WARRIOR FRAMEWORK

USER GUIDE

DOCUMENT HISTORY

Revision	Description	Author	Date
1	Ninja (1.0)	Brian Jishi	9/8/2015
2	Ninja (1.5)	Sathyamoorthy Radhakrishnan	1/7/2016
3	Ninja (1.6)	Sathyamoorthy Radhakrishnan	1/20/2016
4	Ninja (2.0)	Sanika Kulkarni	9/29/2016
5	Ninja (2.9)	Brian Jishi	3/16/2017
6	Ninja (3.0)	Sanika Kulkarni	3/31/2016
7	Ninja(3.0)	Santhosh Nayak	02/06/2019

TABLE OF CONTENTS

WARRIOR FRAMEWORK USER GUIDE

1.0 WHAT IS WARRIOR?

Warrior is a generic automation framework that automates software tasks and processes, in addition to functional, performance and solutions testing. It is designed and implemented by Fujitsu Network Communications.

Warrior is a keyword driven framework that uses xml based test data that is generated by a web interface. The Warrior Framework was designed to allow users to effectively create reusable generic keywords and leave the execution intelligence out to be handled by Warrior. The execution of the keyword will be handled by the framework and designed in the cases, suites and project files. Warrior Cases, Suites and Projects are created in xml. These files may be edited in any xml editor or using Katana. Katana is a web based interface that simplifies editing and creation of xml cases, suites and projects. Katana provides keyword searching, keyword documentation review and keyword, case and project execution instructions.

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

Warrior Framework was designed for automation of end to end solutions. Its architecture allows for testing multiple devices under test within a single case, suite or project file. For example, from a single case, Warrior Framework can execute keywords on a windows based system, a Network Element, a web-based application and a Linux based VM. Warrior Framework also allows the integration of test sets such as Wireshark, Spirent and others within the case. In addition, Warrior Framework can also integrate cases created in other frameworks and execute them in conjunction with Warrior Cases. Warrior Framework's ability to integrate multiple systems, test sets, and frameworks within a single execution simplify the design and execution of end to end testing.

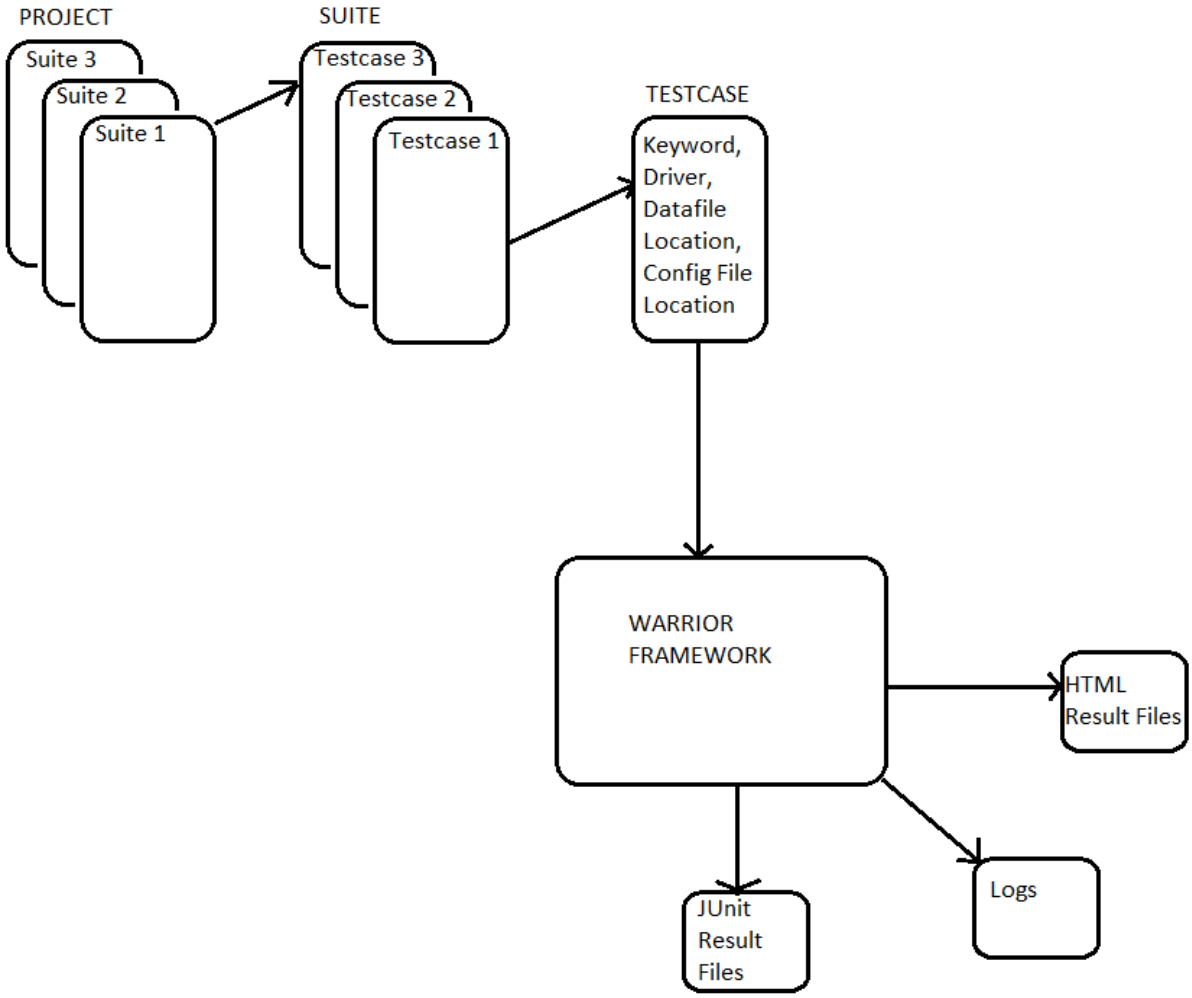
1.1 WHAT MAKES WARRIOR FRAMEWORK GENERIC?

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

1. Framework
2. Keywords
3. Cases (also, Suites and Projects)

4. Data Files

A diagrammatic representation of Warrior Framework:



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

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

Every component here is independent of every other component. Exactly this, makes Warrior Framework a customizable automation framework. Given a set of cases and data files, changing the system

information in the data file or using a different data file altogether, lets you test different systems by changing just a single file. Using a different case, or by changing the existing case, you can test different scenarios without writing a single line in Python.

Warrior Framework comes with a set of Core 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 Framework allows you to write your own drivers and keywords, and those new, specific keywords can be used in your cases.

1.2 WHAT NOT TO DO IN WARRIOR FRAMEWORK?

Warrior Framework 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 Framework. In order for your automation efforts to be successful using Warrior Framework, please follow the following 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 Framework is not a python scripting tool. If your goal is to write cases as python scripts, Warrior Framework should not be used.
- Use the recommended Warriorspace directory structure for creating cases, suites and projects. It will simplify the use of Warrior Framework.
- 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

2.1 CLONING WARRIOR

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

https://github.com/warriorframework/warriorframework.git

Clone Warrior Framework, open up your terminal and type in this command:

```
$ git clone https://github.com/warriorframework/warriorframework.git
```

You may be prompted to enter your GitHub user name and password while this repository is being cloned.

And that's it! You have successfully cloned Warrior on your system.

Please make a note that as of today, Warrior is supported only on Linux.

2.2 SETTING UP THE ENVIRONMENT TO EXECUTE WARRIOR

2.2.1 THROUGH WARHORN

We have automated the process of setting up the environment for Warrior – Warhorn a tool that comes with Warrior, when run will set up this environment for you. Please make sure that your system has Python (any version in the 2.7 family) installed on it.

To run warhorn, you just have to follow these simple steps:

Go into the warhorn directory:

```
$ cd warriorframework/warhorn
```

The warhorn directory is right under the warriorframework directory.

Then run this command:

```
$ python warhorn.py
```

Warhorn will then set up the environment for you.

If you want read the Warhorn User Guide, you can refer to this section.

If you want to set up the environment by yourself, then read through the next section.

2.2.2 MANUALLY

The Warrior environment can be set up manually as well. You will just need Python (any version in the 2.7 family).

1. Pip installation

To install pip on Linux, run this command on your terminal to get pip:

```
$ wget https://bootstrap.pypa.io/get-pip.py
```

Next, run this command to install pip on Linux:

```
$ sudo python get-pip.py
```

This would install pip on your machine.

2. Package installation

Depending on what you want to use Warrior for, you may need to install one or all of these packages, although, we do recommend that you run all the commands below:

To install lxml:

```
$ sudo pip install lxml==3.3.3
```

To install paramiko:

```
$ sudo pip install paramiko==1.16.0
```

To install pexpect:

```
$ sudo pip install pexpect==4.2
```

To install pysnmp:

```
$ sudo pip install pysnmp==4.3.2
```

To install requests:

```
$ sudo pip install requests==2.9.1
```

To install selenium:

```
$ sudo pip install selenium==2.53.0
```

And this would set up the Warrior environment for you.

2.3 RUNNING WARRIOR

Once your environment to run Warrior is set up, you can execute Warrior by implementing the following steps –

Go into the warrior directory:

```
$ cd warriorframework/warrior
```

The warrior directory is right under the warriorframework directory.

Then run this command:

```
$ python Warrior Warriorspace/Testcases/Demo_Cases/Demo_Test_Of_Inventory_
Management_System.xml
```

This would run one of the Demo Cases that Warrior has.

The standard format to execute Warrior is:

```
$ python path/to/the/Warrior/executable  path/to/case
```

3.0 SETTING UP WARRIOR

3.1 JIRA SETTINGS

Warrior can be set up to automatically report defects into JIRA if a failure and/or error is observed during its execution.

You can follow the steps below to configure Warrior to report defects automatically:

1. Open the JIRA configuration file which is located in

   ```
   warriorframework/warrior/Tools/Jira/jira_config.xml
   ```

   This is the JIRA configuration file looks like this:

```xml
<?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>

```

2. Now, each system tag carries information about the project that you want to upload defects to if failures are observed.
3. Enter a name for that project in the name attribute for the system tag.
4. Type in the URL for that project in the url child tag
5. Fill in the username and the password for the project
6. And finally, enter the key for that project.

There are some rules by which Warrior uses this XML file:

1. The XML file can contain multiple systems each being a separate Jira project. Only one project can be defined as a default project.

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

3. To define a project as default, it should be marked as default=”true”.

4. If multiple projects are defined as default, Warrior will use the first project marked as default.

5. If no project is marked default, then Warrior will use the first project in the xml file. In the screenshot below the last project is marked default

Now, you can execute Warrior either via command line or through Katana.

3.2 ENCRYPTION SETTINGS

Warrior allows for the usage of encrypted passwords in its Cases and data files. 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.

3.3 LOGS & RESULTS DIRECTORY SETTINGS

Warrior, by default, uses this directory to store the files created during its execution – these are the log and result files. You can read more about those here.

If you want, you can configure Warrior to create these log files and directories in a specific location rather than the default. This can be achieved with the following steps:

1. Open the w_settings file which is located in:

```
Warriorframework/warrior/Tools/w_settings.xml
```

2. This file will look something like this:

```
<?xml version="1.0" ?>
<Default>
    <Setting name="def_dir">
        <Logsdir></Logsdir> <!--Optional, uses default, support absolute path, relative path and environment variable(${ENV.DEF_LOGSDIR})-->
        <Resultsdir></Resultsdir> <!--Optional, uses default, support absolute path, relative path and environment variable(${ENV.DEF_RESULTSDIR})-->
    </Setting>
    <Setting name="mail_to" mail_on="">
    <!-- 'mail_on' attribute is to specify 'when to send an email'. Supported values are 'per_execution, first_failure & every_failure'.
            1. per_execution(default) - to send an email after the end of case/suite/project execution
            2. first_failure - to send an email as soon as the first failure occurs during suite/project execution
            3. every_failure - to send an email whenever a case fails during suite/project execution
         When 'mail_on attribute' is not given or left empty, default value will be used for sending email.
         It also accepts comma separated values, example mail_on="per_execution, first_failure" -->
        <smtp_host></smtp_host> <!-- Optional, use FNC smtp host smtp.fnc.fujitsu.com to notify test case execution results -->
        <sender></sender> <!-- Optional, use sender's email such as Warrior_PBO@fnc.fujitsu.com -->
        <receiver></receiver> <!-- Optional, use receiver's email such as FNC.User@fnc.fujitsu.com -->
        <subject></subject> <!-- Optional, use this as subject line such as WARRIOR , additional test case execution results Pass/Fail and Test Execution File shall be appended to this -->
    </Setting>
</Default>
```

3. Under the “Setting” tag with the value for the name attribute as “def_dir”, there is a child tag for the Logs Directory (Logsdir) and for the Results directory (Resultsdir)

4. To specify a directory for the Log files, add that file path as a value to the Logsdir tag

5. To specify a directory for the Result files, add that file path as a value to the Resultsdir tag

Warrior will then use the directory paths mentioned above to store the generated log and/or result files.

3.4 EMAIL SETTINGS

If you want, you can configure Warrior to send a notification email to you after the execution has finished, or as soon as a failure is observed during the execution. This can be achieved with the following steps:

1. Open the w_settings file which is located in:

```
Warriorframework/warrior/Tools/w_settings.xml
```

2. This file will look something like this:

```
<?xml version="1.0" ?>
<Default>
    <Setting name="def_dir">
        <Logsdir></Logsdir> <!--Optional, uses default, support absolute path, relative path and environment variable(${ENV.DEF_LOGSDIR})-->
        <Resultsdir></Resultsdir> <!--Optional, uses default, support absolute path, relative path and environment variable(${ENV.DEF_RESULTSDIR})-->
    </Setting>
    <Setting name="mail_to" mail_on="">
    <!-- 'mail_on' attribute is to specify 'when to send an email'. Supported values are 'per_execution, first_failure & every_failure'.
            1. per_execution(default) - to send an email after the end of case/suite/project execution
            2. first_failure - to send an email as soon as the first failure occurs during suite/project execution
            3. every_failure - to send an email whenever a case fails during suite/project execution
         When 'mail_on attribute' is not given or left empty, default value will be used for sending email.
         It also accepts comma separated values, example mail_on="per_execution, first_failure" -->
        <smtp_host></smtp_host> <!-- Optional, use FNC smtp host smtp.fnc.fujitsu.com to notify test case execution results -->
        <sender></sender> <!-- Optional, use sender's email such as Warrior_PBO@fnc.fujitsu.com -->
        <receiver></receiver> <!-- Optional, use receiver's email such as FNC.User@fnc.fujitsu.com -->
        <subject></subject> <!-- Optional, use this as subject line such as WARRIOR , additional test case execution results Pass/Fail and Test Execution File shall be appended to this -->
    </Setting>
</Default>
```

3. Under the “Setting” tag with the value for the name attribute as “mail_to”, you can add the information about the host, sender, receiver, and subject so that you will receive an email with the results and logs attached to it.

4. Now, if you want Warrior to send this notification email only after the execution has finished, then enter the value “per_execution” in the mail_on attribute for the Settings tag

5. If you want Warrior to send in the notification email only when the first failure is observed during the execution, then enter the value “first_failure” in the mail_on attribute for the Settings tag

6. If you want Warrior to send in the notification email after each failure that may be observed during the execution, then enter the value “every_failure” in the mail_on attribute for the Settings tag

7. You can also enter a combination of the three options above, as comma separated values in the “mail_on” attribute.

8. If these details are wrong or left unfilled, Warrior will not be able to send an email.

3.5 DATABASE SETTINGS

If all you files are stored in a database, then you can connect to that database so that Warrior can access those files. As of today, Warrior supports the mongoDB as its database.

You can give Warrior the information related to this database in the file database_config_xml located in

```
Warriorframework/warrior/Tools/database
```

This config file has a section <dataservers> in which you can provide connection information of the database to Warrior. You can add as many databases as you need.

```
<dataservers>

    <system name="localhost_td" default="true">
        <dbtype>mongodb</dbtype>
        <host>localhost</host>
        <port>27017</port>
        <uri></uri>
        <dbname>testdata_config</dbname>
    </system>

    <system name="localhost_var">
        <dbtype>mongodb</dbtype>
        <host>localhost</host>
        <port>27017</port>
        <uri></uri>
        <dbname>variable_config</dbname>
    </system>

</dataservers>
```

Required fields for MongoDB server are:

1. dbtype - Type of the database
2. host - DB server IP, optional when URI is specified. The default host is localhost
3. port - DB server port, optional when URI is specified. The default port – 27017
4. uri - Connection string in URI format, optional when host & port are specified. Example: mongodb://localhost:27017
5. dbname - Database name

Each database will contain a db_name which is the database name that Warrior should look for. Information about that database can be given in separate blocks in database_config.xml like this:

```
<testdata>
    <td_system>localhost</td_system>
    <td_collection>testdata</td_collection>
    <td_document>cli_global_testdata</td_document>
    <global_system>localhost</global_system>
    <global_collection>testdata_global</global_collection>
    <global_document>cli_global_block</global_document>
</testdata>

<variable_config>
    <var_system>localhost_td</var_system>
    <var_collection>variable_config</var_collection>
    <var_document>cli_use_var_varconfig</var_document>
</variable_config>
```

Warrior also allows you to store the results and logs generated after each executed in the database. Systems specified under 'resultservers' block will be used for storing Warrior HTML/XML results in the database server.

```
<resultservers>

    <system name="localhost" default="true">
        <dbtype>mongodb</dbtype>
        <host>localhost</host>
        <port>27017</port>
        <uri></uri>
        <dbname>warrior_results</dbname>
    </system>

</resultservers>
```

With the information given above, Warrior would be able to store the generated results and logs in the database with the name warrior_results.

System name of 'resultservers' can be passed as an option (dbsystem) when executing Warrior via CLI. Example:

```
./Warrior <xml_path> -dbsystem localhost.
```

If the system name is not passed during Warrior execution, system with 'default=true' under resultservers block will be used. If there is no default system then the first system specified under resultservers block will be used as database server for storing HTML/XML results.

3.6 SMART ANALYSIS SETTINGS

Warrior can identify which system it is communicating with using this feature. The smart analysis tries to match the end prompt received by Warrior on connection to identify the system and it then sends relevant

commands to it obtain details about that systems – like which software it is running, the current version and so on.

This feature uses the connect_settings.xml file located in:

```
Warriorframework/warrior/Tools/connection
```

The file looks something like:

```
<?xml version="1.0" ?>
<credentials>
    <!--Below is the sample of how a smart analysis data block would work
        The smart analysis will look for each search_string in the connect/disconnect prompt
        If it match a system, it will then send the command from the testdata file of that system -->
    <!-- <system name="1finity">
        <search_string>1FINITY</search_string>
        <testdata>configs/1finity_command_data.xml</testdata>
    </system>
    <system name="ubuntu">
        <search_string>Ubuntu 14.04</search_string>
        <testdata>configs/ubuntu_command_data.xml</testdata>
    </system> -->
</credentials>
```

Here, you can add a system tag with a name attribute and then provide a search string to Warrior. Warrior will look for this search string in the system's response and then identify that system.

Now, you can also provide a testdata file to Warrior as a relative path is the testdata tag under the system tag. Typically, this testdata file should be stored in the configs directory located in

```
Warriorframework/warrior/Tools/connection
```

This testdata file would look something like this:

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

<data>
    <testdata title="connect" execute="yes">
        <command send="pwd" end=":"/>
        <command send="ls" end=":"/>
    </testdata>

    <testdata title="disconnect" execute="yes">
        <command send="pwd" end=":"/>
        <command send="ls" end=":"/>
    </testdata>
</data>
```

Commands that should be sent to this identified system can be given in this format. These commands would be sent by Warrior to the system and then, the response thus obtained would be displayed on the console.

4.0 DOCUMENTATION

4.1 USER GUIDE

The user Guide for Warrior is located at:

```
warriorframework/docs/Warrior_Framework_UserGuide.pdf
```

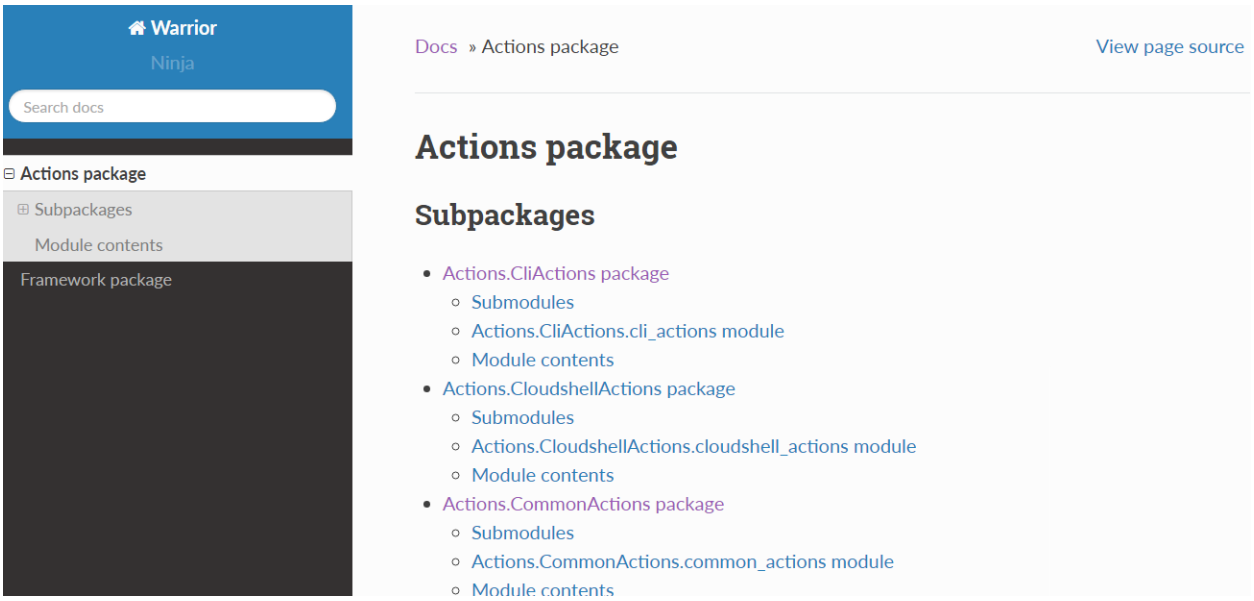
4.2 READ THE DOCS

The documentation for all the Warrior Actions (i.e., keywords) and all the utilities provided by Warrior can be found at:

```
warriorframework/docs/build/html/warrior_index.html
```

Note: Since this is an HTML file, you will need a browser (Firefox, Chrome and such) to open and read this file.

The page would look something like this:



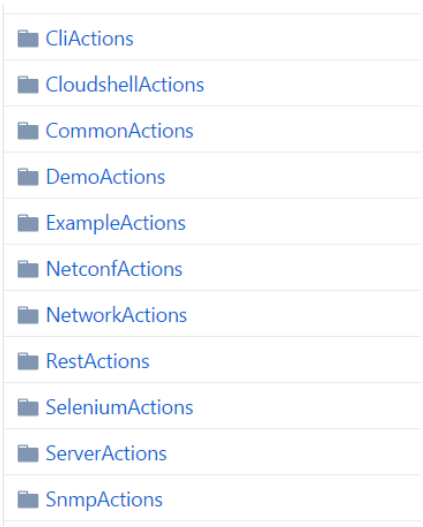
4.3 KEYWORD DOCUMENTATION

The documentation for each keyword can be found in the Action file containing that keyword.

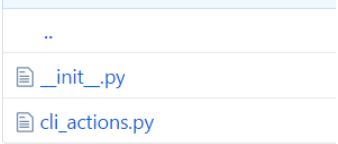
For example: If you are interested in knowing more about the connect keyword in the CLI module, you can follow the steps below to read the documentation for that keyword:

1. Go to the warriorframework/warrior/Actions directory.

2. You will see a directory structure like this:



3. Open the directory for the module that you interested in – in this case, the CLI module. This CLI module has just one file inside of it, as of today.



4. Open the cli_actions.py file. The documentation for each available keyword will be given right below the declaration of that keyword. Like this:

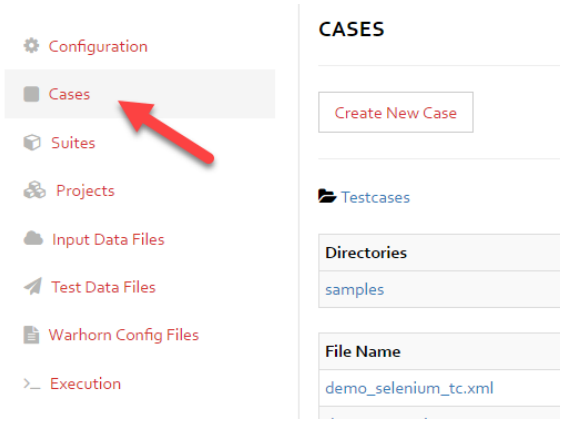
```
def connect(self, system_name, session_name=None, prompt=".*(%|#|\$)", ip_type="ip"):
    """
    This is a generic connect that can connect to ssh/telnet based
    on the conn_type provided by the user in the input datafile.

    :Datafile usage:

        Tags or attributes to be used in input datafile for the system or subsystem
        If both tag and attribute is provided the attribute will be used.

        1. ip = IP address of the system.\

            Default value for ip type is ip, it can take any type of ip's
            to connect to (like ipv4, ipv6, dns etc)

            Users can provide tag/attribute for any ip_type under the system
            in the input datafile and specify the tag/attribute name
            as the value for ip_type argument, then the connection will be
            established using that value.

