is set to 'no', or is left empty, or is taken out altogether, existing Katana will not get deleted.
clone	Katana would be cloned only if this ta is set to "yes". If it is left empty, or set to "no", Katana will not be cloned.

6.4 THE <DRIVERS> TAG

Warhorn provides you with the ability to clone entire repositories containing all drivers and actions packages or you can select only the drivers that you need and the corresponding actions packages with it will be cloned for you.

This is the format for adding specific drivers that you want to clone from a particular Keyword repository:

```
        <repository url="http://repository/one/url.git" clone="yes" label=""
        all_drivers="no">
    <driver name="driver_one_name" clone="yes"></driver>
                <driver name="driver_two_name" clone=""></driver>
                <driver name="driver_three_name" clone="no"></driver>
                <driver name="driver_four_name"></driver>
</repository>
```

This is the format for cloning the entire Keyword repository:

```
        <repository url="http://repository/two/url.git" clone="yes"
        label="feature/xyz" all_drivers="yes">
</repository>
```

Attributes in a drivers tag:

Attribute	Description
url	Holds the URL of the repository that you want to clone.
all_drivers	This attribute lets you clone all the drivers and all the actions packages when set to yes. On the chance that you do not want to clone all the drivers, but only want to clone specific drivers, the following steps need to be taken: 1. Set the all_drivers attribute to 'no' 2. Create a driver tag under the repository tag as mentioned in the sample 3. Fill out the attribute of that driver tag. The driver tag has one attribute – name that takes in the name of the driver that you want and another attribute "clone" that lets you turn "off" a driver cloning. 4. Repeat steps 2 and 3 till all the needed driver tags are created.
label	Indicates the branch name, tag name, or commit-id that you may want to checkout.
clone	Gives you the ability to "turn off" the cloning of that particular repository. If this attribute is set explicitly to 'no', only then the repository would not be cloned. If

	it is set to 'yes', or is left blank, or is removed altogether, the repository will get cloned.

You can add as many repository tags as you want and Warhorn would clone all of them for you as long as all of them carry valid information. If you do not want to clone any Keyword repositories, the main drivers tag can be left empty.

6.5 THE <WARRIORSPACE> TAG

The warriorspace tag holds information regarding the product specific warriorspace repositories that you may want to clone. The process is similar to the drivers tag - you can add as many repository tags as you want as long as all of them carry valid information. The sample below represents the way in which the warriorspace tag should be filled out.

```
<warriorspace>
    <!-- Sample
    <repository url="http://warriorspace/repository/url.git" label="f455scd00i">
    </repository>
    -->
</warriorspace>
```

Attribute	Description
url	Holds the URL of the repository that you want to clone.
label	Indicates the branch name, tag name, or commit-id that you may want to checkout.

7. HOW TO RUN WARHORN?

Warhorn can be run by going the command line and running the warhorn.py file using the command:

```
python warhorn.py
```

This command supplies no arguments and therefore, Warhorn uses the defaut_config.xml to get the data needed to perform its tasks. It is not recommended that default_config.xml be moved from its default location and the name of the file be changed to anything else, since, the command above will not work if that is done. Warhorn would still run but only if the location of the xml file is passed as an argument.

If you need to run a particular xml file, the command should be:

```
python warhorn.py path_to_directory/file_name.xml
```

This command lets Warhorn use a particular configuration file to get its data from.

The Warhorn Interactive mode can create the configuration file in the correct XML format. Running the following command creates that XML file, and gives you the option of saving and then running that newly created file. You can also directly run the configuration file without saving it. To enter the Warhorn Interactive mode, type in this command:

```
python warhorn.py –interactive
```

8. THE LOG FILES

Log files are generated each time warhorn.py is run. Log files are not generated when the Warhorn Interactive mode is used to generate the configuration file only.

There are two kinds of log files: console_log.txt and print_log.txt. The console_log.txt logs all the console output of warhorn.py, including the print statements and errors. The print_log.txt logs just the print statements.

Both these files are stored in a time stamped directory underneath the logs folder - which gets generated when warhorn.py is run for the first time – along with the configuration file that was used by warhorn.py during that particular run.