        2. username = username for the  session.
        3. password = password for the  session.
        4. timeout = use if you want to set timeout while connecting,\
            used for both ssh and telnet
```

5. The screenshot above shows the (partial) documentation for the connect keyword.

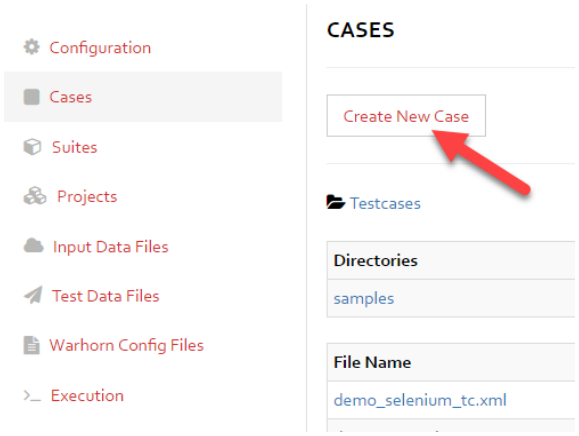
Now, opening a python file and reading though it can be cumbersome for a Case developer. So there is an easier way to the same thing. You can you use Katana to view the documentations for all the available keywords. For that you will need to:

1. If you have never used Katana before, then you will have to run katana first, and then set it up. If your Katana is already set up, you can skip this step.

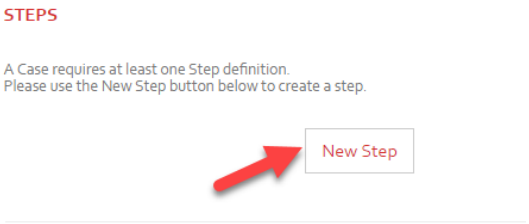
2. Once that is done, click on the Case tab in Katana:



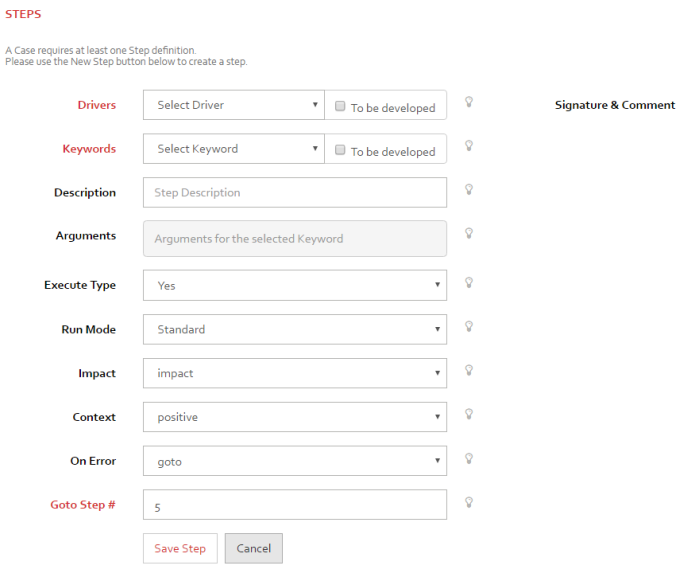
3. Now click on the “Create New Case” button:



4. And then scroll down to the “New Step” button and click on that:



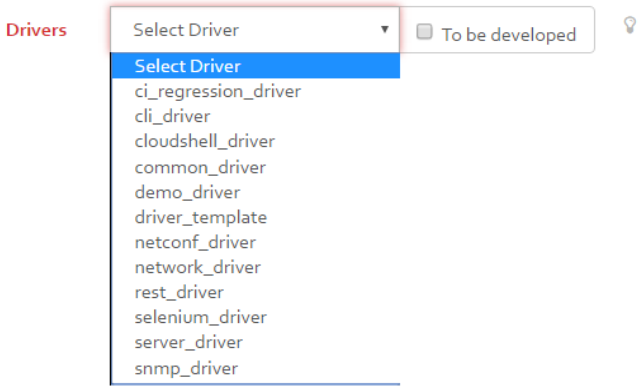
5. On clicking the “New Step” button, a new set of fields drop down for you to be filled. All the newly appeared fields are a part of this new “step”.



6. Every step has a driver field in it.

Katana lets you select a driver by showing all the drivers that you have in your Warrior Framework directory in the dropdown. You can select any driver out of those.

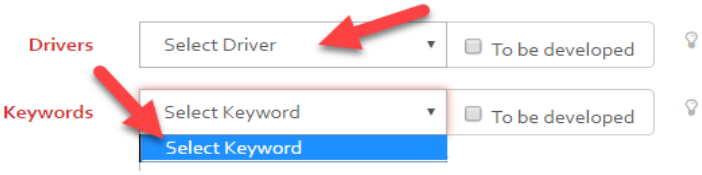
Note: The list of drivers on your Katana may differ since you may have different drivers in your ProductDrivers directory.



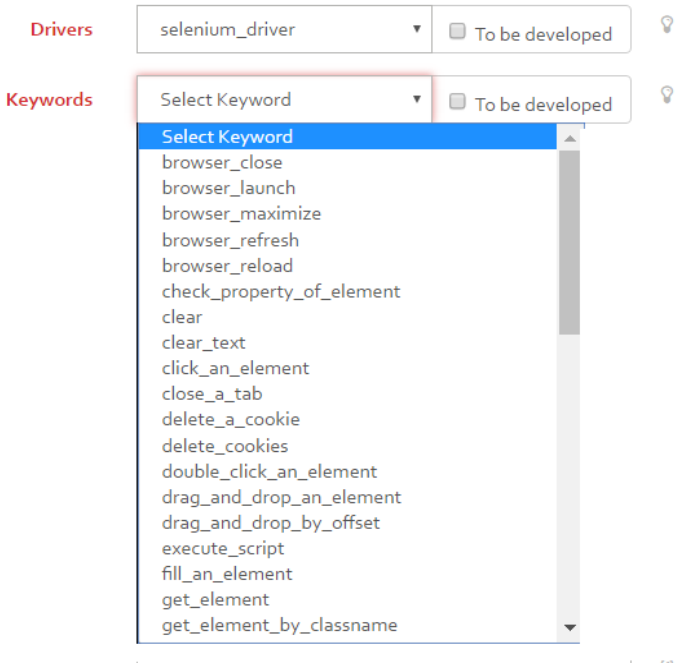
You can then select any driver from that list:

7. Now you can choose the keyword you want to know more about

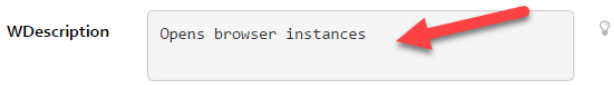
Note: If no driver is selected, no keywords will show up in the dropdown. You have to choose the driver before you can choose the Keyword.



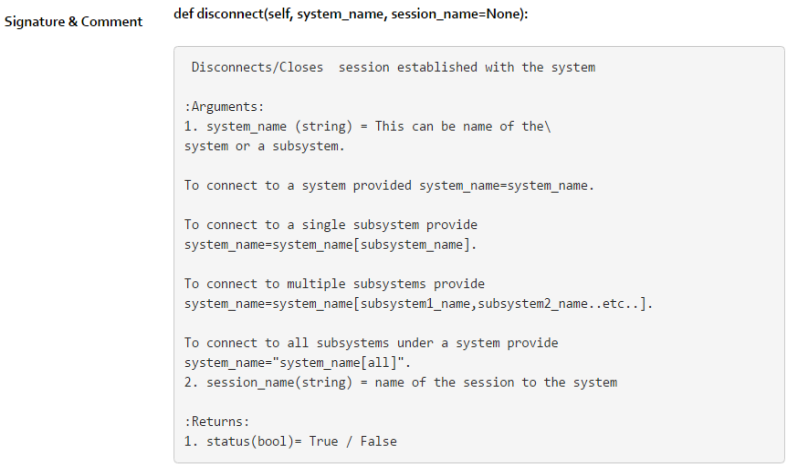
If the driver is chosen, a list of Keywords would show up. You can select any Keyword out of those.



8. The WDescription field will be populated when the Keyword is chosen.



9. The Signature & Comment field gets populated when a developed Keyword is selected.



Here, you will be able to view the documentation for any keyword.

5.0 WARRIOR TOOLS

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

5.1 WARHORN: AN OVERVIEW

Warhorn is a tool that automates the process of updating Warrior versions and assembling external Keywords and Warriorspace for you. Warhorn also installs the pre-requisites that Warrior Framework recommends you need 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

5.2 KATANA: AN OVERVIEW

Katana is Warrior Framework's web based case creation and execution tool. From Katana, users can create cases, suites, projects, and input data files. Cases, 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 cases, suites, and projects from it's execute tab.

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

5.3 IRONCLAW: AN OVERVIEW

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

5.3.1 VERIFICATION OF CASE XML FILES:

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

1. Verify conformance to 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.

5.3.2 VERIFICATION OF SUITE XML FILES:

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

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

5.3.3 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 suites listed in project exist.
3. Verify each suite as described in Section 8.0 of this User Guide

5.3.4 RUNNING IRONCLAW

To run Ironclaw execute the following command:

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

5.4 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:

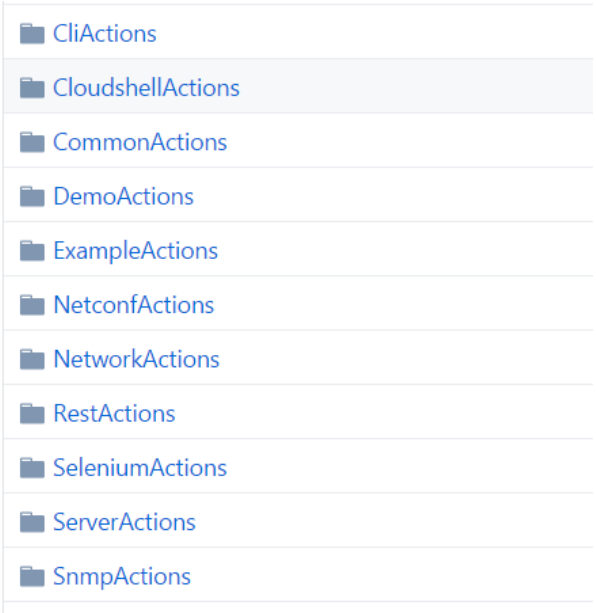
```
python Warrior –mockrun path/to/case
```

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/case
```

6.0 UNDERSTANDING THE WARRIOR DIRECTORY STRUCTURE

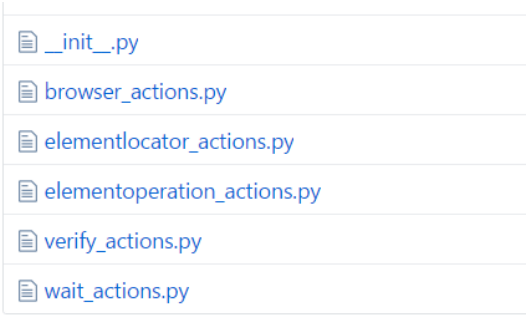
There are in total seven directories and one Warrior Framework executable in the Warrior Framework directory. What to expect in each directory and its utility has been explained in detail below. Whether you are a case developer or a keyword developer, acclimatizing yourself to Warrior Framework directory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



6.1 ACTIONS DIRECTORY

Warrior Framework'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 7.0 of this User Guide.

An Actions package in Warrior looks something like:



This is the Selenium Actions directory located in:

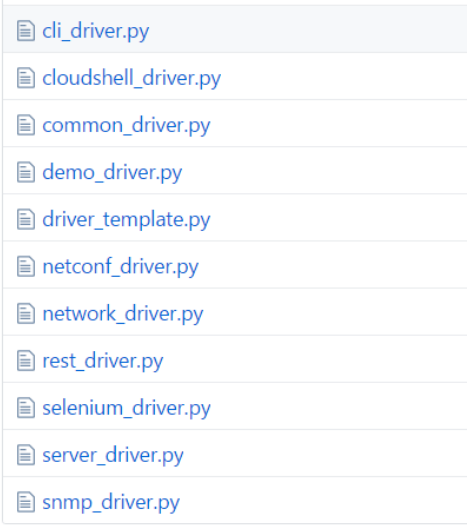
```
Warriorframework/warrior/Actions/SeleniumActions
```

This Selenium Actions Package currently has five files that have been created as such by grouping Selenium Keywords in accordance to the functionality they serve.

Now, the browser_actions.py has all the selenium keywords that deal with the browser aspect – like opening and closing a browser, maximizing and resizing it, opening new tabs and so on. The elementoperation_actions.py has selenium keywords related to the operations that can be performed on a webpage like clicking, typing text, double clicking, drag and drop and so on.

6.2 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 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 automating a command line interface based system.
cloudshell_driver	This product driver invokes execution of all keyword functions related to cloudshell.
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.
network_driver	Product driver that invokes execution of all keyword functions that can be used for all network keywords.
rest_driver	Product driver that invokes execution of all keyword functions that can be used for REST.

selenium_driver	Product driver that invokes execution of all keyword functions that can be used for all selenium keywords.
server_driver	Product driver that invokes execution of all keyword functions that can be used for all server related keywords
snmp_driver	Product driver that invokes execution of all SNMP keywords.

6.3 FRAMEWORK DIRECTORY

Warrior Framework'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 here:

```
warriorframework/docs/build/html/warrior_index.html
```

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

Further, the ClassUtils directory has subdirectories – WNetwork and WSelenium and several sub-files. All these 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 pre-requisite that Warrior Framework 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. xl_utils_class.py

This module has class and methods required to parse from and write to an excel workbook.

9. WNetwork

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

9.1 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)

9.2 connection.py

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

9.3 diagnostics.py

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

9.4 file_ops.py

This is the Warrior Framework 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.

9.5 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.

9.6 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.

10. WSelenium

The WSelenium package provides utilities for operations related to browser functionalities and element locators. It has the following files:

10.1 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.

10.2 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.

10.3 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.

10.4 Verify_operations

This is the verify operations library with functionalities for verifying HTML elements on a page.

10.5 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.

11. Utils Directory

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

11.1 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.

11.2 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.

11.3 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.

11.4 datetime_Utils.py

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

11.5 demo_Utils.py

This API is for demo purposes only.

11.6 dict_Utils.py

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

11.7 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 Framework.

11.8 email_Utils.py

Provides utility to send email using SMPT

11.9 encryption_Utils.py

This file provides utilities for data encryption.

11.10 file_Utils.py

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

11.11 import_Utils.py

This file provides utilities for package verifications and imports.

11.12 list_Utils.py

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

11.13 print_Utils.py

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

11.14 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 case or the datafile.

11.15 selenium_utils

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

11.16 string_Utils.py

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

11.17 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.

11.18 case_Utils.py

This module contains methods for case 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 Case, create tags with a particular name under the case tag, update the keyword result file, get value of impact, context, and

onError from the case, and more such functionalities. Please refer to the full documentation available in the warrior_docs repository.

11.19 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.

6.4 TOOLS DIRECTORY

The tools directory contains directories containing files specific to Warrior Frameworks integration with other tools. The following directories and files are contained under this directories:

6.4.1 THE DATABASE DIRECTORY

This directory contains information about the databases that you may want to use with Warrior.

6.4.2 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. More information about this can be found here.

6.4.3 THE JIRA DIRECTORY

The Jira directory contains a jira.xml file that can be configured with information about Jira projects that you may to upload defects to via Warrior after the execution finishes. More information about this can be found here.

6.4.4 THE REPORTING DIRECTORY

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

6.4.5 THE CONNNECTION DIRECTORY

The connection directory consists of files that help Warrior detect which CLI system it is communicating to and get relevant information from it, like, which software version it is currently running. More information about this can be found here.

6.4.6 THE XSD DIRECTORY

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

6.4.7 THE W_SETTINGS FILE

The w_settings file is used to set up the email notifications and also used to set the directories for log and result files. More information about the email notifications can be found here. More information about the logs and results directory setting can be found here.

6.5 WARRIOR CORE DIRECTORY

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

6.6 WARRIORSPACE DIRECTORY

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

6.6.1 CASES DIRECTORY

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

6.6.2 SUITES DIRECTORY

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

6.6.3 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 Framework for you to save your Projects. You may use a different location.

6.6.4 DATA DIRECTORY

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

6.6.5 CONFIG_FILES DIRECTORY

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

6.6.6 EXECUTION DIRECTORY

1. Warrior stores all the execution related information like the log and result files in the Execution directory by default.

2. The Execution directory will be automatically created by Warrior under Warriorspace directory. If the Execution directory already exists, Warrior will use the existing directory.

3. If a case, suite or project is being executed for the first time, the Warrior will create a subdirectory inside the Execution folder with the same name as the case, suite, or project, and all the related log and result files will be stored inside that directory

4. If a case, suite or project is being executed for the nth time, Warrior will create as subdirectory named after that case, suite, or project and then will append the time/date of the execution. Under this time/date stamped directory, the result and log files will be stored.

5. To change this default behavior and make Warrior store the execution information in a different directory, you will have to change the setting for it in the w_settings.xml file. More information about this can be found here.

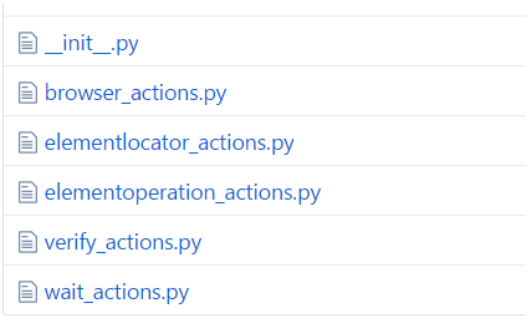
7.0 UNDERSTANDING THE CORE KEYWORDS

Warrior Framework provides the user with a set of built-in generic keywords also more commonly known as core keywords.

The documentation for these modules can be found here: warriorframework/docs/build/html/warrior_index.html

The keywords are located inside an Actions package which in turn are located inside the Actions directory.

An Actions package in Warrior looks something like:



This is the Selenium Actions package located in:

```
Warriorframework/warrior/Actions/SeleniumActions
```

This Selenium Actions Package currently has five files that have been created as such by grouping Selenium Keywords in accordance to the functionality they serve.

Now, the browser_actions.py has all the selenium keywords that deal with the browser aspect – like opening and closing a browser, maximizing and resizing it, opening new tabs and so on. The elementoperation_actions.py has selenium keywords related to the operations that can be performed on a webpage like clicking, typing text, double clicking, drag and drop and so on.

Warrior has other Actions packages to offer and they are located at:

```
Warriorframework/warrior/Actions
```

All the actions packages are described in detail below.

7.1 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.

7.2 CLOUDSHELL ACTIONS PACKAGE

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

7.3 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.

7.4 DEMO ACTIONS PACKAGE:

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

7.5 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.

7.6 NETWORK ACTIONS PACKAGE

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

7.7 REST ACTIONS PACKAGE

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

7.8 SELENIUM ACTIONS PACKAGE

The SeleniumActions package contains keywords related to Selenium operations.

7.9 SNMP ACTIONS PACKAGE

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

8.0 CREATING A CASE

A Warrior Case is a file that has a pre-defined structure established by Warrior Framework. This section gives a general overview of what a Warrior Case is made up of. To correctly design a Warrior Case, please refer to Section 13.0.

You can create this file through Katana – this has been explained in detail here.

A Case consists of three sections – Details, Steps, Requirements. We will go over each of these sections, one by one.

8.1 DETAILS

The section includes all the information necessary for Warrior to understand the Case. The details section consists of the following components:

8.1.1 CASE NAME

This is the Case name. It must be unique. The name of the file and the value of this field should match. This is a mandatory field.

8.1.2 CASE TITLE

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

8.1.3 ENGINEER

This is the Case developer's name. This is a mandatory field.

8.1.4 CATEGORY

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

8.1.5 DATE

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

8.1.6 TIME

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

8.1.7 STATE

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

This feature lets you create a Case while the keywords for this functionality are being simultaneously developed. You can either add a yet-to-be-developed driver and yet-to-be-developed keyword into this step OR if the driver has been developed but the keyword is still under development, then you can add the yet-to-be-developed keyword.

A Case has to be in the “Draft” state when such steps are added into it.

Creating a step containing yet-to-be-developed drivers and keywords is explained in Katana User Guide Section 3.14 and Section 3.15.

8.1.8 INPUT DATA FILE

Location of the data file to be used by the keyword. If the case 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 casename_data.xml in the Data directory of the Warriorspace.

8.1.9 DATA TYPE

This determines the interaction of the case with the data file. If Iterative is selected, the case data file needs to use the system data file format. The steps of the case 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 ‘Custom’.

8.1.10 DEFAULT ON ERROR

This is the default error handling for the case. This is where you set what happens if a case 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. The Default_OnError options are: Next, Goto, and Abort.

- Selecting Next will result in the next step being executed if the step fails.
- Selecting Goto and the step number to go to will result in execution of the step indicated. The execution of steps will continue from that step going forward.
- Selection Abort will terminate the execution of the case upon failure or error. The step status would be set to FAIL

a. The Actions attribute

This is an attribute of the Default_OnError. This is an optional attribute. If not set, Warrior will default to "next", that is, it would proceed to the next Case. The options for this attribute are next, abort, and goto.

b. 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 does not pass.

8.1.11 LOGSDIR

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.

8.1.12 RESULTSDIR

Location of directory where result files will be stored. If field is left blank, Warrior will use what is in the case, 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.

8.1.13 EXPECTED RESULTS

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

8.2 REQUIREMENTS

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

8.2.1 REQUIREMENT

Every requirement needed for this case must be added inside the requirements section. This is an optional field.

8.3 STEPS

8.3.1 STEP

You 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. You will need to indicate the product driver to be used in the step and the keyword. A step can be built from Katana. This has been explained in this section.

8.3.2 ARGUMENTS

Section within the step where the arguments are detailed.

a. Argument

The argument tag is where you can pass arguments directly to the keywords through the Case. 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 case.

8.3.3 ON ERROR

Set what happens if the step fails. This will override the default of the case. Options are Next, Goto, Abort and Abort_as_error. The behavior is as follows:

1. Selecting Next will result in the next step being executed if the step fails.

a. Going to the next step after a failure is observed in a step is the default behavior of Warrior.
b. Usage example: consider a scenario where you have already logged in onto a website and now you just want to make rest calls to its API and verify the data received from the rest calls. To achieve this, you have decided to use the REST Core Keywords provided by Warrior Framework. This case would be something like:
    - Send a REST request.
    - Compare the response captured in step 1 with the expected response.
    - Repeat steps 1 and 2 until all responses have been verified.
c. In the case like above, a failure in the verification of one response would not affect the next verification and hence the onError for each step can be set to “next”.
d. Creating this type of a step in Katana is explained here.

2. Selecting Goto and the step number to go to will result in execution of the step indicated. The execution of steps will continue from that step going forward.

    a. When the on error is set to GoTo, you would need to provide a step number to Warrior to go to. The on error of GoTo means that, if a failure is observed, Warrior would jump to executing the mentioned step.

    b. Usage example: If you are creating a Case that updates the software of a system whenever run. Now, when such sensitive Cases are developed, it is necessary to have a restore facility handy in case something goes wrong while updating the software. Now, the steps containing the restore commands should be triggered only when the steps executing the software upgrade fail. This can be done by executing the steps conditionally. Thus, when this Case runs perfectly, the restore steps would not be run at all.

    c. But, we would still need to trigger these if a failure is observed. Setting the on error to GoTo would ensure that the restore steps are called after a failure is observed.

    d. Creating this type of a step in Katana is explained here.

3. Selection Abort will terminate the execution of the case if the step does not pass. The step status would be set to FAIL

a. The execution of a Case can be aborted by setting the On Error in a step to abort. This prevents the remaining Case from running and thus unnecessary failures can be avoided.

b. For example: If you are connecting to CLI based system using Warrior. The first keyword would be connecting to the system and then the rest of the commands would follow. Now if the connection fails, the steps after that are bound to fail. Thus, here it makes sense to abort the execution of a Case as soon as step establishing the connection is observed. This can be achieved by setting the onError of each step to "abort".

c. Creating an abort on failure step in Katana is explained here.

4. Selection Abort_as_error will terminate the execution of the case if the step does not pass. The step status would be set to Error. Creating this type of a step in Katana is explained here.

This is an optional field.

8.3.4 DESCRIPTION

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.

8.3.5 ITERATION TYPE

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 case while the 'once_per_iter' mode lets that particular step for every iteration that the set of steps go through.

8.3.6 EXECUTE TYPE

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 Execute Type attribute has the options: If, If Not, Yes, No.
Creating a step which gets executed conditionally is explained in this section.

1. When the execute type for a step is set to "Yes", the step will get executed no matter what.

2. When the execute type for a step is set to "No", the step would not get executed under any circumstance. This option exists because when running a Case, you may not want to run some steps every time you run that Case.

3. When the execute type for a step is set to "No", Warrior will execute this step only *if a certain condition is met, otherwise it will skip this step and then* either *proceed to the next step* or *proceed to a specified step* or *abort execution of the Case*. Once the execute type has been set to "If", you would need a condition to go along with it. This condition can either be the result of a previously executed step, or a specific key-value pair that will be available in Warrior's datarepository.

    - If you want to execute this step based on the result of a previously executed step, the condition would be *step_<step-number>_result* and the value for this condition can be *PASS*, *FAIL*, *ERROR*, or *SKIPPED* depending upon when you want to execute this step.

    - If you want to execute this step based on a value of a key available in Warrior's datarepository, you will need to add the key *as it would be found in the datarepository* as the condition and its value *as you expect it to be* as the condition value.

    - An "else" for this "if" is also available in case the condition is not met. The value set in this else will let Warrior know what to do in case the condition is not met. You have the option of continuing the execution by executing the next step, or executing a particular step, or aborting the execution of the Case.

4. When the execute type for a step is set to "No", Warrior will execute this step only *if a certain condition is NOT met, otherwise it will skip this step and then* either *proceed to the next step* or *proceed to a specified step* or *abort execution of the Case*. Once the execute type has been set to "If Not", you would need a condition to go along with it. This condition can either be the result of a previously executed step, or a specific key-value pair that will be available in Warrior's datarepository.

    - If you want to execute this step based on the result of a previously executed step, the condition would be *step_<step-number>_result* and the value for this condition can be *PASS*, *FAIL*, *ERROR*, or *SKIPPED* depending upon when you want to execute this step.

- If you want to execute this step based on a value of a key available in Warrior's datarepository, you will need to add the key *as it would be found in the datarepository* as the condition and its value *as you expect it to be* as the condition value.

- An "else" for this "if" is also available in case the condition is met. The value set in this else will let Warrior know what to do in case the condition is met. You have the option of continuing the execution by executing the next step, or executing a particular step, or aborting the execution of the Case.

8.3.7 CONTEXT

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

1. Every step created is, by default, is in a positive context i.e., the end result of that step is reported as is to Warrior. If the step fails, then the end result is reported as a failure. If it passes, then the end result is reported as a pass.

2. Creating a step in the positive context in Katana is explained here.

3. A step can be evaluated in the negative context i.e., the end result of that step is flipped and then reported to Warrior. If the step fails, then the end result is reported as a pass. If it passes, then the end result is reported as a fail.

    - For example: If you know that a failure is expected as the result of executing a step that means that a failure actually means that the execution of the step went correctly. Such a step can be created in the negative context (by setting the value of the context to 'negative') such that the end result (in this case a FAIL) will be evaluated as a PASS. Please note that the end result of a "PASS" will be evaluated as a "FAIL".

4. Creating a step in the negative context in Katana is explained here.

8.3.8 IMPACT

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

1. Every step created is, by default, an impacting step i.e., the end result of that step affects the end result of the Case. So, if the step fails or errors out, the Case is marked as failed.

2. Creating an impacting step in Katana is explained here.

3. A step can be marked as a non-impacting step i.e., the end result of that step would not affect the end result of the Case. So, if a non-impacting step fails or errors out, the end status of the Case is not affected by it.

4. Creating a non-impacting step in Katana is explained in here.

8.3.9 RUN MODE

This tag can be filled up to run a case 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.

1. When the Runmode for a step is set to "RMT", you will have to provide the number of times that step should run. Warrior would then execute this step those many number of times. If you do not provide Warrior with the number of times the step should be run, then this step would run once. Creating an RMT step in Katana is explained here.

2. When the Runmode for a step is set to "RUF", you will have to provide the maximum number of times that step should run. Warrior would then keep on executing this step as long it passes and will stop as soon as it fails. If the step reaches the maximum number of attempts, Warrior would stop executing the RUF step and proceed to the next step in the Case. Creating an RUF step in Katana is explained here.

3. When the Runmode for a step is set to "RUP", you will have to provide the maximum number of times that step should run. Warrior would then keep on executing this step as long it fails and will stop as soon as it passes. If the step reaches the maximum number of attempts, Warrior would stop executing the RUP step and proceed to the next step in the Case. Note: If the RUP step does

not pass in the first attempt and hence failures are seen, it will be marked at FAIL even though the step eventually passes – this is to alert the Case developer that somewhere a failure has occurred. Creating an RUP step in Katana is explained here.

9.0 CREATING A SUITE

THROUGH CLI

You can create a suite from existing Cases through the command line

Warrior Framework offers CLI support for creation of a default suite from a list of cases.

Arguments:

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

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

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

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

`-tcdir` (optional): The case directory

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

Format:

```
./Warrior -createsuite -suitename suite_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 suite_file_name -suitelocn location_to_create_test_suite_xml file -tcdir path/to/dir1 path/to/dir2 path/to/dir3
```

Commands:

Creates a suite with provided name and with the provided list of cases in the current working directory, with default values for other suite parameters.

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

Creates a suite with provided name and provided list of cases in the provided suitelocn (suite location), with default values for other suite parameters.

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

Create a suite with provided name in the current working directory. Search the current working directory for cases matching at least one of the provided categories cat1 cat2 cat3.

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

Search the current working directory for cases matching at least one of the provided categories cat1 cat2 cat3.

Create a suite with provided name in the provided suitelocn.

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

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

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

THROUGH KATANA

A Warrior Suite is a file that has a pre-defined structure established by Warrior Framework. A Suite consists of a collection of Cases. This section gives a general overview of what a Warrior Suite is made up of. To correctly design a Warrior Suite, please refer to Section 14.0. A Suite can be created using Katana. Please refer to this section a complete guide to creating a suite using Katana.

There are three sections in a Suite: Details, Cases, and Requirements. Each section has been described in detail below:

9.1 DETAILS

This section includes information about the Suite that Warrior would need to execute it. This sections has the following fields in it:

9.1.1 SUITE NAME

This is the Suite name. It must be unique. The name of the file value of this field should match. This is a mandatory field.

9.1.2 SUITE TITLE

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

9.1.3 ENGINEER

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

9.1.4 DATE

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

9.1.5 TIME

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

9.1.6 TYPE

The exectype attribute is used to determine how the 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 case does not fail. The default exectype is sequential.

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

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

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

9.1.7 STATE

This field 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 Case. This is an optional field.

9.1.8 DEFAULT ON ERROR

Default error handling for the suite. This is where you set what happens if a case fails. User can override the default by selecting the on error in the case section. If the case 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 case. You will be able to set different onError setting per case if needed. That will be done at the case 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 case number, that is, the number of the case that you want to jump to in case the current case throws an Error. Warrior will then directly go to executing the case corresponding to the number given if the current case throws an Error.

9.1.9 RESULTSDIR

Location of directory where result files will be stored here. If field is left blank Warrior will use what is in the case, 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.

9.1.10 INPUT DATA FILE

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

9.2 REQUIREMENTS

User can add as many requirements as needed.

9.2.1 REQUIREMENT

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

9.3 CASES

User can add as many cases as needed but at least one case is required.

9.3.1 CASE PATH

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

9.3.2 CONTEXT

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

9.3.3 RUN TYPE

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

9.3.4 ON ERROR

Set what happens if the case fails. This will override the default of the suite. Set what happens if the step fails. This will override the default of the suite. Options are Next, Goto, Abort and Abort_as_error. The behavior is as follows:

- Selecting Next will result in the next case being executed if the case fails.
- Selecting Goto and the case number to go to will result in execution of the case indicated. The execution of cases will continue from that case going forward.
- Selection Abort will terminate the execution of the suite if the case does not pass. The case status would be set to FAIL

- Selection Abort_as_error will terminate the execution of the suite if the case does not pass. The case status would be set to ERROR.
- This is an optional field.

9.3.5 RUN MODE

This field has two attributes - “type” and “value”. The value provided to the type attribute indicates whether this particular case 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 case should go through before completing the execution of this case – 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 case till it passes/fails, while for RMT, Warrior would run the case for as many times as indicated. This is an optional field.

9.3.6 IMPACT

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

9.3.7 EXECUTE TYPE

This is an optional field that decides when a Case should be executed. If this field is not set, the default will be to execute the Case. The Execute Type attribute has the options: If, If Not, Yes, No.
If: The Case will be executed if the condition below is met, if the condition is not met, the Else action will be executed.
If Not: The Case will be executed if the condition below is not met, if the condition is met, the Else action will be executed.
Yes: The Case will be executed without any conditions.
No: The Case will not be executed.

10.0 CREATING A PROJECT

A Warrior Project is a file that has a pre-defined structure established by Warrior Framework. A Project consists of a collection of Suites. This section gives a general overview of what a Warrior Project is made up of. A Project can be created using Katana. Please refer to this section a complete guide to creating a project using Katana.

There are two sections in a Suite: Details and Suites. Each section has been described in detail below:

10.1 DETAILS

This section contains all the details about the Project. The following are its sub-sections:

10.1.1 PROJECT NAME

This sub-section contains the project name. It must be unique. This name should match the file name. This is a mandatory field.

10.1.2 PROJECT TITLE

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

10.1.3 ENGINEER

This is the name of the Project designer. This is a mandatory field.

10.1.4 PROJECT STATE

This is 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 Case. This is an optional field.

10.1.5 DATE

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

10.1.6 TIME

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

10.1.7 DEFAULT ON ERROR

This is the default behavior if any of the suite fails. User can override the default by selecting the on error in the suite section. If the suite 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 suite. You will be able to set different onError setting per suite if needed. That will be done at the suite 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 suite number, that is, the number of the suite that you want to jump to in case the current suite throws an Error. Warrior will then directly go to executing the suite corresponding to the number given if the current suite throws an Error.

10.2 SUITE

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

10.2.1 SUITE PATH

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

10.2.2 ON ERROR

Set what happens if the suite fails. This will override the default of the project. Options are Next, Goto, Abort and Abort_as_error. The behavior is as follows:

- Selecting Next will result in the next suite being executed if the suite fails.

- Selecting Goto and the suite number to go to will result in execution of the suite indicated. The execution of suites will continue from that suite going forward.
- Selection Abort will terminate the execution of the suite if the suite does not pass. The suite status would be set to FAIL
- Selection Abort_as_error will terminate the execution of the suite if the step does not pass. The suite status would be set to ERROR.
- This is an optional field.

10.2.3 IMPACT

This will determine if the failure of this 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'.

10.2.4 EXECUTE TYPE

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

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

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

Yes: The Suite will be executed without any conditions.

No: The Suite will not be executed.

11.0 CREATING A 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 Case.

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.

12.0 THE DATA REPOSITORY

- Warrior provides a data repository within a case, suite, and project. The data repository within a 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 case.
- Warrior provides a data repository within a suite as well. The data repository within a suite contains all the case statuses. The status of each executed case in that suite is available for every other case 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 expects all data returned by a keyword should be returned in a python dictionary. This returned dictionary will be stored in the case, suite, or project data repository as a key-value pair.
- Keywords will have to access the case data repository to get the data returned from a previous keyword, cases in a suite will have access to case statuses returned from the previous cases, 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 case 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')
```

13.0 DESIGNING A CASE

A case should be designed according to what is needed out of it. If the purpose of a Case is to test a software, a product, or an environment, then the Case should be designed in a way that manual intervention should not be required to determine if a Case was successful in testing the software, product, or environment. The "PASS" at the end should be enough to determine that the Case was successful.

Warrior provides you with various different ways to design the Case to suit your requirements.

13.1 DESIGNING AN ITERATIVE CASE

An iterative Case requires an Input Data File to be included in that Case. To design an iterative Case, open a Case in Katana and implement the following steps to make a Case Iterative:

i. Include an Input Data File in the Case by providing a relative path to the data file in the field for "Input Data File" in the Details section of the Case

ii. Set the Data Type field in the Details section of a Case to "Iterative"

iii. Now, there is no need to provide system names in the arguments to the keywords in a step as all the steps would be executed on all the systems in that data file

13.2 DESIGNING A CUSTOM CASE

A custom Case does not require an Input Data File but it accepts it. To design a Custom Case, open a Case in Katana and implement the following steps to make a Case Iterative:

1. Include an Input Data File in the Case by providing a relative path to the data file in the field for "Input Data File" in the Details section of the Case. If you don't want to include a Data file, check the "I Don't Want to Include a Data File" box. This would let Warrior know that an Input Data File has not been included

2. Set the Data Type field in the Details section of a Case to "Custom"

3. Now, it is required to enter the system name in the arguments to the keywords in all the steps. That step would then run on that particular system.

13.3 DESIGNING A HYBRID CASE

To design a hybrid Case, open a Case in Katana and implement the following steps to make a Case Hybrid:

1. Change the Data Type field in the Details Section of a Case to “Hybrid”
2. Now every step in that Case would have a new field “Iteration Type”
3. If you want to run a particular step as a standard iterative step, the set the “Iteration Type” to “standard”.
4. If you want to run a particular step only once throughout the Case, the set the “Iteration Type” to “once_per_tc”. If you want to run a particular step after all the other steps in a Case have been executed, the set the “Iteration Type” to “end_of_tc”. If there are multiple “end_of_tc” steps in a Case, then all of them will be executed at the end and the order in which they would be executed would be the order in which they appear in a Case.

13.4 DESIGNING A DRAFT CASE

This feature lets you create a Case while the keywords for this functionality are being simultaneously developed. You can also create a regular Case and make it a Draft Case if you don’t want anyone to run this Case.

You can either add a yet-to-be-developed driver and yet-to-be-developed keyword into at least one step in the Case OR if the driver has been developed but the keyword is still under development, then you can add the yet-to-be-developed keyword.

This Case is then considered a “Draft” Case when such steps are added into it.

Creating a step containing yet-to-be-developed drivers and keywords is explained in Katana User Guide Section 3.14 and Section 3.15.

Warrior would not run a Draft Case.

To mark a Case as Draft, open a Case in Katana and implement the following steps:

1. Change the Case State field in the Details Section of a Case to “Draft”.

13.5 DESIGNING A CASE WITHOUT USING AN INPUT DATA FILE

A custom Case does not require an Input Data File. So, if you don't want to add a data file into the Case, you will have to create a Custom Case. To design such a Case, open a Case in Katana and implement the following steps to make a Case Iterative:

1. Check the "I Don't Want to Include a Data File" box included along with the "Input Data File" field in Details section of the Case. This would let Warrior know that an Input Data File has not been included

2. Set the Data Type field in the Details section of a Case to "Custom"

3. Now, all the necessary values to the arguments of a keyword would have to be passed through the Case.

13.6 DESIGNING A CONDITIONAL CASE

13.6.1 EXECUTE TYPE: YES

When the execute type for a Case is set to "yes", the Case will get executed no matter what.

Follow the steps below set the Execute Type of a Case to "Yes":

1. Create a case containing the keywords you want.

2. Create a suite or modify an existing one.

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

4. Set the Execute Type to Yes.

13.6.2 EXECUTE TYPE: NO

When the execute type for a Case is set to "no", the step would not get executed under any circumstance. This option exists because when running a Suite, you may not want to run some cases every time you run that Suite.

Follow the steps below set the Execute Type of a Case to "No":

1. Create a case containing the keywords you want.

2. Create a suite or modify an existing one.

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

4. Set the Execute Type to No.

13.6.3 EXECUTE TYPE: IF

This is the programming equivalent of an "If" statement in Warrior– *If a certain condition is met, execute this Case, otherwise skip this Case and then* either *proceed to the next Case* or *proceed to a specified Case* or *abort execution of the Suite*. Once the execute type has been set to "If", you would need a condition to go along with it. This condition can has to be the result of a previously executed Case. This previously executed Case has to be inside the current Suite and cannot be from a different Suite.

To execute this Case based on the result of a previously executed Case, the condition would be *testcase_<case-number>_status* and the value for this condition can be *PASS*, *FAIL*, *ERROR*, or *SKIPPED* depending upon when you want to execute this Case.

An "else" for this "if" is also available in case the condition is not met. The value set in this else will let Warrior know what to do in case the condition is not met. You have the option of continuing the execution by executing the next Case, or executing a particular Case, or aborting the execution of the Suite.

Follow the steps below set the Execute Type of a Case to "If":

1. Create a case containing the keywords you want.

2. Create a suite or modify an existing one.

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

4. Set the Execute Type to If.

5. Fields for Condition, Condition Value, and Else would appear in Katana. Type the condition in the format *testcase_<case-number>_status*, the Condition Value as either *PASS*, *FAIL*, *ERROR*, or *SKIPPED*, and then set the Else.

13.6.4 EXECUTE TYPE: IF NOT

This is the programming equivalent of an "! if" statement in Warrior – *If a certain condition is NOT met, execute this Case, otherwise skip this step and then* either *proceed to the next Case* or *proceed to a specified Case* or *abort execution of the Suite*. Once the execute type has been set to "If Not", you would need a condition to go along with it. This condition has to be the result of a previously executed Case. This previously executed Case has to be inside the current Suite and cannot be from a different Suite.

To execute this step based on the result of a previously executed step, the condition would be *testcase_<case-number>_status* and the value for this condition can be *PASS*, *FAIL*, *ERROR*, or *SKIPPED* depending upon when you want to execute this Case.

An "else" for this "if" is also available in case the condition is met. The value set in this else will let Warrior know what to do in case the condition is met. You have the option of continuing the execution by executing the next Case, or executing a particular Case, or aborting the execution of the Suite.

Follow the steps below set the Execute Type of a Case to "If Not":

1. Create a case containing the keywords you want.
2. Create a suite or modify an existing one.
3. In the case block of the Suite, add the case created in step 1
4. Set the Execute Type to If Not.
5. Fields for Condition, Condition Value, and Else would appear in Katana. Type the condition in the format *testcase_<case-number>_status*, the Condition Value as either *PASS*, *FAIL*, *ERROR*, or *SKIPPED*, and then set the Else.

13.7 DESIGNING STEPS IN A CASE TO RUN SEQUENTIALLY

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

1. Create a case to contain the keywords that need to be run in sequence.
2. Create a suite or modify an existing one.

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

4. Set the runtype tag to “sequential_keywords”.

13.8 DESIGNING STEPS IN A CASE TO RUN IN PARALLEL

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

1. Create a case to contain the keywords that need to be run in parallel.

2. Create a suite or modify an existing one.

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

4. Set the runtype tag to “parallel_keywords”.

13.9 DESIGNING A CASE TO RUN MULTIPLE TIMES

When the Runmode for a Case is set to “RMT”, you will have to provide the number of times that step should run. Warrior would then execute this Case those many number of times. If you do not provide Warrior with the number of times the Case should be run, then this Case would be treated as a Standard Case.

An RMT Case is a Standard Case executed multiple times. This Case would run those many times whether it passes, fails, or errors out. Just like a standard Case, whether every attempt in the step will get executed or not will depend on the kind of execute type set. All attempts will report the status back depending upon the context and all attempts can either be impacting or non-impacting.

Such a functionality is usually used to test the performance of the system under test. Warrior has the ability to keep to running and then re-running a step so as to tax the system and see if any failures occur.

To create a Case with this behavior, implement the following steps:

1. Create a case containing the keywords that you need.

2. Create a suite or modify an existing one.

3. In the case block of the Suite, add the Case created in step 1

4. Set the Run Mode field to “RMT”.

5. Set the Max Attempts for RMT

13.10 DESIGNING A CASE TO RUN UNTIL IT FAILS

When the Runmode for a Case is set to “RUF”, you will have to provide the maximum number of times that Case should run. Warrior would then keep on executing this Case as long it passes and will stop as soon as it fails. If the Case reaches the maximum number of attempts, Warrior would stop executing the RUF Case and proceed to the next step in the Case.

The Case would be re-run only if the Case passes. If the Case fails or errors out, the Case would be marked as such and then execution of this Case will be stopped so as to proceed to the execution of the next Case.

Just like a standard Case, whether every attempt in the Case will get executed or not will depend on the kind of execute type set. It is recommended against making this a non-impacting or a negative context Case.

Such a functionality is usually used to test the performance of the system under test. Warrior has the ability to keep to running and then re-running a Case so as to tax the system and see if any failures occur.

To create a Case with this behavior, implement the following steps:

1. Create a case containing the keywords that you need.

2. Create a suite or modify an existing one.

3. In the case block of the Suite, add the Case created in step 1

4. Set the Run Mode field to “RUF”.

5. Set the Max Attempts for RUF

13.11 DESIGNING A CASE TO RUN UNTIL IS PASSES

When the Runmode for a Case is set to “RUP”, you will have to provide the maximum number of times that Case should run. Warrior would then keep on executing this Case as long it fails and will stop as soon

as it passes. If the Case reaches the maximum number of attempts, Warrior would stop executing the RUP Case and proceed to the next step in the Suite.

The Case would be re-run only if the Case fails. If the Case passes or errors out, the step would be marked as such and then execution of this Case will be stopped so as to proceed to the execution of the next Case.

Just like a standard Case, whether every attempt in the Case will get executed or not will depend on the kind of execute type set. It is recommended against making this a non-impacting or a negative context Case.

Such a functionality is usually used to test or automate a system that still has some bugs and is not stable and it is expected for the system to fail sometimes because of issues known or unknown.

Note: If the RUP Case does not pass in the first attempt and hence failures are seen, it will be marked at FAIL even though the Case eventually passes – this is to alert the Suite developer that somewhere a failure has occurred.

To create a Case with this behavior, implement the following steps:

1. Create a case containing the keywords that you need.
2. Create a suite or modify an existing one.
3. In the case block of the Suite, add the Case created in step 1
4. Set the Run Mode field to “RUP”.
5. Set the Max Attempts for RUP

13.12 DESIGNING AN IMPACTING CASE

Every Case created is, by default, an impacting Case i.e., the end result of that Case affects the end result of the Suite. So, if the Case fails, the Suite is marked as failed.

To create a Case with this behavior, implement the following steps:

1. Create a case containing the keywords that you need.

2. Create a suite or modify an existing one.

3. In the case block of the Suite, add the Case created in step 1

4. Set the Impact field to “impact”.

13.13 DESIGNING A NON-IMPACTING CASE

A Case can be marked as a non-impacting Case i.e., the end result of that Case would not affect the end result of the Suite. So, if a non-impacting Case fails, the end status of the Suite is not affected by it.

To create a Case with this behavior, implement the following steps:

1. Create a case containing the keywords that you need.

2. Create a suite or modify an existing one.

3. In the case block of the Suite, add the Case created in step 1

4. Set the Impact field to “noimpact”.

13.14 DESIGNING A CASE IN A POSITIVE CONTEXT

Every Case created is, by default, is in a positive context i.e., the end result of that Case is reported as is to Warrior. If the Case fails, then the end result is reported as a failure. If it passes, then the end result is reported as a pass.

To create a Case with this behavior, implement the following steps:

1. Create a case containing the keywords that you need.

2. Create a suite or modify an existing one.

3. In the case block of the Suite, add the Case created in step 1

4. Set the Context field to “positive”.

13.15 DESIGNING A CASE IN A NEGATIVE CONTEXT

Negative cases are cases that are expected to fail. If a negative case fails, the cases will be marked as a Pass in results. All cases in Warrior are positive cases by default unless marked as negative. To create a Case with this behavior, implement the following steps:

1. Create a case containing the keywords that you need.
2. Create a suite or modify an existing one.
3. In the case block of the Suite, add the Case created in step 1
4. Set the Context field to “Negative”.

13.16 ON ERROR ACTIONS FOR CASES

13.16.1 ON ERROR: ABORT

The execution of a Suite can be aborted by setting the On Error Action in a Case to abort. This prevents the remaining Cases from running and thus unnecessary failures can be avoided.

To create an aborting Case, implement the following steps:

1. Create a case containing the keywords that you need.
2. Create a suite or modify an existing one.
3. In the case block of the Suite, add the Case created in step 1
4. Set the On Error field to “abort”.

13.16.2 ON ERROR: NEXT

Going to the next Case after a failure is observed in a Case is the default behavior of Warrior.

To create a Case with this behavior, implement the following steps:

5. Create a case containing the keywords that you need.
6. Create a suite or modify an existing one.

7. In the case block of the Suite, add the Case created in step 1

8. Set the On Error field to “next”.

13.16.3 ON ERROR: ABORT_AS_ERROR

The execution of a Suite can be aborted by setting the On Error in a Case to abort_as_error but instead of marking the Case as a failure, its status would be reported as an error.

To create a Case with this behavior, implement the following steps:

1. Create a case containing the keywords that you need.

2. Create a suite or modify an existing one.

3. In the case block of the Suite, add the Case created in step 1

4. Set the On Error field to “abort_as_error”.

13.16.4 ON ERROR: GOTO

When the on error is set to GoTo, you would need to provide a Case number to Warrior to go to. The on error of GoTo means that, if a failure is observed, Warrior would jump to executing the mentioned Case.

To create a Case with this behavior, implement the following steps:

1. Create a case containing the keywords that you need.

2. Create a suite or modify an existing one.

3. In the case block of the Suite, add the Case created in step 1

4. Set the On Error field to “goto”.

5. Specify the Case number to go to.

14.0 DESIGNING A WRAPPER FILE

Wrapper file is designed to keep setup and cleanup steps of the testcase or testsuite.

Wrapper file has Details, Setup and Cleanup sections,

- ✓ Details Section :
    1. User can specify Name, Title, Date, Time, Engineer
    2. Datatype – custom/iterative/hybrid
    3. Runtype – sequential/parallel

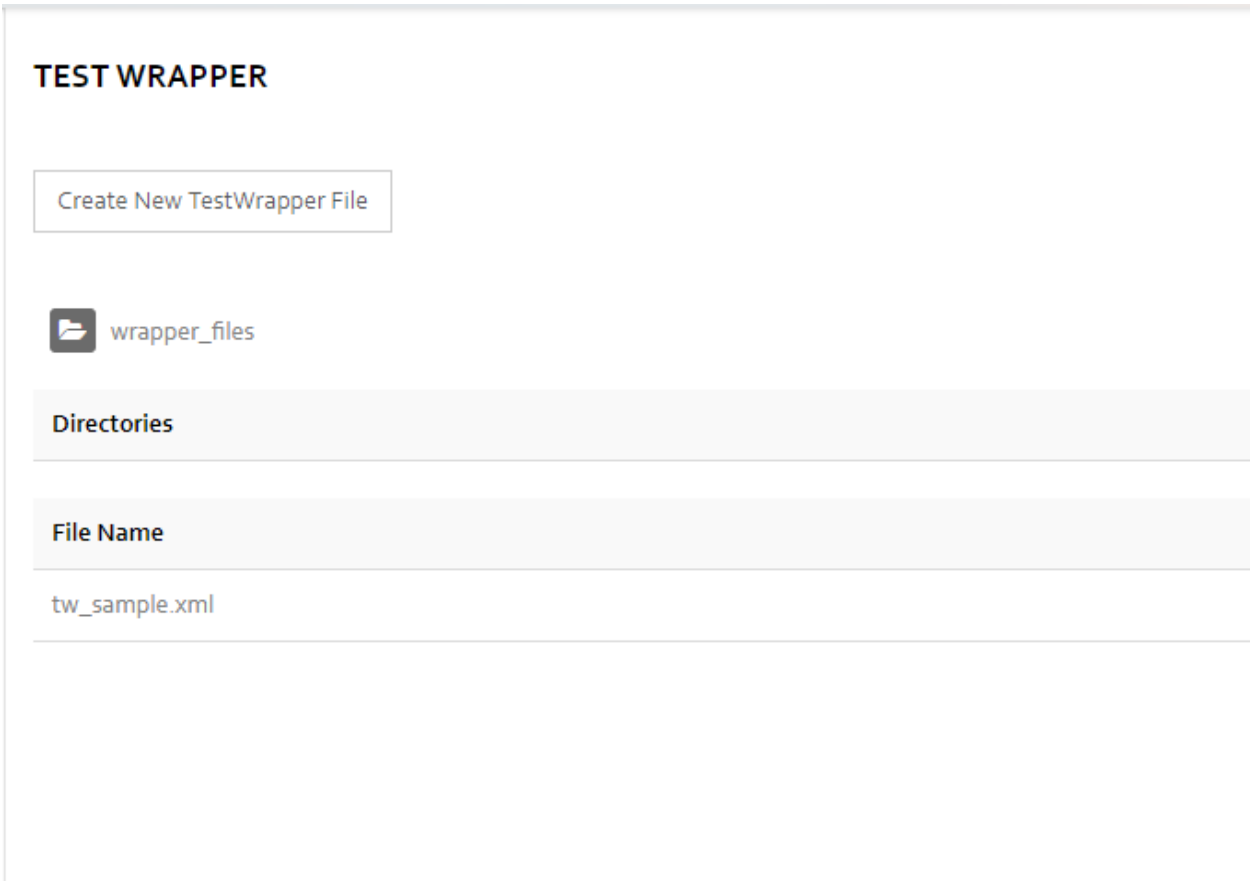
- ✓ Setup: All the setup steps should be included here, which are prerequisite for executing testcase or testsuite. These steps (setup steps) are executed before test steps execution

- ✓ Cleanup: All the cleanup steps should be included here. These steps (cleanup steps) are executed after testcase steps execution

```
<TestWrapper>
    <Details>
        <Name> </Name>
        <Title> </Title>
        <Datatype> </Datatype>
        <Runtype> </Runtype>
        <Date></Date>
        <Time></Time>
        <Engineer> </Engineer>
    </Details>

    <Setup>
        <step Driver=" " Keyword=" " TS="">
        </step>
        <step Driver=" " Keyword="" TS="">
        </step>
    </Setup>
    <Cleanup>
        <step Driver=" " Keyword=" " TS="">
        </step>
        <step Driver=" " Keyword=" " TS="">
        </step>
    </Cleanup>
</TestWrapper>
```

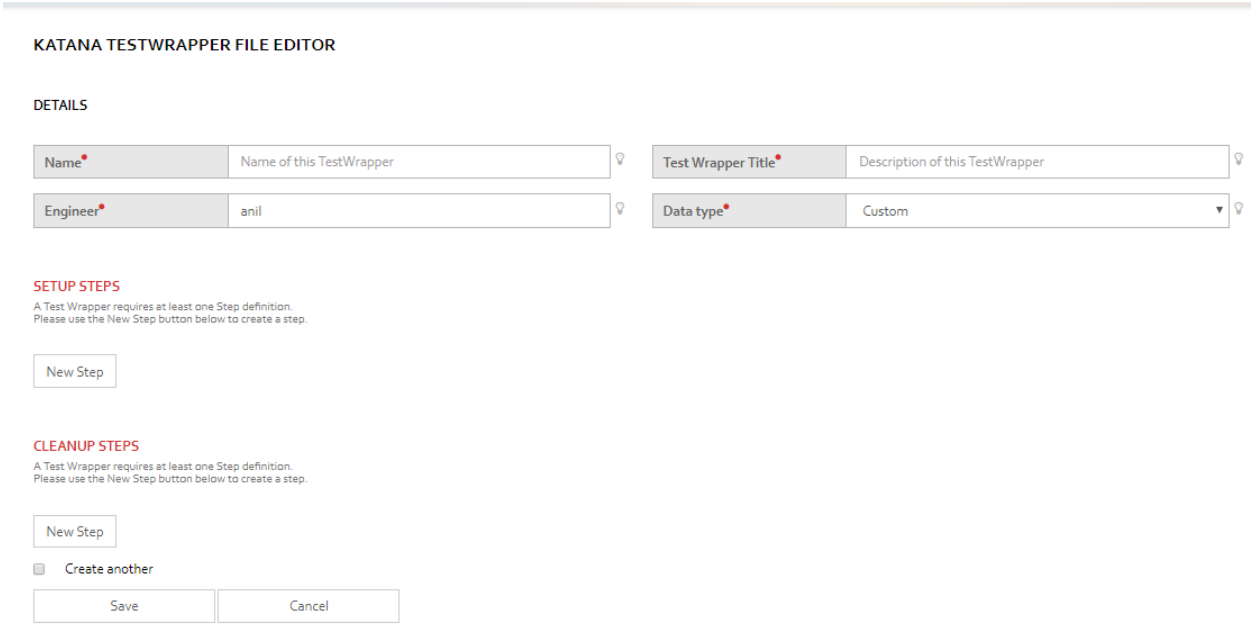
15.0 CREATING A WRAPPER FILE FROM KATANA

Katana allows you to create wrapper files through its user interface. You have to start up Katana and go to its test wrapper file tab. There, you will see a “Create New TestWrapperFile” button.



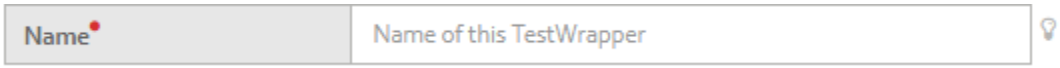
Note: The list of wrapper files below the “Create New TestWrapperFile” button may differ as Katana would show the wrapper files saved in your wrapper file directory.

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

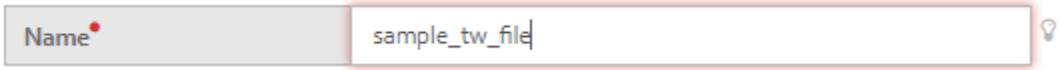


A step by step guide to create a wrapper file is Katana is given below

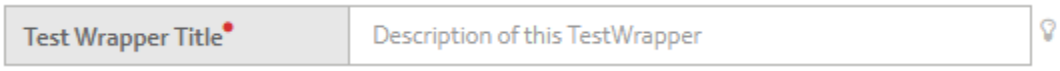
15.1 WRAPPER FILE NAME



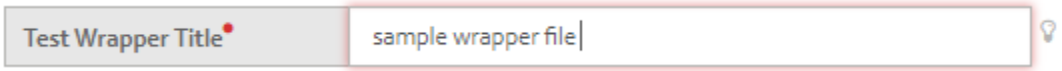
Here enter the name of the wrapper file. A name can be anything that you recognize. This is a mandatory field.



15.2 WRAPPER FILE TITLE



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

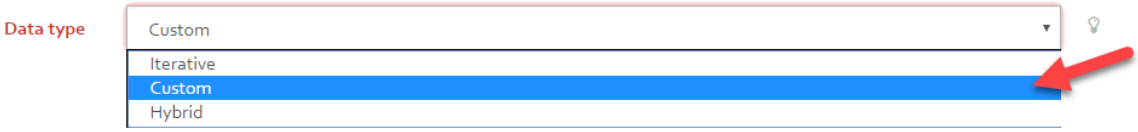


15.3 ENGINEER

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

15.4 DATA TYPE

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



15.5 STEPS

We have two sections in steps

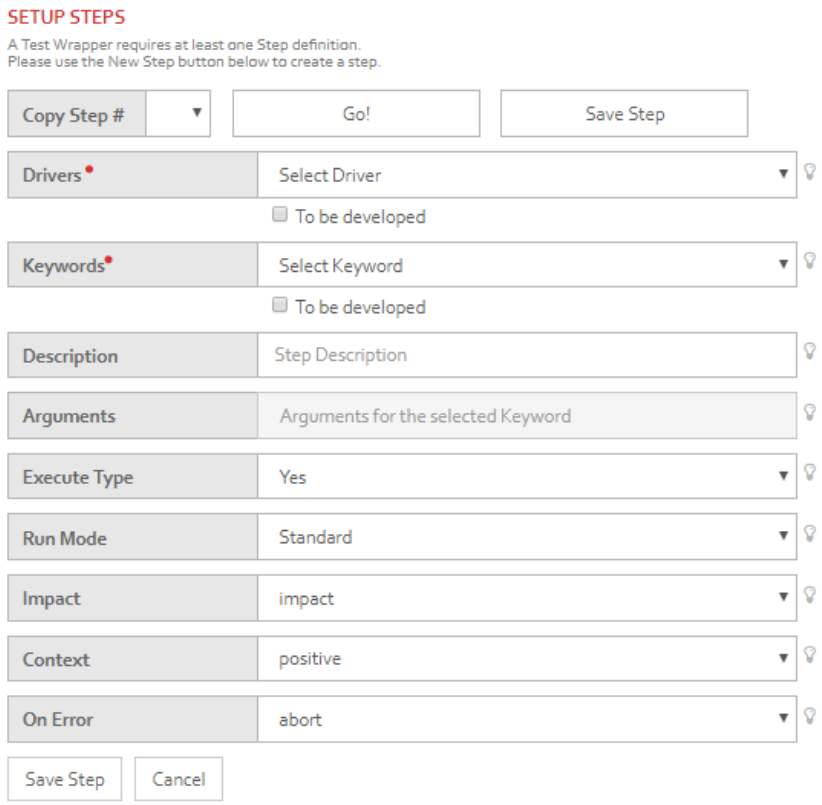
1) Setup steps

2) Cleanup steps

15.6 SETUP STEPS

Setup steps are optional. A new setup step can be added to the wrapper file setup section by clicking the "New Step" button.

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”.

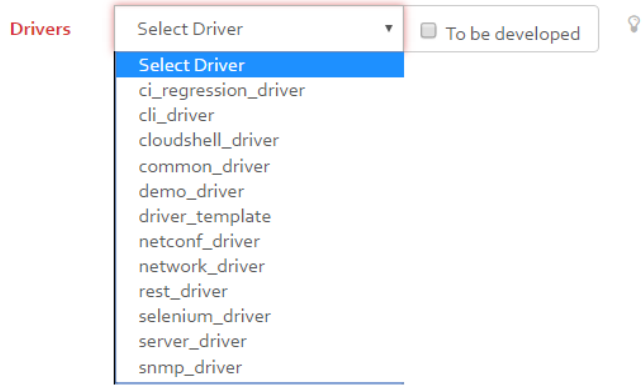


15.7 DRIVERS

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

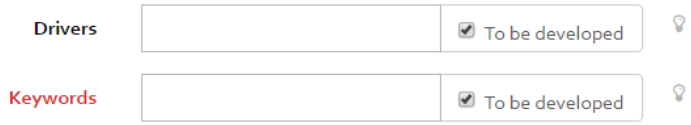
Katana lets you select a driver by showing all the drivers that you have in your Warrior Framework directory in the dropdown. You can select any driver out of those. You would not be able to select any driver which lies outside of the ProductDrivers directory. This is because Warrior enforces a strict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



You can then select any driver from that list:

If you want to include a Driver that has not been developed yet, you can simply check the “To be developed” box and add a driver name of your own.



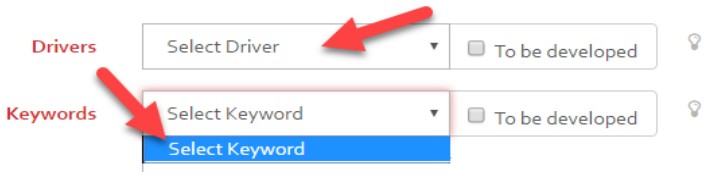
Be informed that checking the “To Be Developed” box for the driver automatically checks the “To Be Developed” box for the Keyword as a keyword for a non-developed driver cannot exist,.

You would then be able to include a driver name of your choice. You can also leave this field blank as checking the “To Be Developed” box renders the Drivers field as non-mandatory.

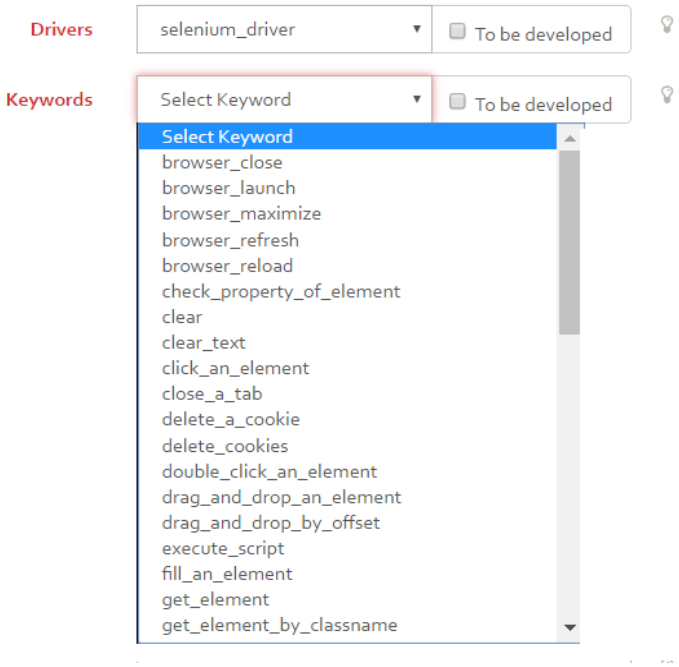
15.8 KEYWORDS

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.



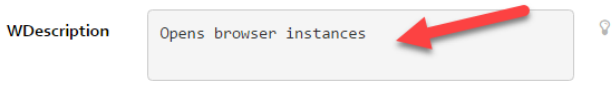
You can then select any keyword that you need.

If you want to include a Keyword that has not been developed yet, you can simply check the “To be developed” box and add a keyword name of your own.

A keyword name – even when you are including a non-developed keyword – is a mandatory field.

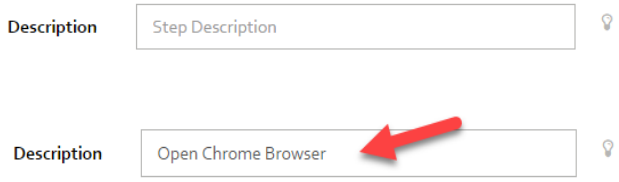
15.9 WDESCRIPTION

This field is populated when the Keyword is chosen. The description is for your reference, so that, you know exactly what this keyword does. If a non-developed keyword is chosen, this field would remain empty. This field is a non-editable field.



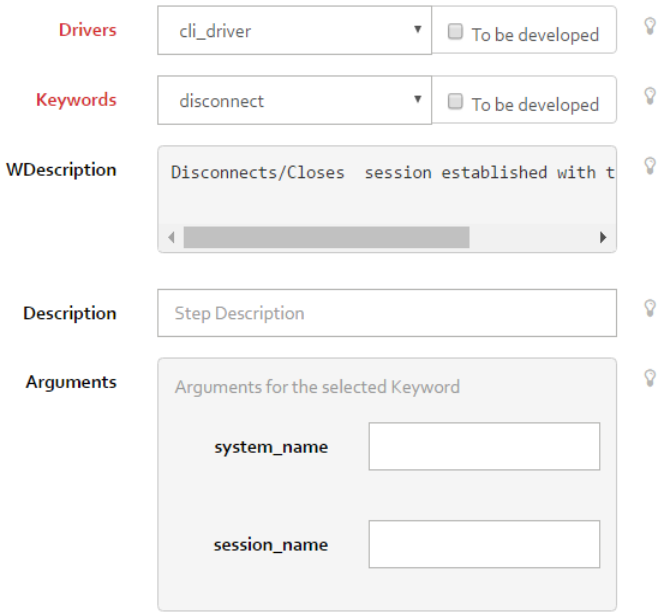
15.10 DESCRIPTION

This field is provided to you so that you can add a description of your own for the chosen Keyword. This description would be displayed when this steep is run. This is an optional field.

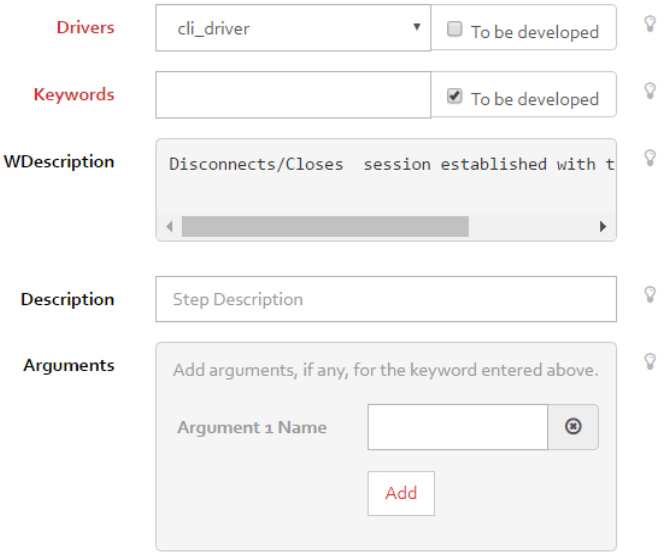


15.11 ARGEMENTS

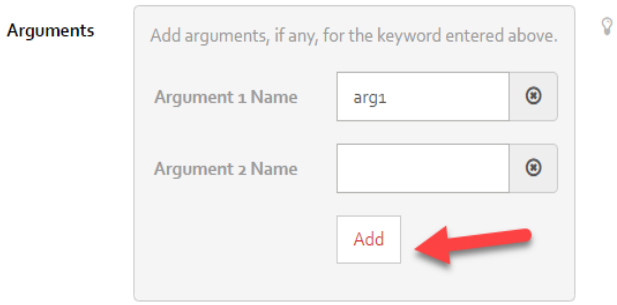
This space gets populated when a developed keyword is selected. All the arguments accepted by the Keyword show up here. You can type in the values associated with each value against that argument name. To identify which are the mandatory arguments, you can go through the next section.



When a non-developed keyword is selected, this field is populated a little differently.

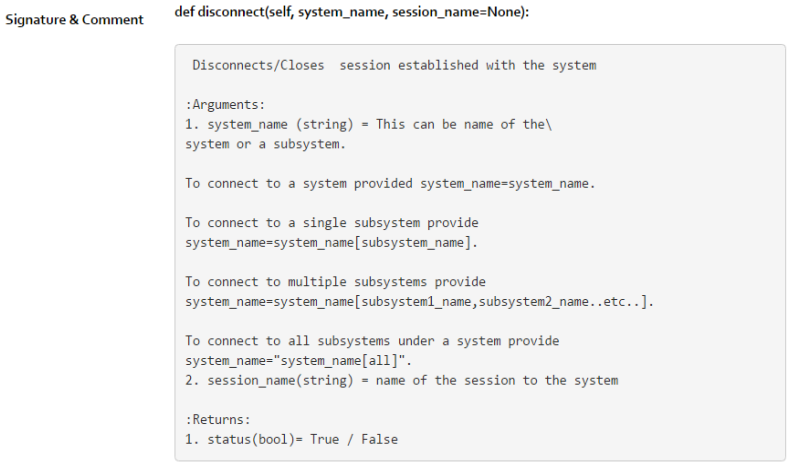


Here, in the space where Arguments for a developed keyword should show up, input fields that will let you define the arguments that you think this keyword should have would be displayed. You can then add the arguments names for the non-developed keyword.



15.12 SIGNATURE & COMENTS

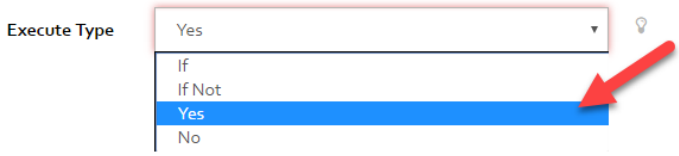
This field also appears prepopulated when a developed Keyword is selected. This is for your benefit, so that you understand what the keyword does when it runs, what kind of arguments it accepts, and what kind of values it returns.



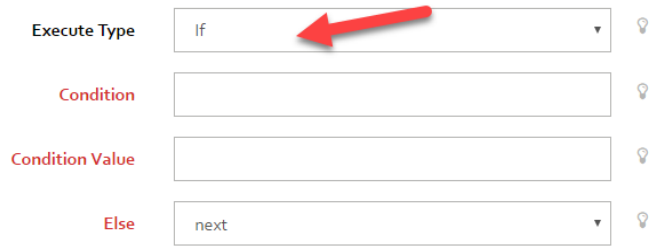
This field would remain blank when a non-developed keyword is selected.

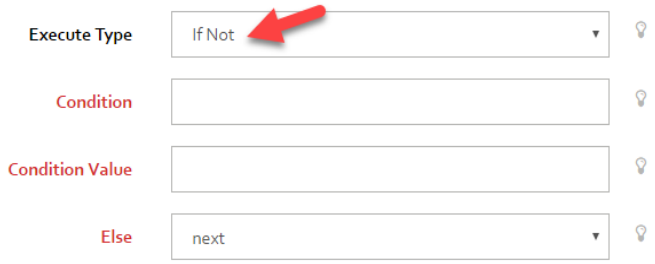
15.13 EXECUTE TYPE

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.

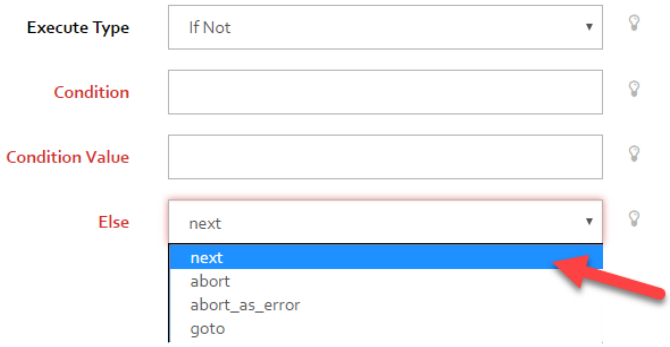


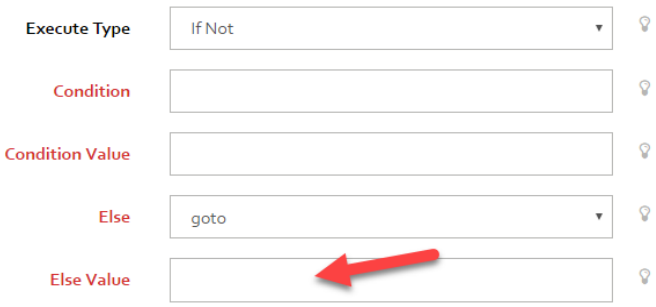
On clicking on 'If' and 'If Not', two other input boxes show up that let you add your conditions.





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.

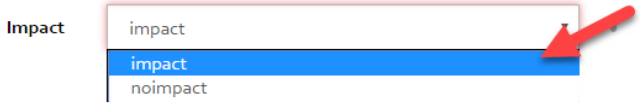




To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.1.

15.14 IMPACT ON FAILURE

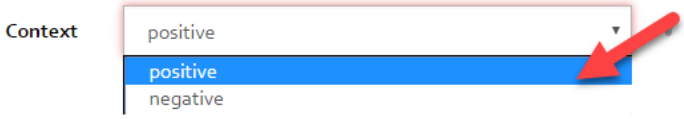
On clicking the dropdown you would be able to select from one of the options presented to you. The default is 'impact'.



To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.6 and Section 11.7.

15.15 CONTEXT

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

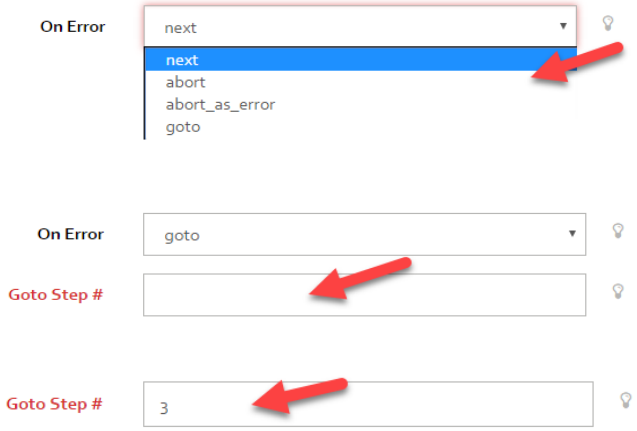


To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.8 and Section 11.9.

15.16 ON ERROR

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

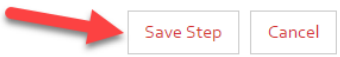
You can click on the dropdown to assign a different error handling for this step.



To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.10, Section 11.11, Section 11.12, and Section 11.13.

15.17 SAVE STEP

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



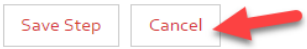
Once the step is saved, a summary of it would appear on that page. This would look something like this:

SETUP STEPS

#	Driver	Keyword	Description	Arguments	Execute	Run Mode	Impact	Context	onError	Insert	Edit	Delete
1	common_driver	get_time_delta			Type = Yes	Standard	impact	✚	abort			

15.18 CANCEL STEP

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



15.19 EDIT STEP

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

SETUP STEPS

#	Driver	Keyword	Description	Arguments	Execute	Run Mode	Impact	Context	onError	Insert	Edit	Delete
1	common_driver	get_time_delta			Type = Yes	Standard	impact	✚	abort			

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

SETUP STEPS

#	Driver	Keyword	Description	Arguments	Execute	Run Mode	Impact	Context	onError	Insert	Edit	Delete
1	common_driver	get_time_delta			Type = Yes	Standard	impact	✚	abort			

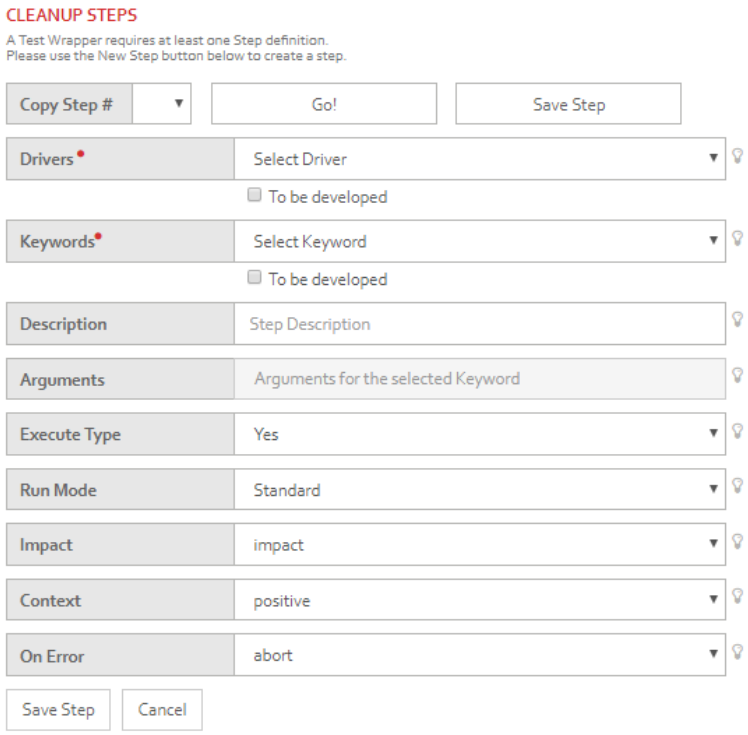
SETUP STEPS

#	Driver	Keyword	Description	Arguments	Execute	Run Mode	Impact	Context	onError	Insert	Edit	Delete
1	common_driver	get_time_delta			Type = Yes	Standard	impact	✚	abort			

15.20 CLEANUP STEPS

Cleanup steps are optional. A new cleanup step can be added to the wrapper file by clicking the "New Step" button.

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”.



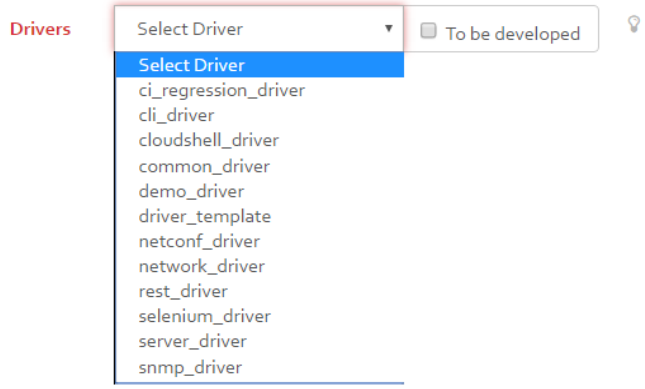
15.21 DRIVERS

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 Framework directory in the dropdown. You can select any driver out of those. You would not be able to select any

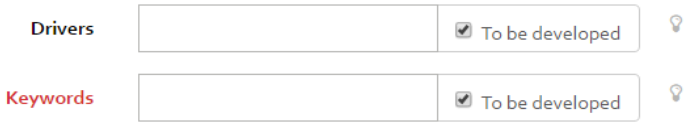
driver which lies outside of the ProductDrivers directory. This is because Warrior enforces a strict directory structure for the Actions and ProductDrivers.

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



You can then select any driver from that list:

If you want to include a Driver that has not been developed yet, you can simply check the “To be developed” box and add a driver name of your own.



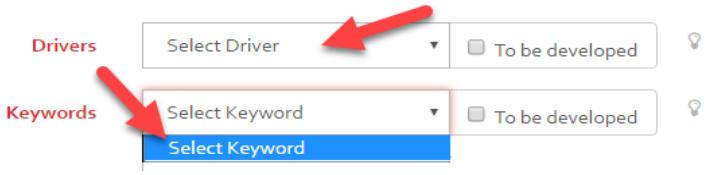
Be informed that checking the “To Be Developed” box for the driver automatically checks the “To Be Developed” box for the Keyword as a keyword for a non-developed driver cannot exist,.

You would then be able to include a driver name of your choice. You can also leave this field blank as checking the “To Be Developed” box renders the Drivers field as non-mandatory.

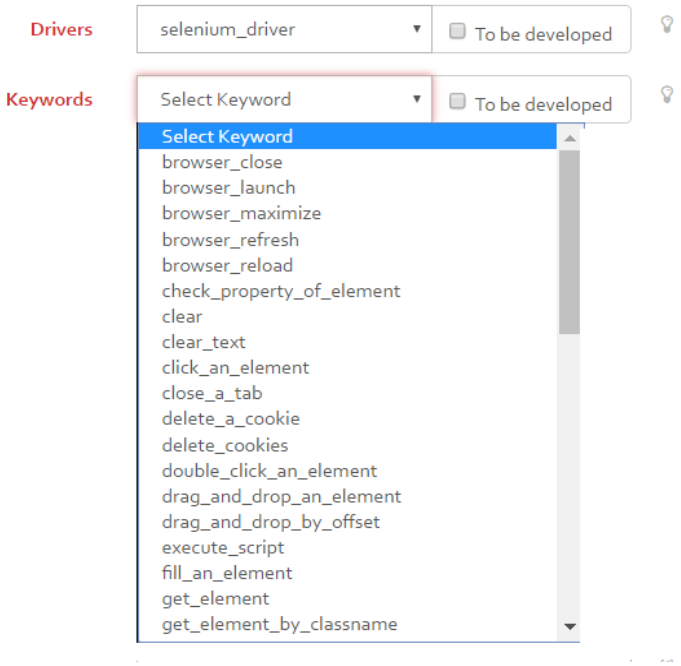
15.22 KEYWORDS

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.



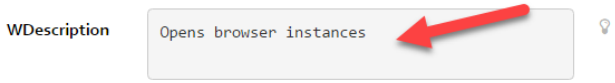
You can then select any keyword that you need.

If you want to include a Keyword that has not been developed yet, you can simply check the “To be developed” box and add a keyword name of your own.

A keyword name – even when you are including a non-developed keyword – is a mandatory field.

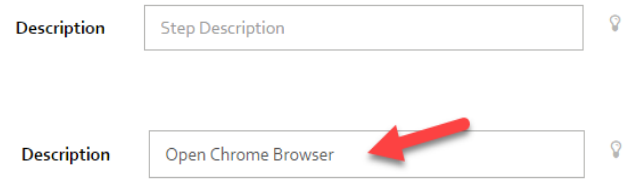
15.23 WDESCRIPTION

This field is populated when the Keyword is chosen. The description is for your reference, so that, you know exactly what this keyword does. If a non-developed keyword is chosen, this field would remain empty. This field is a non-editable field.



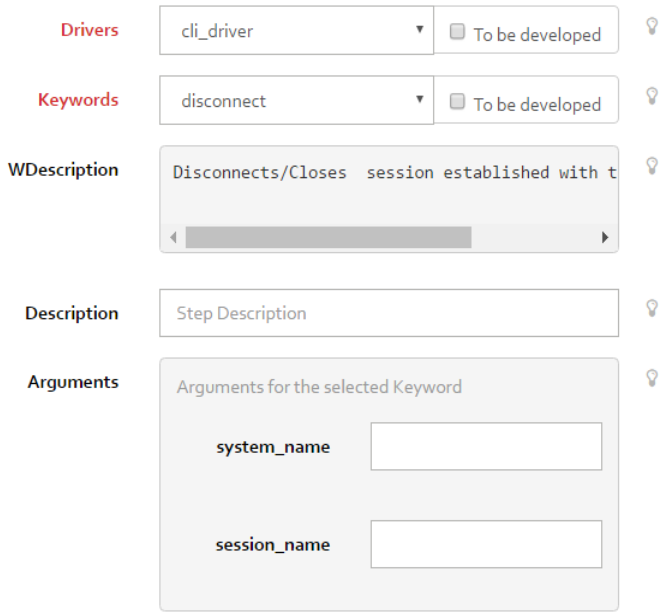
15.24 DESCRIPTION

This field is provided to you so that you can add a description of your own for the chosen Keyword. This description would be displayed when this steep is run. This is an optional field.

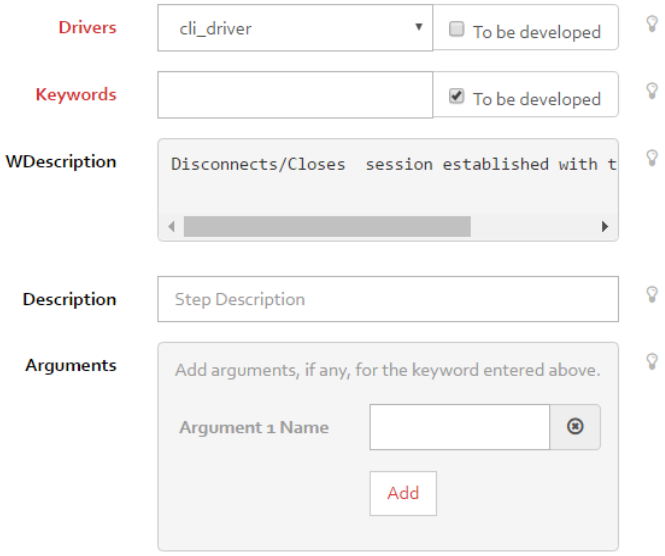


15.25 ARGUMENTS

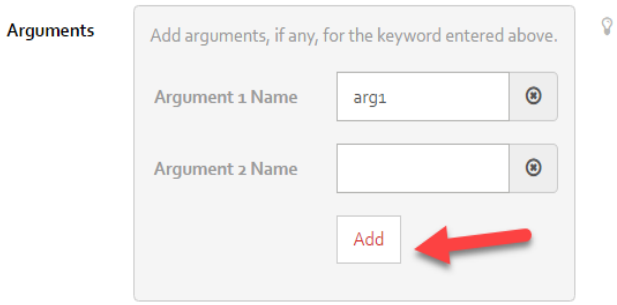
This space gets populated when a developed keyword is selected. All the arguments accepted by the Keyword show up here. You can type in the values associated with each value against that argument name. To identify which are the mandatory arguments, you can go through the next section.



When a non-developed keyword is selected, this field is populated a little differently.

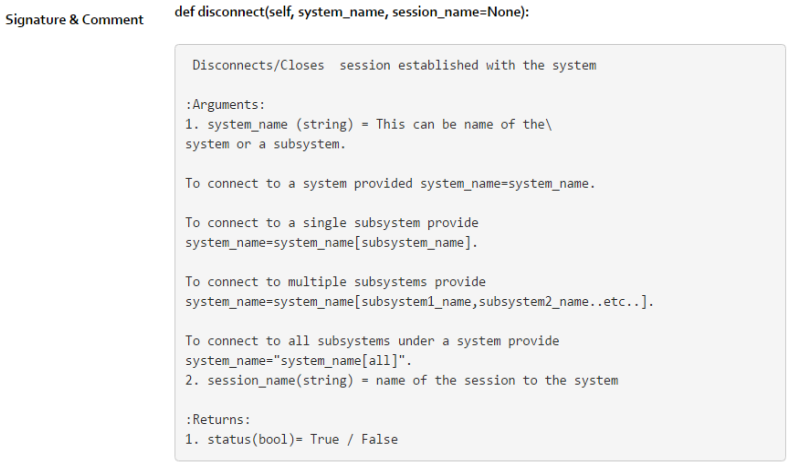


Here, in the space where Arguments for a developed keyword should show up, input fields that will let you define the arguments that you think this keyword should have would be displayed. You can then add the arguments names for the non-developed keyword.



15.26 SIGNATURE & COMMENTS

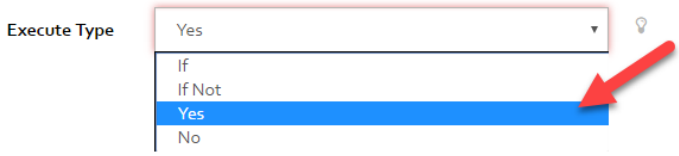
This field also appears prepopulated when a developed Keyword is selected. This is for your benefit, so that you understand what the keyword does when it runs, what kind of arguments it accepts, and what kind of values it returns.



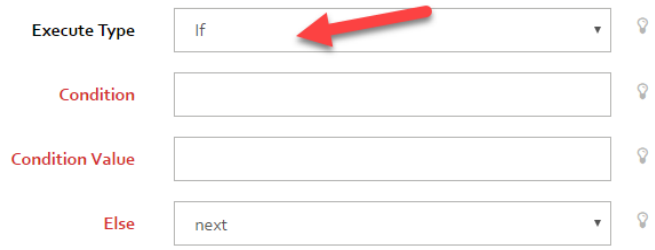
This field would remain blank when a non-developed keyword is selected.

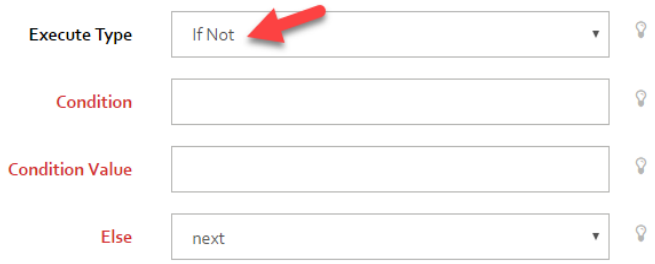
15.27 EXECUTE TYPE

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.

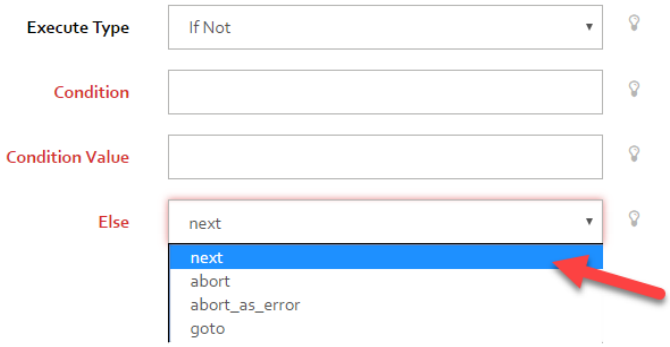


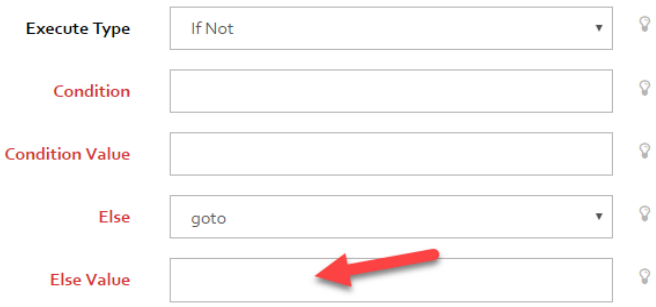
On clicking on 'If' and 'If Not', two other input boxes show up that let you add your conditions.





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.

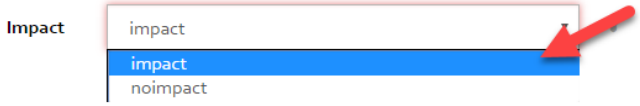




To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.1.

15.28 IMPACT ON FAILURE

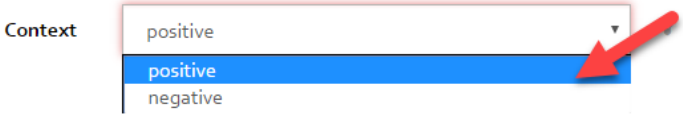
On clicking the dropdown you would be able to select from one of the options presented to you. The default is 'impact'.



To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.6 and Section 11.7.

15.29 CONTEXT

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

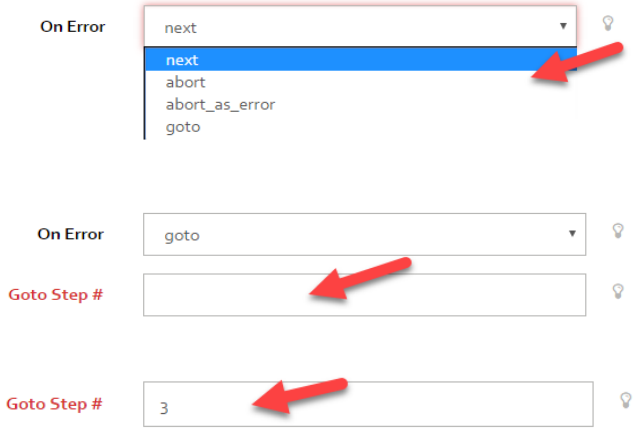


To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.8 and Section 11.9.

15.30 ON ERROR

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

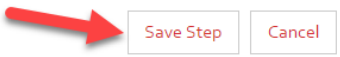
You can click on the dropdown to assign a different error handling for this step.



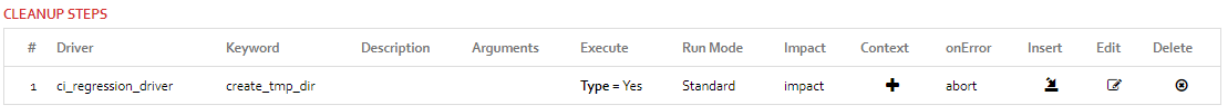
To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.10, Section 11.11, Section 11.12, and Section 11.13.

15.31 SAVE STEP

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

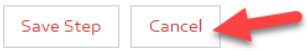


Once the step is saved, a summary of it would appear on that page. This would look something like this:



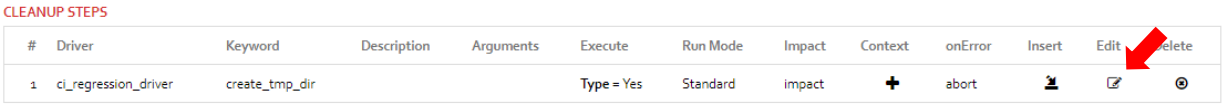
15.32 CANCEL STEP

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

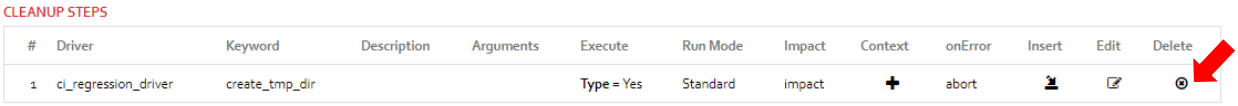


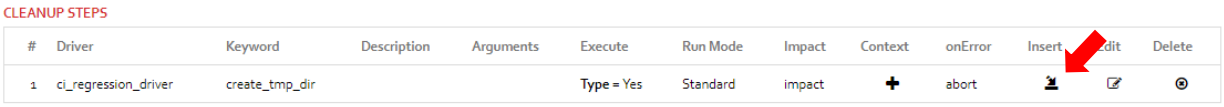
15.33 EDIT STEP

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



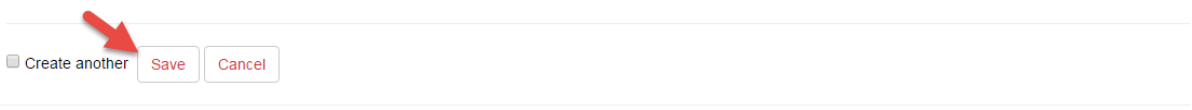
This opens up the step editor once again and you can make the desired changes.



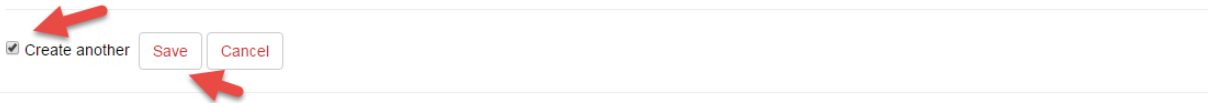


15.34 SAVE WRAPPER FILE

Clicking on 'Save' saves the entire wrapper file. If any unsaved step is present when this button is clicked, hat unsaved step will get saved.

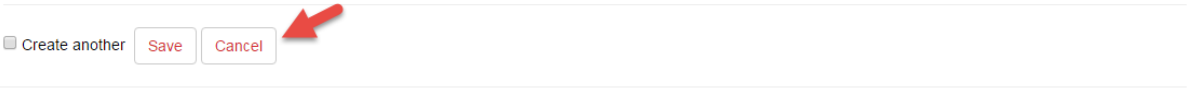


Checking the box against “Create Another” and then hitting Save, lets you save the case and directly opens up a page where you can start creating a new wrapper file.

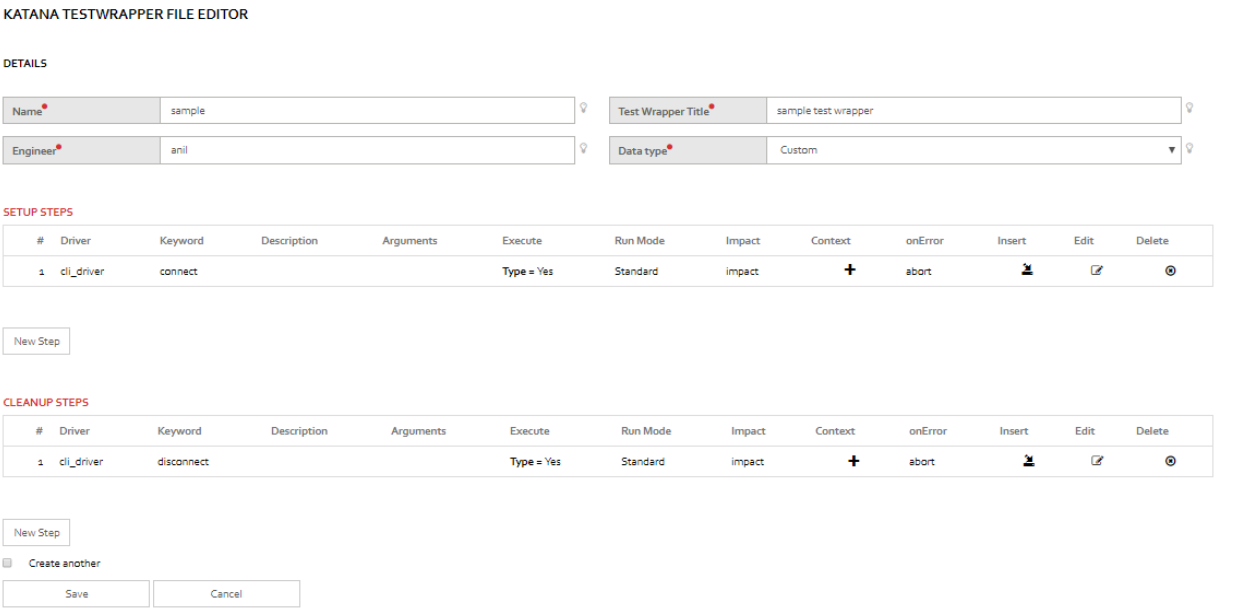


15.35 CANCEL WRAPPER FILE

Clicking on ‘Cancel’ discards all the changes performed.



The wrapper file thus created would look something like this:



16.0 DESIGNING A CASE FOR COMMON SETUP AND CLEANUP

While designing a testcase keep all the setup and cleanup steps in wrapper file and keep only test steps in testcase file.

Test Wrapper file can be included in Test Case as mentioned below in <Details> section with **<TestWrapperFile> tag.**

```
<Testcase>
    <Details>
        <TestWrapperFile> </TestWrapperFile>
    </Details>
    <Requirements>
    </Requirements>
    <Steps>
    </Steps>
</Testcase>
```

- ✓ When Test Wrapper file is included in test case details section, test case execution follows the given order
    1. Setup steps (from test wrapper file <Setup> block)
    2. Testcase steps (steps mentioned in <Steps> block)
    3. Cleanup steps(from test wrapper file <Cleanup block)

17.0 DESIGNING A SUITE

17.1 DESIGNING CASES IN A SUITE

A suite has the ability to control how the cases contained in it get executed. The Cases in a suite can be run in the following ways:

17.1.1 IN SEQUENCE

Warrior Cases defined within a suite can be executed in sequence. Follow the steps below to run the case in sequence:

1. Create the case that you would like to run in sequence.
2. Create a suite or modify an existing one.
3. Set the type tag in the suite to “Sequential”.
4. In the case block of the Suite, add the case created in step 1
5. Execute the suite. The cases will run in sequence.

17.1.2 IN PARALLEL

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

1. Create the case that you would like to run in parallel.
2. Create a suite or modify an existing one.
3. Set the type tag in the suite to “Parallel”.
4. In the case block of the Suite, add the case created in step 1
5. Execute the suite. The cases will run in parallel.

17.1.3 ITERATIVELY AND IN SEQUENCE

Warrior Cases defined within a suite can be executed on each system given in a data file iteratively. That is, if a Suite contains 2 Cases and the Data File contains 3 systems, with the “iterative_sequential” functionality, the Cases would get executed as follows:

1. Case 1 would get executed on system 1
2. Case 2 would get executed on system 1
3. Case 1 would get executed on system 2
4. Case 2 would get executed on system 2
5. Case 1 would get executed on system 3
6. Case 2 would get executed on system 3

Follow the steps below to run the case in iterative_sequentially:

1. Create the cases that you would like to run in iterative_sequentially.
2. Create a suite or modify an existing one.
3. Set the type tag in the suite to iterative_sequential.
4. Add the Input Data File in the Input Data File field in the details section of the Suite. This Input Data File will be used to run the Cases iterative_sequentially.
5. In the case block of the Suite, add the cases created in step 1
6. Execute the suite. The cases will run in iterative_sequentially.

17.1.4 ITERATIVELY AND IN PARALLEL

Warrior Cases defined within a suite can be executed on each system given in a data file iteratively and also in parallel. That is, if a Suite contains 2 Cases and the Data File contains 3 systems, with the "iterative_parallel" functionality, the Cases would get executed as follows:

1. Case 1 would get executed on system 1, system 2, and system 3 simultaneously
2. Case 2 would get executed on system 1, system 2, and system 3 simultaneously

Follow the steps below to run the case in iterative_sequentially:

1. Create the cases that you would like to run in iterative_parallel.
2. Create a suite or modify an existing one.
3. Set the type tag in the suite to iterative_parallel.
4. Add the Input Data File in the Input Data File field in the details section of the Suite. This Input Data File will be used to run the Cases iterative_sequentially.

5. In the case block of the Suite, add the cases created in step 1
6. Execute the suite. The cases will run in iterative_sequentially.

17.1.5 MULTIPLE TIMES

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

1. Create a suite or modify an existing one.

2. Set the type tag in the suite to "Run_Multiple" and then type in the value for maximum number of attempts.

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

17.1.6 RUN UNTIL FAILURE

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

1. Create a suite or modify an existing one.

2. Set the type tag in the suite to Run_Until_Failure and then type in the value for maximum number of attempts.

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

17.1.7 RUN UNTIL PASS

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

4. Create a suite or modify an existing one.

5. Set the type tag in the suite to Run_Until_Pass and then type in the value for maximum number of attempts.

6. Execute the suite. The cases in the suite will run until the suite passes or the suite is executed as indicated in the Max_Attempts attribute.

17.2 DESIGNING A CONDITIONAL SUITE

17.2.1 EXECUTE TYPE: YES

This is pretty straightforward; when the execute type for a Suite is set to “yes”, the suite will get executed no matter what.

Follow the steps below set the Execute Type of a Suite to “Yes”:

1. Create a Project to contain the Suites to be executed.
2. After selecting the suite path, set its Execute Type to Yes.

17.2.2 EXECUTE TYPE: NO

When the execute type for a Suite is set to “no”, the suite would not get executed under any circumstance. This option exists because when running a Project, you may not want to run some Suites every time you run that Project.

Follow the steps below set the Execute Type of a Suite to “No”:

1. Create a Project to contain the Suites to be executed.
2. After selecting the Suite path, set its Execute Type to No.

17.2.3 EXECUTE TYPE: IF

This is the programming equivalent of an “If” statement in Warrior– *If a certain condition is met, execute this Suite, otherwise skip this Suite and then* either *proceed to the next Suite* or *proceed to a specified Suite* or *abort execution of the Project*. Once the execute type has been set to “If”, you would need a condition to go along with it. This condition can has to be the result of a previously executed Suite. This previously executed Suite has to be inside the current Project and cannot be from a different Project.

To execute this Suite based on the result of a previously executed Suite, the condition would be test*suite_<suite-number>_status* and the value for this condition can be *PASS*, *FAIL*, *ERROR*, or *SKIPPED* depending upon when you want to execute this Suite.

An “else” for this “if” is also available in case the condition is not met. The value set in this else will let Warrior know what to do in case the condition is not met. You have the option of continuing the execution by executing the next Suite, or executing a particular Suite, or aborting the execution of the Project.

Follow the steps below set the Execute Type of a Suite to "If":

1. Create a Project to contain the Suites to be executed.
2. After selecting the Suite path, set its Execute Type to If.
3. Fields for Condition, Condition Value, and Else would appear in Katana. Type the condition in the format *testsuite_<suite-number>_status*, the Condition Value as either *PASS*, *FAIL*, *ERROR*, or *SKIPPED*, and then set the Else.

17.2.4 EXECUTE TYPE: IF NOT

This is the programming equivalent of an "! if" statement in Warrior – *If a certain condition is NOT met, execute this Suite, otherwise skip this Suite and then* either *proceed to the next Suite* or *proceed to a specified Suite* or *abort execution of the Project*. Once the execute type has been set to "If Not", you would need a condition to go along with it. This condition has to be the result of a previously executed Suite. This previously executed Suite has to be inside the current Project and cannot be from a different Project.

To execute this Suite based on the result of a previously executed Suite, the condition would be *testsuite_<suite-number>_status* and the value for this condition can be *PASS*, *FAIL*, *ERROR*, or *SKIPPED* depending upon when you want to execute this Suite.

An "else" for this "if" is also available in case the condition is met. The value set in this else will let Warrior know what to do in case the condition is met. You have the option of continuing the execution by executing the next Suite, or executing a particular Suite, or aborting the execution of the Project.

Follow the steps below set the Execute Type of a Suite to "If Not":

1. Create a Project to contain the Suites to be executed.
2. After selecting the Suite path, set its Execute Type to If Not.
3. Fields for Condition, Condition Value, and Else would appear in Katana. Type the condition in the format *testsuite_<suite-number>_status*, the Condition Value as either *PASS*, *FAIL*, *ERROR*, or *SKIPPED*, and then set the Else.

18.0 DESIGNING A SUITE FOR COMMON SETUP AND CLEANUP

Test Wrapper file can be included in Test Suite as mentioned below in <Details> section with <TestWrapperFile> tag.

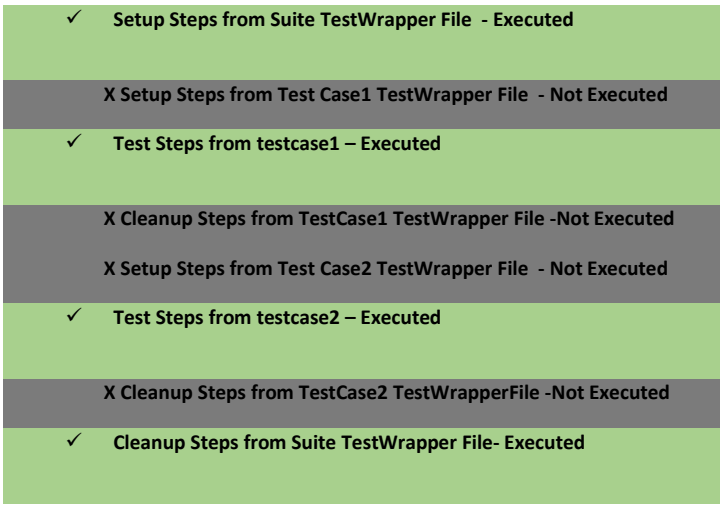
```
<TestSuite>
    <Details>
```

```
        <TestWrapperFile> </TestWrapperFile>
       <InputDataFile> </InputDataFile>
    </Details>
    <Requirements>
    </Requirements>
    <Testcases>
    </Testcases>
</TestSuite>
```

- ✓ When test wrapper file is included in test suite details section, test suite execution follows the given order:
    1. Setup steps (from wrapper file <Setup> block)
    2. All Testcases in testsuite (testcases mentioned in <Testcases> block)
       While executing these testcases only test steps will be executed which are in <Steps> block (test case level testwrapper file's setup and cleanup steps will not be executed)
       All these testcases will be executed as specified in exectype of test suite (parallel/sequential/RMT/RUP/RUF)
    3. Cleanup steps(from wrapper file <Cleanup block)

Refer below scenarios to understand execution order with or without TestWrapper File in suite and test case global section

- ✓ **Suite TestWrapper File – Yes, Test Case1 TestWrapper File – Yes, Test Case2 TestWrapper File - Yes**



- ✓ **Suite TestWrapper File – No , Test Case1 TestWrapper File – Yes, Test Case2 TestWrapper File – Yes**

X Setup Steps from Suite TestWrapper File -Not Present

✓ **Setup Steps from Test Case1 TestWrapper File - Executed**

✓ **Test Steps from testcase1 – Executed**

✓ **Cleanup Steps from TestCase1 TestWrapper File -Not Present**

✓ **Setup Steps from Test Case2 TestWrapper File - Executed**

✓ **Test Steps from testcase2 – Executed**

✓ **Cleanup Steps from TestCase2 TestWrapper File -Not Present**

X - Cleanup Steps from Suite TestWrapper File- Not Present

Exec type RMT, RUP, RUF behavior:

1) Setup Steps and Cleanup Steps from Suite TestWrapper file in above given blocks will not be considered in RMT, RUP and RUF

X Setup Steps from Suite TestWrapper File - Not Considered for RMT/RUF/RUP
X Cleanup Steps from Suite TestWrapper File- Not Considered for RMT/RUF/RUP

2) All other blocks such as Setup Steps and Cleanup Steps from Test TestWrapper file, Test steps will be considered for RMT/RUP/RUF.
3) TestWrapperFile is Optional, if not provided testcase or testsuite will be executed as per existing design
4) if one of the setup steps fail, execution goes to the cleanup steps without executing test steps in test case or test cases in suite. Test case or test suite result is marked as ERROR
5) test case or test suite is marked as PASS only if all the setup steps, test steps or cases and cleanup steps are passed.
6) All other attributes in step like <runmode>,<context>,<Execute>,<impact> are supported as it is.
7) All TestWrapper files should be placed in directory warrior_tests/wrapper_files with name starts with tw_xxxxx.xml

8) Results Rollup:

Section	Result	Overall Result
Setup	Pass	PASS
Test Steps or Test cases	Pass	
Cleanup	Pass	

Section	Result	Overall Result
Setup	Fail	ERROR
Test Steps or Test cases	Skip	
Cleanup	Pass/ Fail	

Section	Result	Overall Result
Setup	Pass	FAIL
Test Steps or Test cases	Fail	
Cleanup	Pass/ Fail	

Section	Result	Overall Result
Setup	Pass	WARN
Test Steps or Test cases	Pass	
Cleanup	Fail	

19.0 ERROR HANDLING CONTROL IN WARRIOR

Warrior allows error handling to be handled at the keyword, case, suite, 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, case or suite, abort execution or go to another case, step or suite. If the error handling instruction is not present, Warrior will default to proceed to next step/case/suite as applicable.

1. Case Error Handling: User can set a default error handling instruction for the entire 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 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. Suite Error Handling: User can set a default error handling instruction for the entire 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 case.

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

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

c. If 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 case in case of a 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 suite.

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

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

   c. If 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 suite in case of a suite failure.

20.0 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 for the Action classes.
    - Each keyword (along with its driver) is a step in the Warrior 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 FRAMEWORK KEYWORD TEMPLATE AND RULES """


""" Warrior Framework uses standard Python programming for its keyword development, however Warrior Framework 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 Framework, 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 Framework 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 Framework 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 Framework will not report
any                                                                    substep
        in the case result file"""

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



"""
SOME USEFUL Information:


Warrior Framework provides a data repository within a case and a suite. The
data repository within a 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 case. The process by which keywords can access the data returned
by other keywords is described in item #4 below.


Warrior Framework also provides a data repository within a suite. The data
repository within a suite contains all the data returned by the keywords of
the cases 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 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 case
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 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 case result file:

-------------------------------------------

Use case_Utils.pNote to add notes to the result xml file refer to case_Utils.py
for more functions related to case result file reporting


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




3. Get details from input datafile:

--------------------------------------

Warrior Framework 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 the Case or Suite
data repository as a key-value pair. Keywords will have to access the 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 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.case_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.case_Utils.pNote(system_name)
                  Utils.case_Utils.pNote(logfile)
                  Utils.case_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.case_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.case_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 case results.
    3. When a keyword returns a dictionary the case-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
```

21.0 EXECUTING WARRIOR FRAMEWORK CASES, SUITES, AND PROJECTS

21.1 THROUGH CLI

To run Warrior through a Command Line Interface, you will need to be inside this directory:

```
warriorframework/warrior
```

If you want to execute Warrior from any directory, substitute the `Warrior` in the commands below with a path to the Warrior executable.

For example: If you are currently inside your home directory, that is, you are inside `/home/user`, and then your warriorframework directory is inside this directory, then you can simply run Warrior from `/home/user` and the commands below would become:

```
/home/user/warriorframework/warrior/Warrior /path/case.xml
```

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

21.1.1 RUN A CASE

```
Warrior /path/case.xml
```

21.1.2 RUN MULTIPLE CASES

```
Warrior /path/case1.xml /path/case2.xml
```

21.1.3 RUN A CASE MULTIPLE TIMES

```
Warrior -RMT <numeric value> /path/to/case1.xml /path/to/case2.xml
/path/to/case3.xml
```

21.1.4 RUN A CASE UNTIL FAILURE

```
Warrior -RUF <numeric value> /path/to/case1.xml /path/to/case2.xml
/path/to/case3.xml
```

21.1.5 RUN A CASE UNTIL IT PASSES

```
Warrior -RUP <numeric value> /path/to/case1.xml /path/to/case2.xml
/path/to/case3.xml
```

21.1.6 RUN A SUITE

```
Warrior /path/suitename.xml
```

21.1.7 RUN MULTIPLE SUITES

```
Warrior /path/suite1.xml /path/suite2.xml
```

21.1.8 RUN A PROJECT

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

21.1.9 RUN MULTIPLE PROJECTS

```
Warrior /path/projectname1.xml /path/projectname2.xml
```

21.1.10 RUN A COMBINATION OF CASE, SUITE, AND PROJECT

```
Warrior /path/case.xml /path/suite1.xml /path/projectname.xml
```

21.1.11 SCHEDULE EXECUTION

```
Warrior -scheduletime datetime /path/to/case1.xml /path/to/case2.xml
/path/to/case3.xml
```

21.1.12 RUN KEYWORDS IN A CASE IN PARALLEL; CASES IN SEQUENCE

Here, using the command below, you can run all the cases in sequence (i.e., one after the other) and all the keywords in each Case would run in parallel (i.e., all at once)

```
Warrior -kwparallel -tcsequential /path/to/case1.xml /path/to/case2.xml
/path/to/case3.xml
```

21.1.13 RUN KEYWORDS IN A CASE IN SEQUENCE; CASES IN PARALLEL

Here, using the command below, you can run all the cases in parallel (i.e., all at once) and all the keywords in each Case would run in sequence (i.e., one after the other)

```
Warrior -tcparallel /path/to/case1.xml /path/to/case2.xml /path/to/case3.xml
```

21.1.14 EXECUTION BASED ON CATEGORY

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

Arguments:

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

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

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

`-suitelocn` (optional): path to location where the 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 suite_file_name -suitelocn location_to_create_suite_xml_file
```

Commands:

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

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

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

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

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

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

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

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

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

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

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

```
./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
```

21.1.15 JIRA BUG REPORTING

For JIRA bug reporting, configure `Warrior/Tools/Jira/jira_config.xml` file. A complete explanation as to how to do that can be found here.

Default JIRA project definition:

Warrior Framework 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
```

21.2 THROUGH KATANA

Warrior can also be executed through Katana. Please refer to Katana User Guide: Section 9 for all the details.

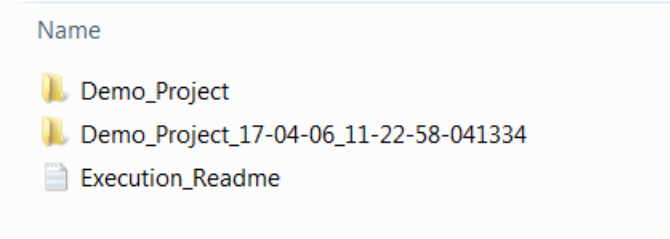
22.0 UNDERSTANDING THE LOGS & RESULTS

When a case, suite, or a project is run, it creates a log and result directory. Being able to understand this result directory is a big asset in debugging and understanding why the keywords and/or the cases, suites, and projects passed or failed.

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 case, suite, or project - if that is left empty, then in the default directory – the Execution folder in Warriorspace.

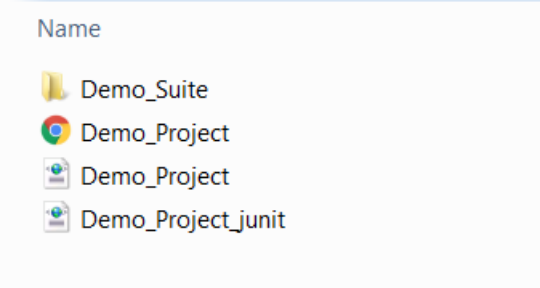
22.1 PROJECT EXECUTION STUCTURE

1. The results directory for a project would of the same name as that the project. If the results directory is the first one in the destination for that particular project, then the time stamp would not be appended to the directory name – like the first directory in the screenshot below:



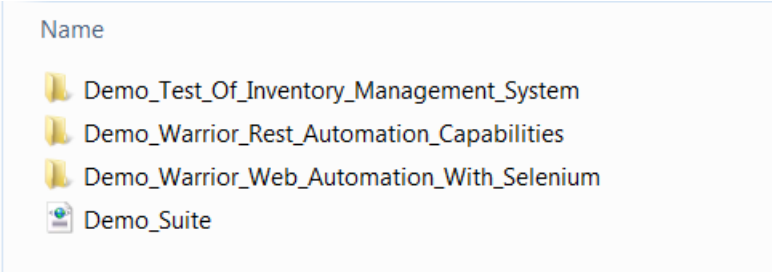
   If the same project is run the second time, then a date time stamp would be appended to the directory name, like the second directory in the above screenshot.

2. This project-execution directory will have directories for each of the suite contained in the project. These directories will have the same name as the Suites that were run. Since the project run above has only one Suite – with the name “Demo_Suite” - in it, there would be only one subfolder – also with the name “Demo_Suite” – in this directory



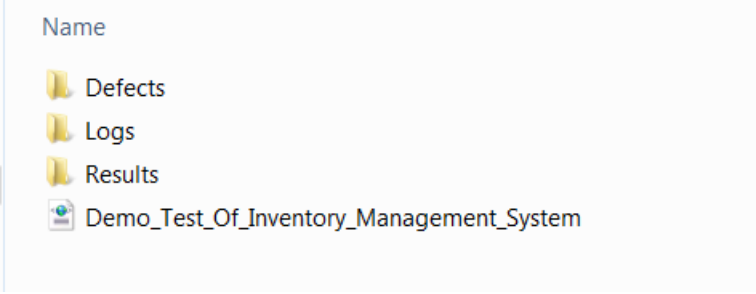
   There is also an HTML file (same name as the Project), a JUnit result file (same name as the Project and with “_junit” appended to it), and the Project file (this is the Project that was executed) inside this directory.

3. The Suite folder will have individual folders for all the Cases contained it. Since the Suite run above contains three Cases, we can see three subfolders with the name of the Cases created here:



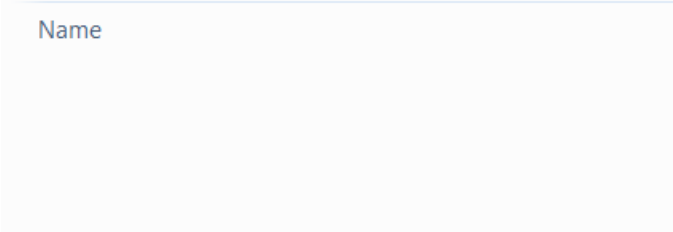
There is also a Suite file (this is the Suite that was executed) inside this directory.

4. When we go into one of the directories for the Case, we can see that there are three subfolders underneath in – Defects, Logs, and Results.

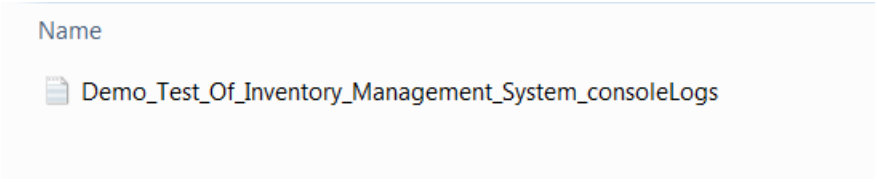


There is also a Case file (this is the Case that was executed) inside this directory.

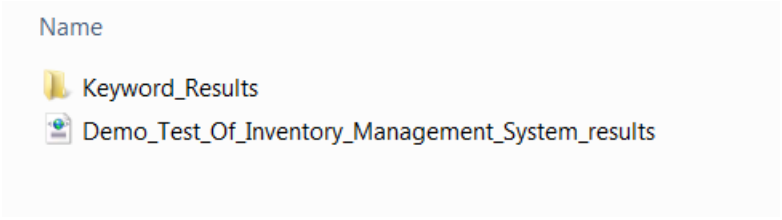
a. If the case passed with no defects, error, or exceptions (which happened in this case), then the defects folder will be empty.



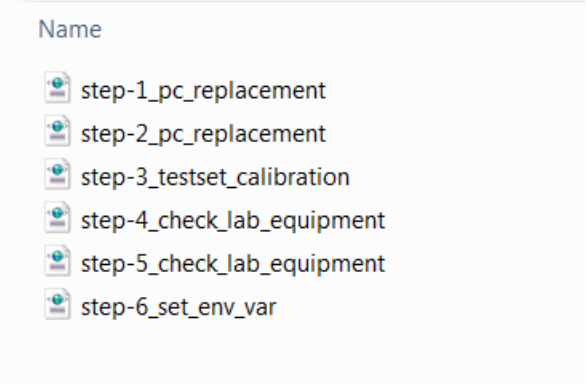
b. The Logs folder would contains logs from all the system that were used by the case and also the consoleLogs. The consoleLogs contain all the activity that went on, on the console when the case was running. These consoleLogs contain all the information you need to know what happened during the execution.



c. Then there is the Results directory which contains a subfolder "Keyword_Results" and a results file for the Case that ran. This results file contains information related to the execution of the Case in XML format and is used by the HTML file.

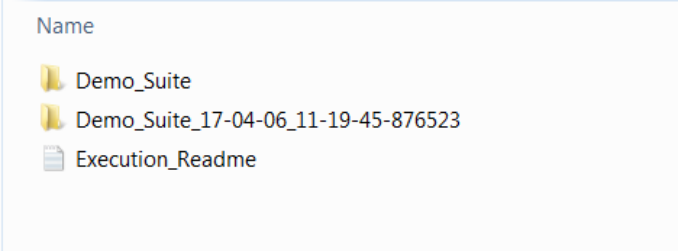


d. The Keyword_Results directory contains result files for each and every keyword that was executed in that Case. These result files have the same name as the keyword that was executed and they are also used by the HTML result file, although can open these files up and go through them if you want to.



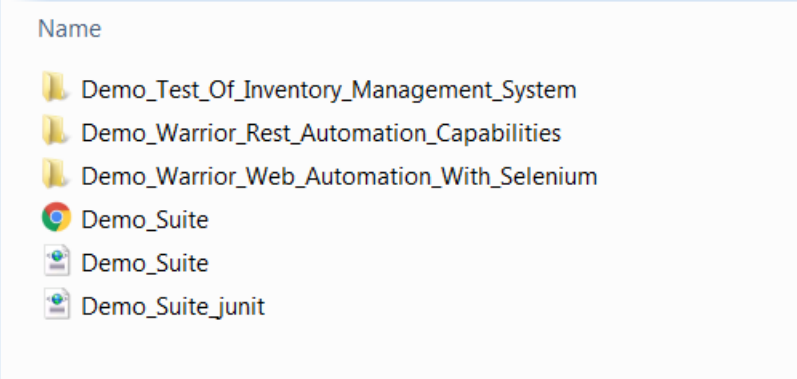
22.2 SUITE EXECUTION STUCTURE

1. The results directory for a suite would of the same name as that the suite. If the results directory is the first one in the destination for that particular suite, then the time stamp would not be appended to the directory name – like the first directory in the screenshot below:



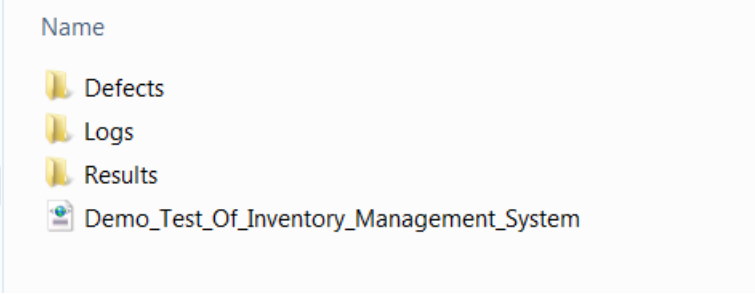
If the same suite is run the second time, then a date time stamp would be appended to the directory name, like the second directory in the above screenshot.

2. This suite-execution directory will have directories for each of the case contained in the suite. These directories will have the same name as the Cases that were run.



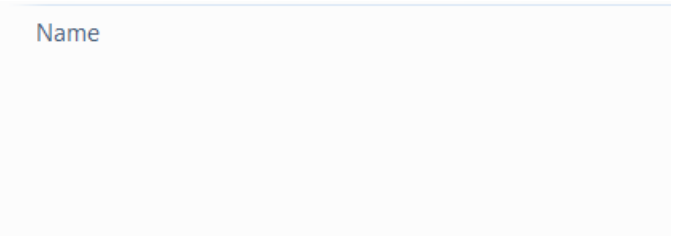
There is also an HTML file (same name as the Suite), a JUnit result file (same name as the Suite and with "_junit" appended to it), and the Suite file (this is the Suite that was executed) inside this directory.

3. When we go into one of the directories for the Case, we can see that there are three subfolders underneath in – Defects, Logs, and Results.



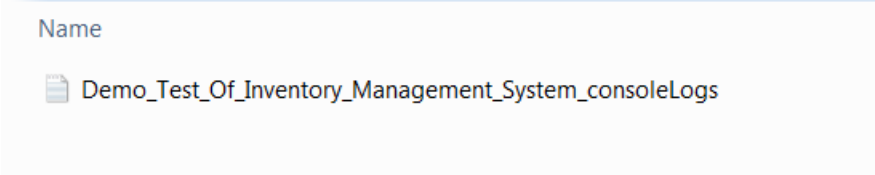
There is also a Case file (this is the Case that was executed) inside this directory.

   a. If the case passed with no defects, error, or exceptions (which happened in this case), then the defects folder will be empty.

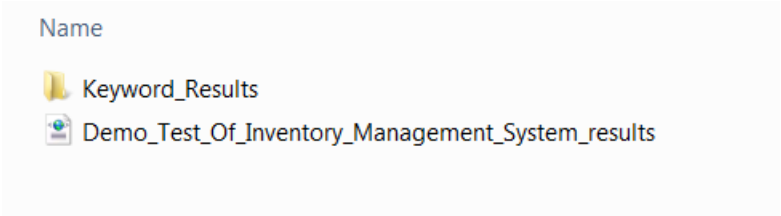


   b. The Logs folder would contains logs from all the system that were used by the case and also the consoleLogs. The consoleLogs contain all the activity that went on, on the

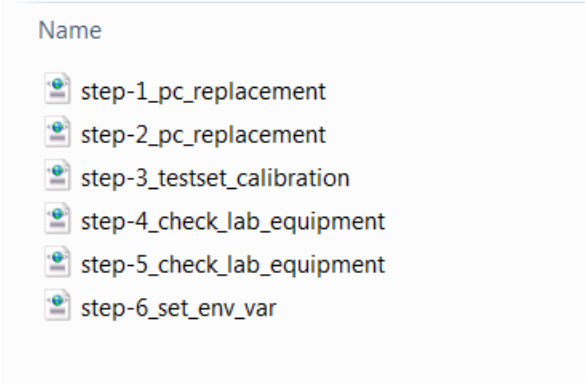
console when the case was running. These consoleLogs contain all the information you need to know what happened during the execution.



c. Then there is the Results directory which contains a subfolder "Keyword_Results" and a results file for the Case that ran. This results file contains information related to the execution of the Case in XML format and is used by the HTML file.

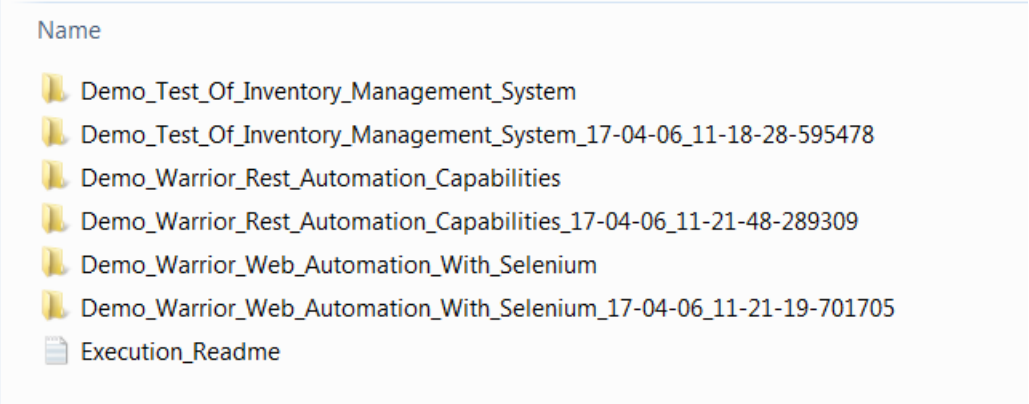


d. The Keyword_Results directory contains result files for each and every keyword that was executed in that Case. These result files have the same name as the keyword that was executed and they are also used by the HTML result file, although can open these files up and go through them if you want to.



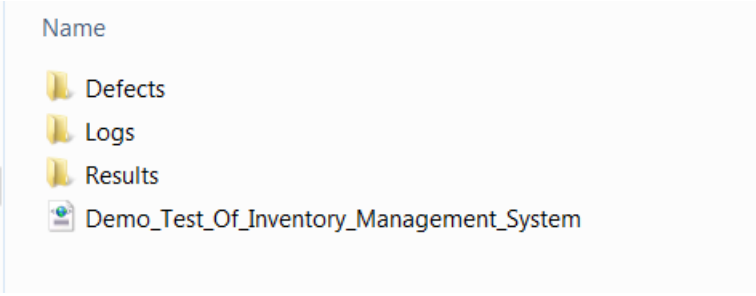
22.3 CASE EXECUTION STUCTURE

1. The results directory for a Case would of the same name as that the Case. If the results directory is the first one in the destination for that particular suite, then the time stamp would not be appended to the directory name – like the first, third, and fifth directories in the screenshot below:



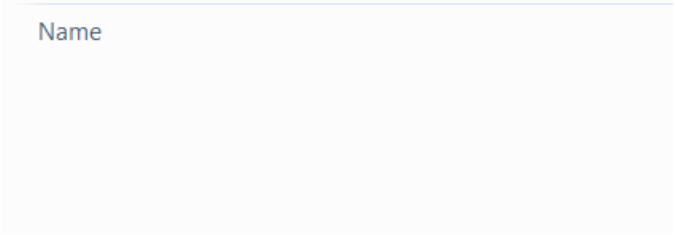
If the same case is run the second time, then a date time stamp would be appended to the directory name, like the second, fourth, and sixth directory in the above screenshot.

2. When we go into one of the directories for the Case, we can see that there are three subfolders underneath in – Defects, Logs, and Results.

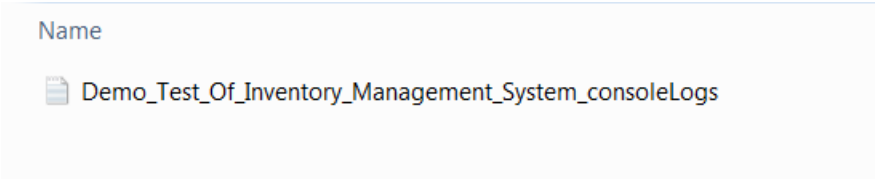


There is also an HTML file (same name as the Case), a JUnit result file (same name as the Case and with "_junit" appended to it), and the Case file (this is the Case that was executed) inside this directory.

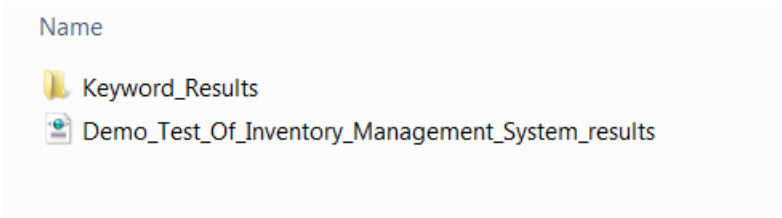
a. If the case passed with no defects, error, or exceptions (which happened in this case), then the defects folder will be empty.



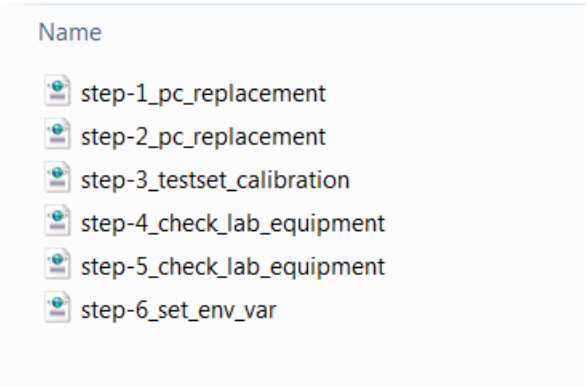
b. The Logs folder would contains logs from all the system that were used by the case and also the consoleLogs. The consoleLogs contain all the activity that went on, on the console when the case was running. These consoleLogs contain all the information you need to know what happened during the execution.



c. Then there is the Results directory which contains a subfolder "Keyword_Results" and a results file for the Case that ran. This results file contains information related to the execution of the Case in XML format and is used by the HTML file.



d. The Keyword_Results directory contains result files for each and every keyword that was executed in that Case. These result files have the same name as the keyword that was executed and they are also used by the HTML result file, although can open these files up and go through them if you want to.



22.4 THE HTML FILE

The HTML files available in each of the execution folders described above can opened in a browser. This HTML result file gives a visual appeal to the result files and make reading the result files easier. The HTML files different slightly for each execution level. A brief explanation for all three levels has been given below.

1. For a Project

This HTML file has the Project at the top, its timestamp, status, and duration

Here you can see, that just by glancing that the result file, you can determine which keywords, cases, and suites passed and which did not.

On the right, a brief summary of the execution is provided where the number of suites, cases, and keywords that passed, failed, threw an Error or an Exception is given.

Results Summary

Type	Name	Info	Timestamp	Duration	Status	Impact	On Error	Results (📈) Logs (🗐) Defects (🐞)	Summary
Project	Demo_Project		2017-04-06 11:22:17	29.0	PASS			📈	Ts Count: 1, Ts Passed: 1, Ts Failed: 0, Ts Errors: 0, Ts Exceptions: , Ts Skipped: 0
Suite	Demo_Suite		2017-04-06T11:22:17	29.0	PASS	Impact	Abort	📈	Tc Count: 3, Tc Passed: 3, Tc Failed: 0, Tc Errors: 0, Tc Exceptions: , Tc Skipped: 0
Case	Demo_Test_Of_Inventory_Management_System	Demo_Test_Of_Inventory_Management_System	2017-04-06 11:22:20	4.0	PASS	Impact	Next	📈 🗐	Kw Count: 6, Kw Passed: 5, Kw Failed: 0, Kw Errors: 0, Kw Exceptions: 0, Kw Skipped: 0
Keyword	pc_replacement	1. system_name = PCs[HP]	2017-04-06 11:22:24	0.0	PASS	Impact	Next	📈	
Keyword	pc_replacement	1. system_name = PCs[HP]	2017-04-06 11:22:24	0.0	PASS	Impact	Next	📈	
Keyword	testset_calibration	1. system_name = TestSets[TBERD]	2017-04-06 11:22:24	0.0	PASS	Impact	Next	📈	
Keyword	check_lab_equipment	1. system_name = PCs[all]	2017-04-06 11:22:24	0.0	PASS	No Impact	Next	📈	
Keyword	check_lab_equipment	1. system_name = TestSets[all]	2017-04-06 11:22:24	0.0	PASS	Impact	Next	📈	
Keyword	set_env_var	1. var_value = Expected_dict_value 2. var_key = dict_key	2017-04-06 11:22:24	0.0	None	Impact	Next	📈	
Case	Demo_Warrior_Rest_Automation_Capabilities	Demo_Warrior_Rest_Automation_Capabilities	2017-04-06 11:22:24	5.0	PASS	Impact	Next	📈 🗐	Kw Count: 7, Kw Passed: 7, Kw Failed: 0, Kw Errors: 0, Kw Exceptions: 0, Kw Skipped: 0
Keyword	perform_http_post	1. system_name = http_bin_1 2. cookies = cookie=cookie_name 3. url = http://httpbin.org/post 4. json = {"postId":1, "comments":"This is a new comment"} 5. expected_response = 200, 302, 404 6. timeout = 0.5, 0.75 7. allow_redirects = yes 8. data = userId=1;id=1;title=Changed Post;body=New Comment	2017-04-06 11:22:28	0.0	PASS	Impact	Next	📈	

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.

2. For a Suite

This HTML file has the Suite at the top, its timestamp, status, and duration

Here you can see, that just by glancing that the result file, you can determine which keywords and cases passed and which did not.

On the right, a brief summary of the execution is provided where the number of cases and keywords that passed, failed, threw an Error or an Exception is given.

Results Summary

Type	Name	Info	Timestamp	Duration	Status	Impact	On Error	Results (📈) Logs (🗐) Defects (🐞)	Summary
Suite	Demo_Suite		2017-04-06T11:20:37	33.0	PASS			📈	Tc Count: 3, Tc Passed: 3, Tc Failed: 0, Tc Errors: 0, Tc Exceptions: , Tc Skipped: 0
Case	Demo_Test_Of_Inventory_Management_System	Demo_Test_Of_Inventory_Management_System	2017-04-06 11:20:40	4.0	PASS	Impact	Next	📈 🗐	Kw Count: 6, Kw Passed: 5, Kw Failed: 0, Kw Errors: 0, Kw Exceptions: 0, Kw Skipped: 0
Keyword	pc_replacement	1. system_name = PCs[HP]	2017-04-06 11:20:44	0.0	PASS	Impact	Next	📈	
Keyword	pc_replacement	1. system_name = PCs[HP]	2017-04-06 11:20:44	0.0	PASS	Impact	Next	📈	
Keyword	testset_calibration	1. system_name = TestSets[TBERD]	2017-04-06 11:20:44	0.0	PASS	Impact	Next	📈	
Keyword	check_lab_equipment	1. system_name = PCs[all]	2017-04-06 11:20:44	0.0	PASS	No Impact	Next	📈	
Keyword	check_lab_equipment	1. system_name = TestSets[all]	2017-04-06 11:20:44	0.0	PASS	Impact	Next	📈	
Keyword	set_env_var	1. var_value = Expected_dict_value 2. var_key = dict_key	2017-04-06 11:20:44	0.0	None	Impact	Next	📈	
Case	Demo_Warrior_Rest_Automation_Capabilities	Demo_Warrior_Rest_Automation_Capabilities	2017-04-06 11:20:44	6.0	PASS	Impact	Next	📈 🗐	Kw Count: 7, Kw Passed: 7, Kw Failed: 0, Kw Errors: 0, Kw Exceptions: 0, Kw Skipped: 0
Keyword	perform_http_post	1. system_name = http_bin_1 2. cookies = cookie=cookie_name 3. url = http://httpbin.org/post 4. json = {"postId":1, "comments":"This is a new comment"} 5. expected_response = 200, 302, 404 6. timeout = 0.5, 0.75 7. allow_redirects = yes 8. data = userId=1;id=1;title=Changed Post;body=New Comment	2017-04-06 11:20:48	1.0	PASS	Impact	Next	📈	
Keyword	perform_http_get	1. system_name = http_bin_3 2. timeout = 5	2017-04-06 11:20:49	0.0	PASS	Impact	Next	📈	

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.

3. For a Case

This HTML file has the Case at the top, its timestamp, status, and duration

Here you can see, that just by glancing that the result file, you can determine which keywords passed and which did not.

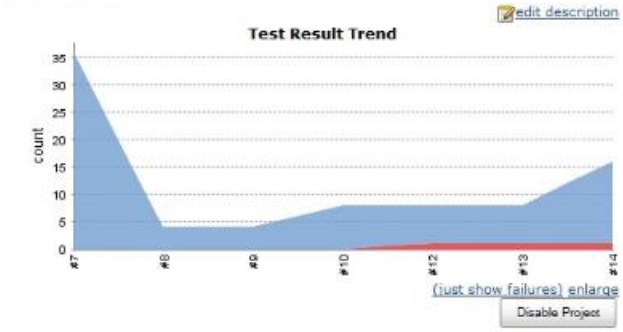
On the right, a brief summary of the execution is provided where the number of keywords that passed, failed, threw an Error or an Exception is given.

Results Summary

Type	Name	Info	Timestamp	Duration	Status	Impact	On Error	Results (📈) Logs (📄) Defects (🐞)	Summary					
Case	Demo_Test_Of_Inventory_Management_System	Demo_Test_Of_Inventory_Management_System	2017-04-06 11:17:15	4.0	PASS			📈 📄	Kw Count	Kw Passed	Kw Failed	Kw Errors	Kw Exceptions	Kw Skipped
									6	5	0	0	0	0
Keyword	pc_replacement	1. system_name = PCs[HP]	2017-04-06 11:17:19	0.0	PASS	Impact	Next	📈						
Keyword	pc_replacement	1. system_name = PCs[HP]	2017-04-06 11:17:19	0.0	PASS	Impact	Next	📈						
Keyword	testset_calibration	1. system_name = TestSets[TBERD]	2017-04-06 11:17:19	0.0	PASS	Impact	Next	📈						
Keyword	check_lab_equipment	1. system_name = PCs[all]	2017-04-06 11:17:19	0.0	PASS	No Impact	Next	📈						
Keyword	check_lab_equipment	1. system_name = TestSets[all]	2017-04-06 11:17:19	0.0	PASS	Impact	Next	📈						
Keyword	set_env_var	1. var_value = Expected_dict_value 2. var_key = dict_key	2017-04-06 11:17:19	0.0	None	Impact	Next	📈						

22.5 THE JUNIT 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 case executions to display the results graphically.



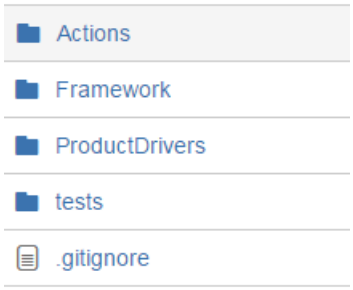
This JUnit file is in the correct JUnit format. This file provides a very high level view of the execution and its result.

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

23.0 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:



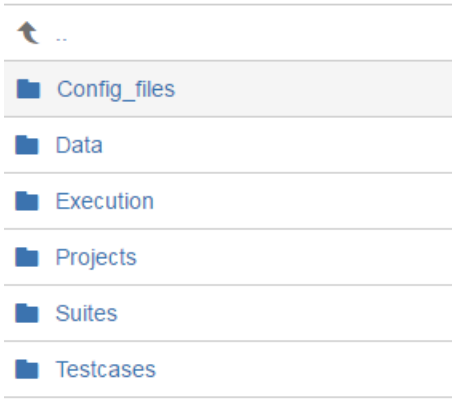
3. The Actions folder must contain all the keywords, the Framework folder must contain all the associated utils, and the ProductDrivers must contain all the product drivers.
4. The tests directory should be used only for uploading the unit tests related to this keyword repository. Do NOT upload Warrior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

24.0 GUIDELINES TO CREATE A WARRIORSPACEREPOSITORY

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 Cases should contain all the Case 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 Framework's web based case creation and execution tool. From Katana, users can create cases, suites, projects, input data files, testdata files, and warhorn configuration files. Cases, suites, projects, input data files, testdata files, and warhorn configuration 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 cases, suites, and projects from the execute tab.

1.0 WHERE IS KATANA LOCATED?

Katana is currently located at:

https://github.com/warriorframework/warriorframework

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.

1.2 HOW TO INSTALL KATANA?

Katana already comes with Warrior. If you do not have Warrior cloned on your system, you can find the instructions to clone Warrior here.

The Katana directory can be found right inside the root directory. You can go to the katana directory with this command:

```
cd warriorframework/katana
```

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 warriorframework/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:

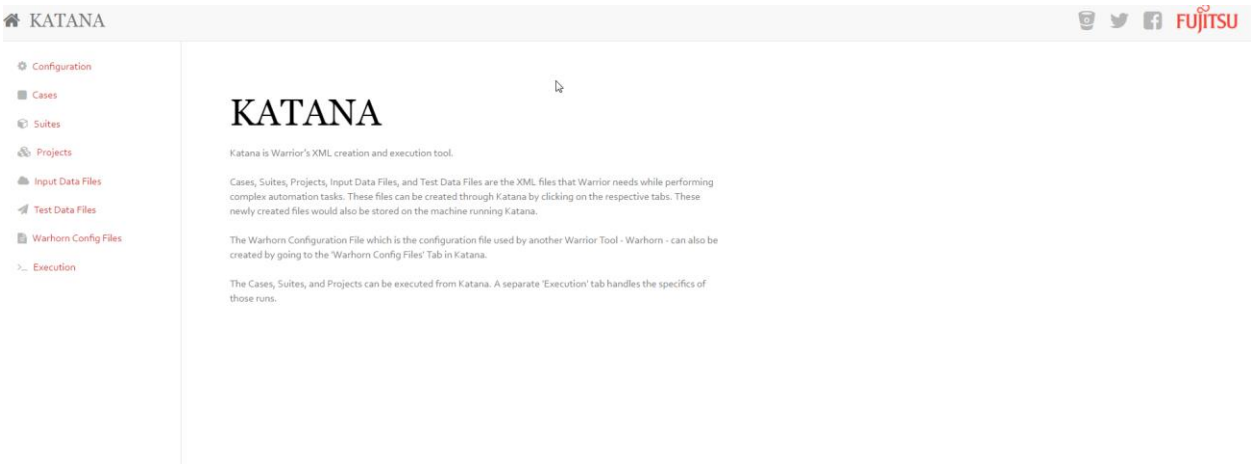
```
python katana.py –p 5001
```

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

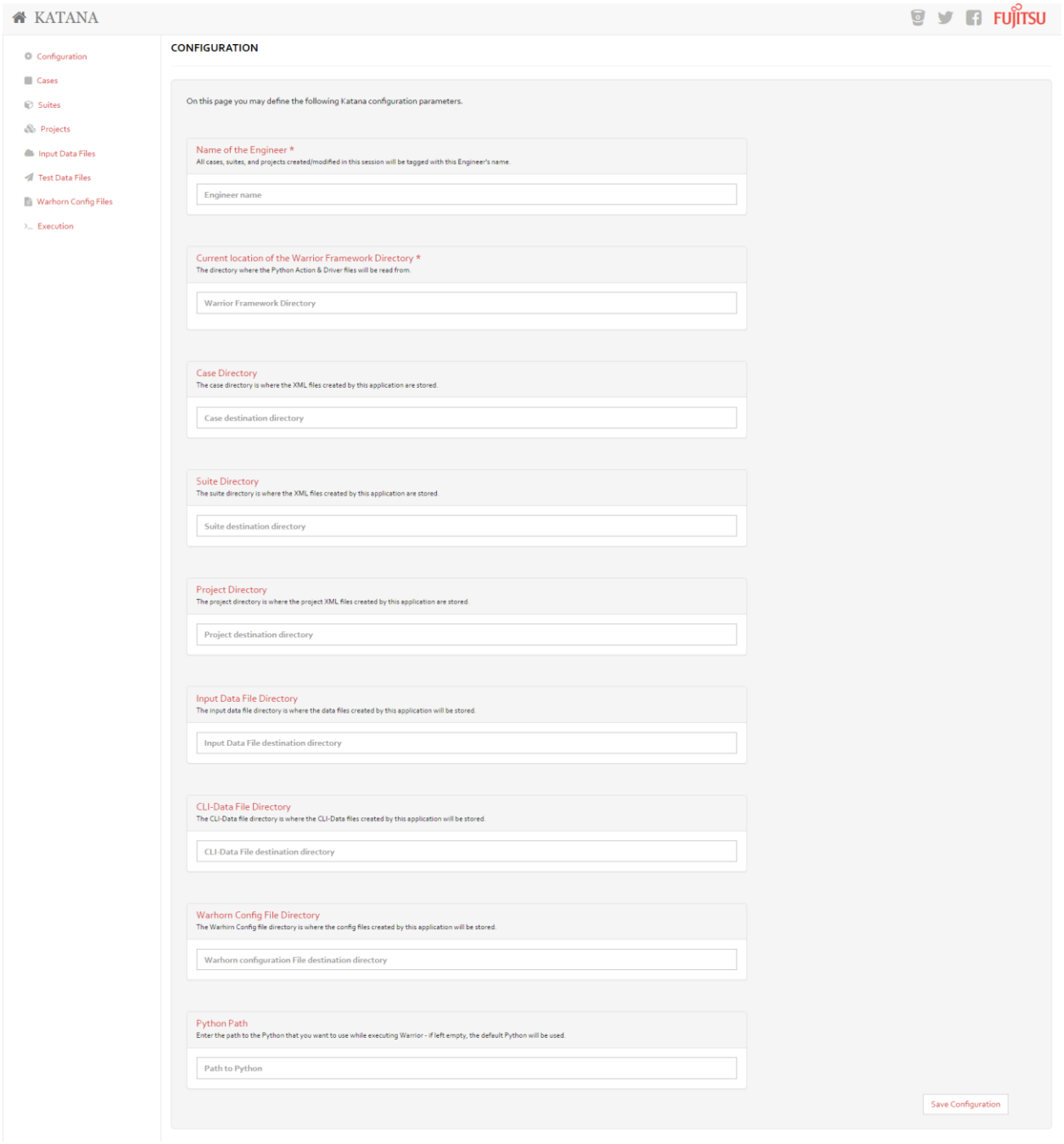
```
localhost:5001
```

2. GETTING STARTED WITH KATANA

Katana starts up on the localhost and this page shows up



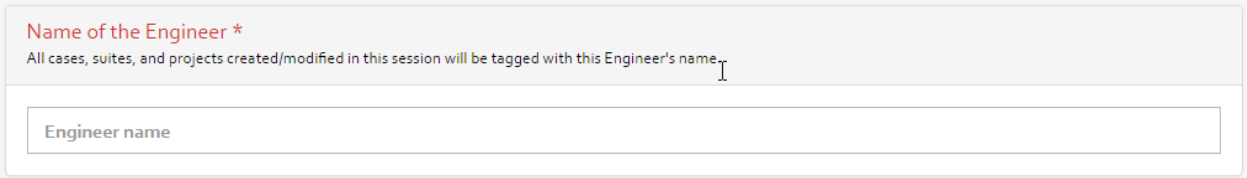
Katana has 7 tabs on the top in total. To configure Katana to your system, you will need to click on the second tab – the configuration tab – and this page would show up.



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.

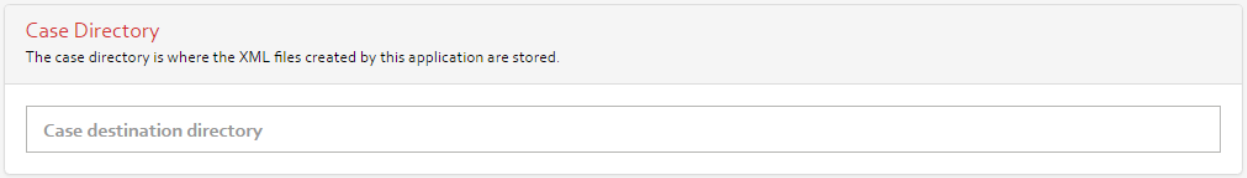


Current Location of the Warrior Framework Directory.

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

Case Directory

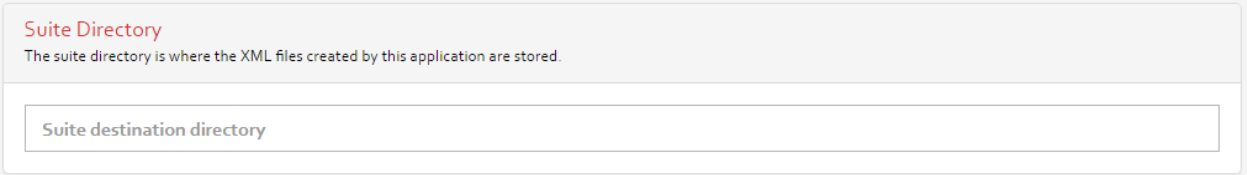
This field should contain the path to the Case directory in your machine. This directory may or may not be inside your Warrior Framework directory



To understand what Warrior Cases are, refer to Warrior Framework User Guide: Section 6.0. To create cases in Katana, refer to Katana User Guide: Section 3

Suite Directory

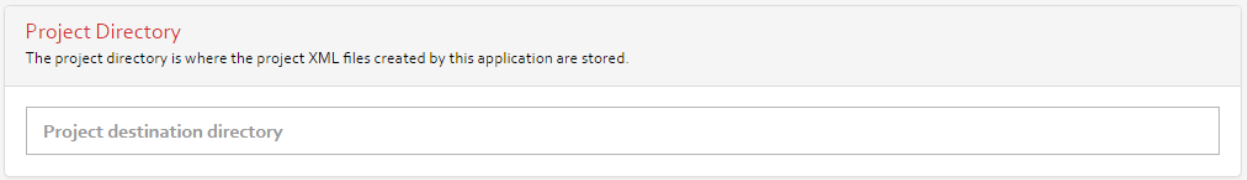
This field should contain the path to the Suite directory in your machine. This directory may or may not be inside your Warrior Framework directory



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

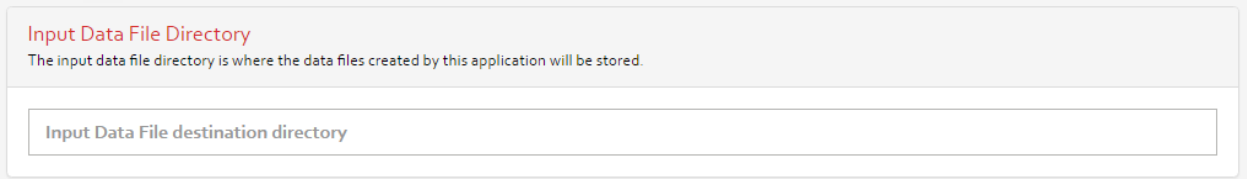
Project Directory

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



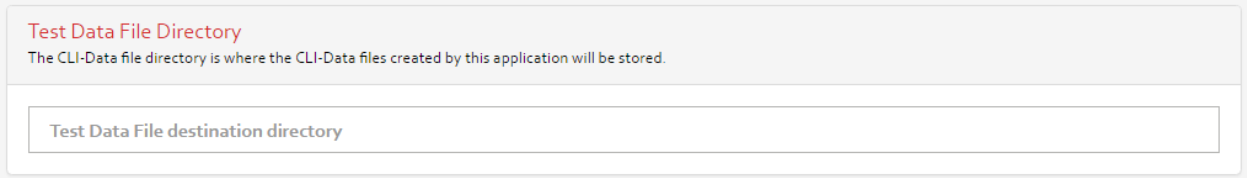
Input Data Directory

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



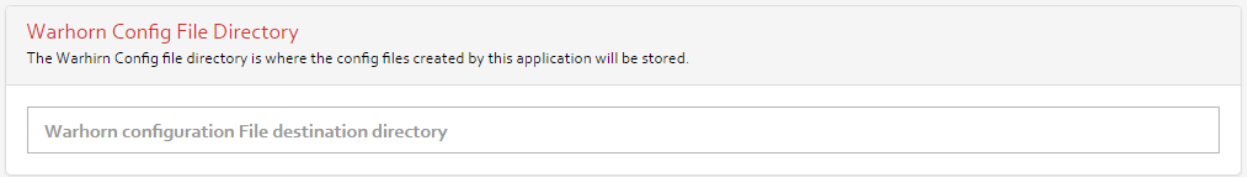
Test Data Directory

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



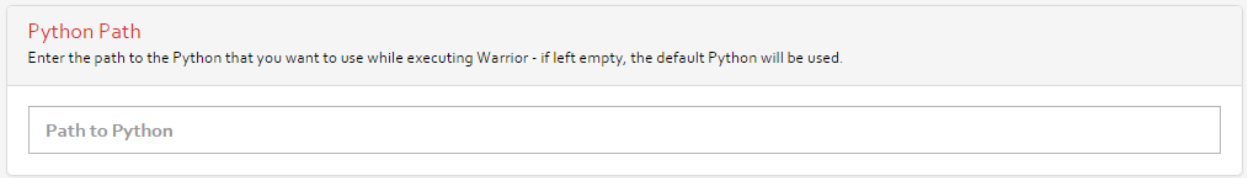
Warhorn Config File Directory

This field should contain the path to the Warhorn configuration files directory in your machine. This directory is typically inside the warhorn directory.

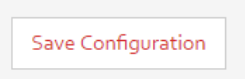


Python Path

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



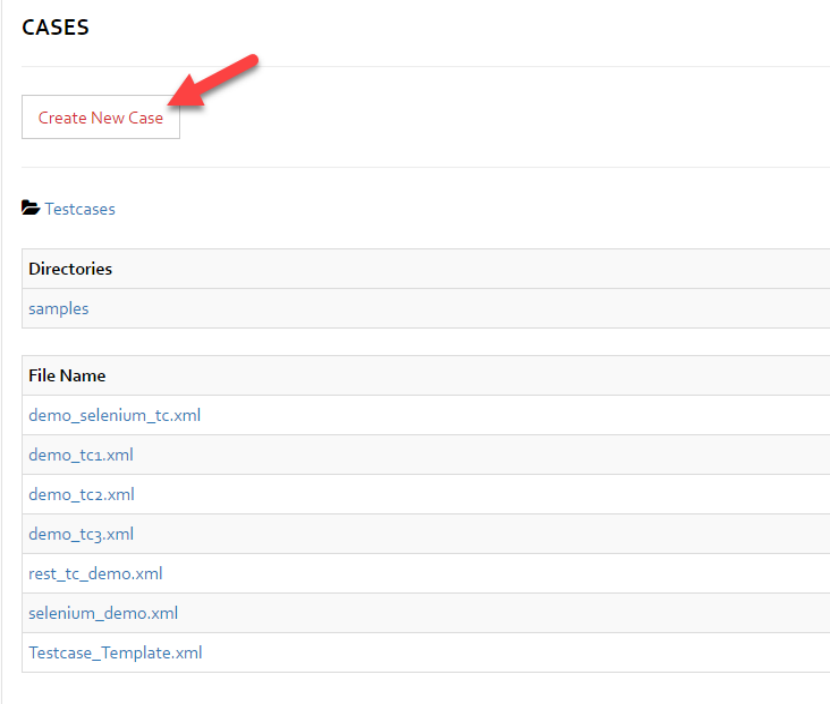
Finally, hit save to save the configuration.



Katana is now ready to be used.

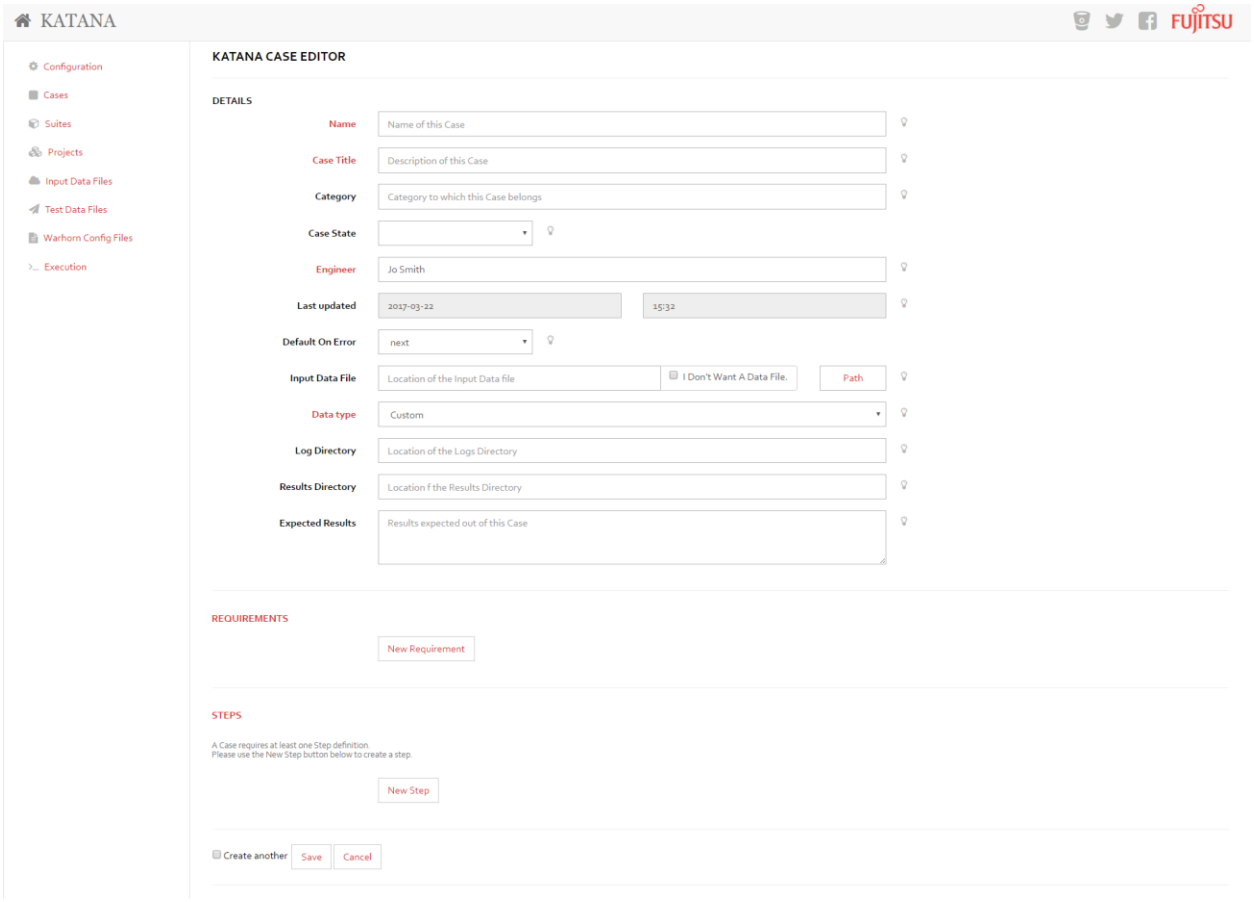
3. CREATING CASES WITH KATANA

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



Note: The list of cases below the “New Case” button may differ as Katana would show the Cases saved in your case directory.

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



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

3.1. CASE NAME

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

3.2. CASE TITLE

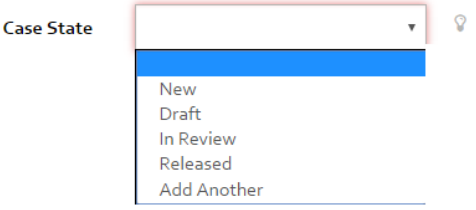
A case Title (or description) is a field where you can add a descriptive title to identify what this case does. This is a mandatory field.

3.3. CATEGORY

In this field, you can classify this case into a particular "category". Categorization of cases is important when you have to run all the cases in the same category together. Please refer Warrior Framework 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 cases.

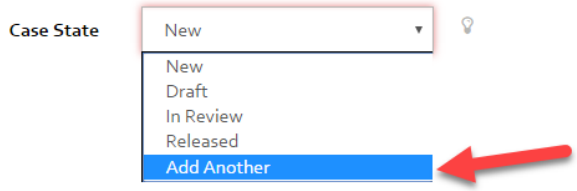
3.4. CASE STATE

A case state is informative when you have to keep track of which cases 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 case is in which state of development.

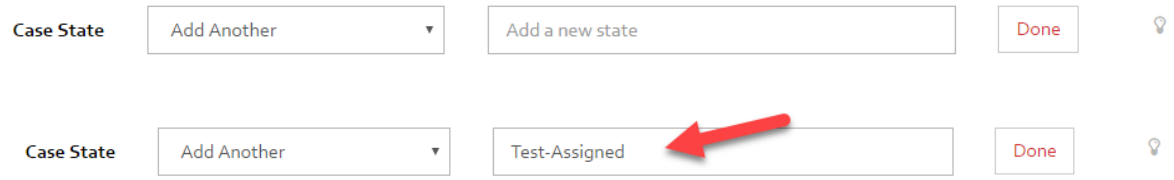


You can select any state of your choice, like "New". This would mark your case as "New" meaning that this is a newly created case.

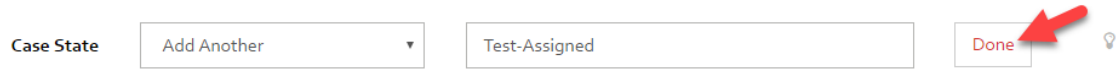
Furthermore, if the available categories do not really fit your case state, you can go ahead and select “Add Another”.



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



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



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

If you want to create a Case that you don’t want anyone to accidentally run, or a Case that should not be run until certain criteria are met, then the Case State should be set as “Draft”. Warrior does not execute a Case in the Draft State.

3.5. ENGINEER

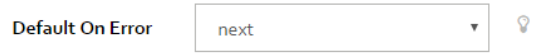
This field indicates the name of the Engineer who created this Case. This is a prefilled field but, you can change it if you want to. This is a mandatory field.

3.6. LAST UPDATED (DATE-TIME STAMP)

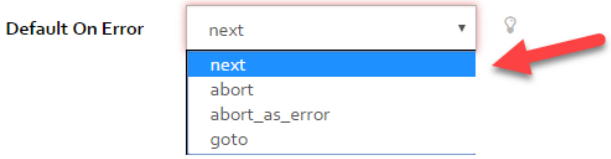
This field indicates the Date and Time on which this Case was last updated. This is a prefilled field, but the contents cannot be changed.

3.7. DEFAULT ON ERROR

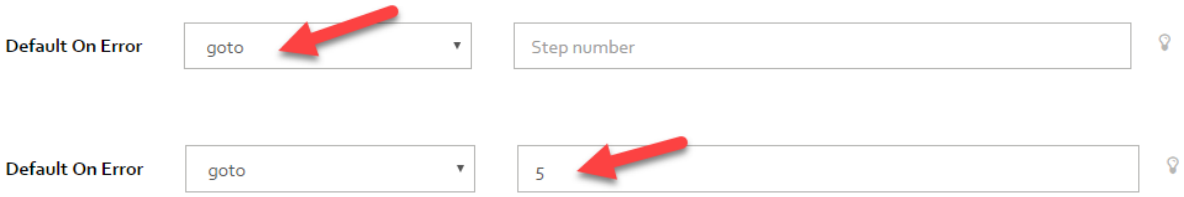
This is field where you can specify what the Case should do it case it errors out. By default, it is "next", that is, if you don't make a change, the Case will proceed to the next Case available for execution if the current case throws an error.



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



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



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

3.8. INPUT DATA FILE

This is field wherein you can add the path to a datafile (Warrior Framework User Guide: Section 9) that you want this case 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.

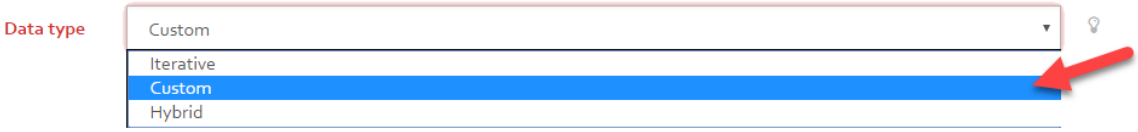
In the case where you want a datafile included in your Case, uncheck the "No Data File" checkbox and add either an absolute path to your datafile or a path relative from your case. It is recommended that you add a relative path here.

You can click on the "Path" button, which would open up a file explorer which would let you select a file.

You can also directly enter the path into the field.

3.9. DATA TYPE

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



3.10. 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.

3.11. 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.

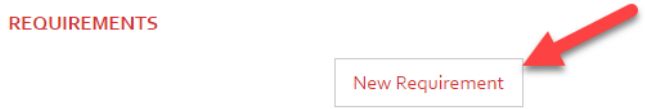
If left empty, the results would be directed and stored in the default location.

12. Expected Results

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

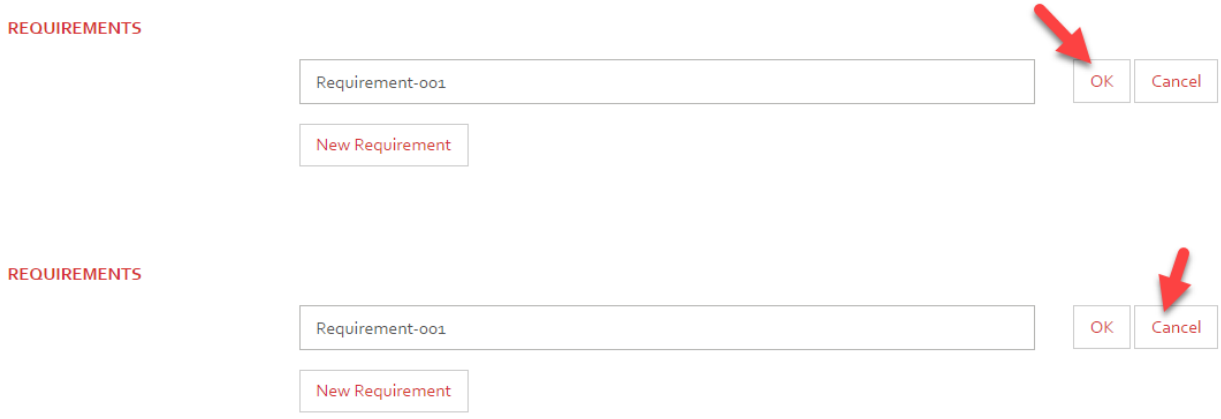
3.12. REQUIREMENTS

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

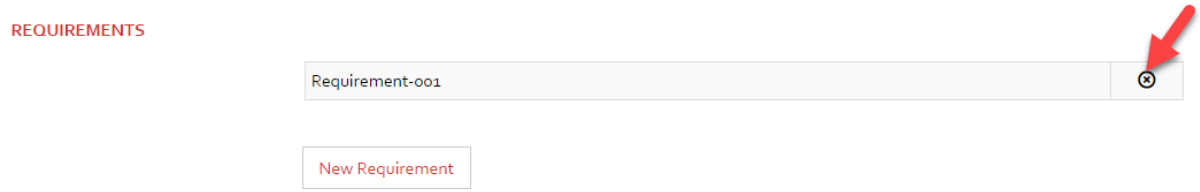


This opens up a new field in which you can specify the requirement.

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

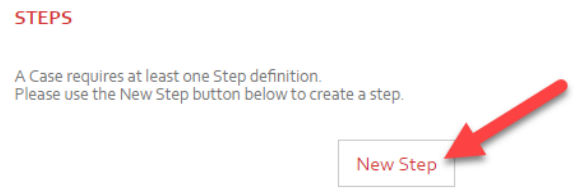


You can also delete an added requirement by clicking the minute “x” against the Requirement that you want to delete.

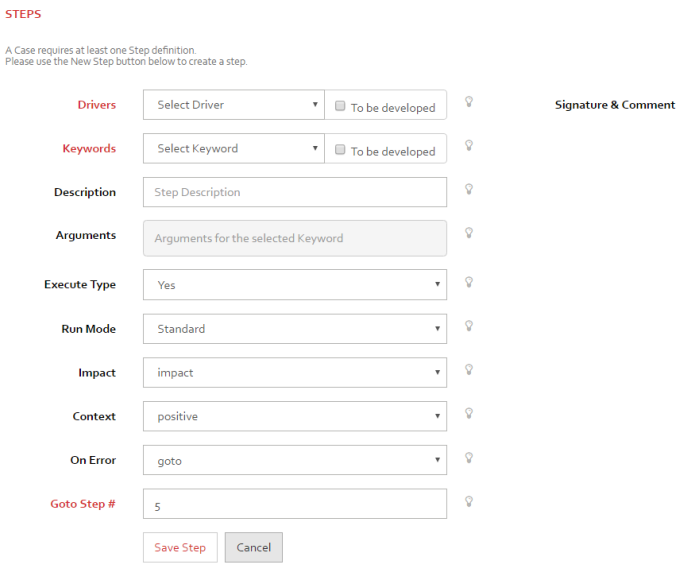


3.13. STEP

Every Case requires at least one step. A new step can be added to the Case 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”.

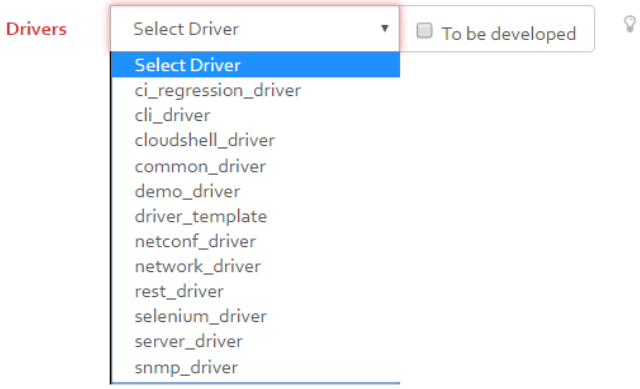


3.14. DRIVERS

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

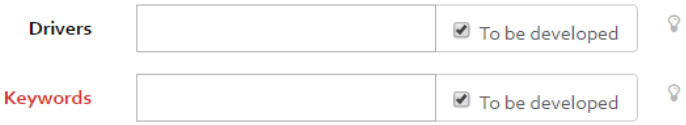
Katana lets you select a driver by showing all the drivers that you have in your Warrior Framework directory in the dropdown. You can select any driver out of those. You would not be able to select any driver which lies outside of the ProductDrivers directory. This is because Warrior enforces a strict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



You can then select any driver from that list:

If you want to include a Driver that has not been developed yet, you can simply check the “To be developed” box and add a driver name of your own.



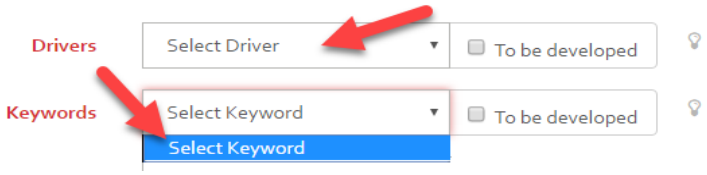
Be informed that checking the “To Be Developed” box for the driver automatically checks the “To Be Developed” box for the Keyword as a keyword for a non-developed driver cannot exist,.

You would then be able to include a driver name of your choice. You can also leave this field blank as checking the “To Be Developed” box renders the Drivers field as non-mandatory.

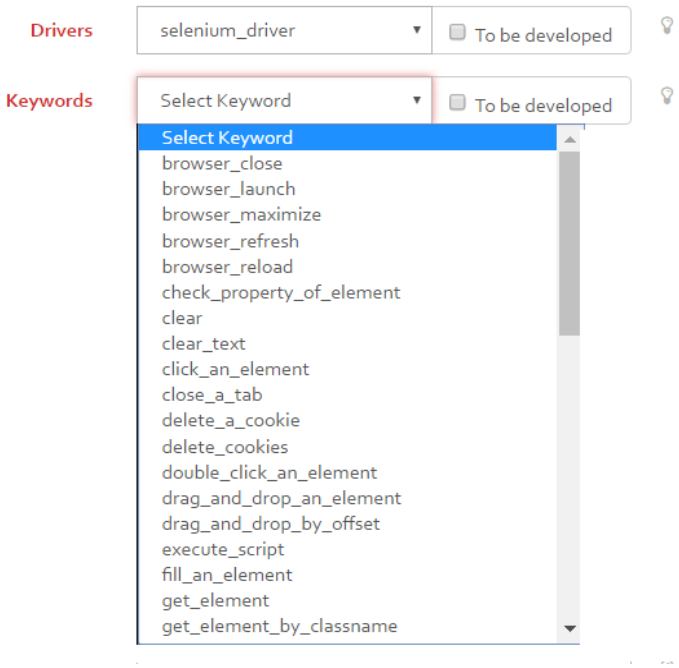
3.15. KEYWORDS

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.



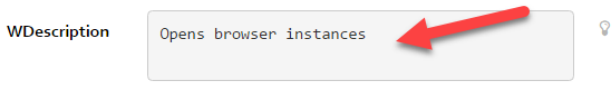
You can then select any keyword that you need.

If you want to include a Keyword that has not been developed yet, you can simply check the “To be developed” box and add a keyword name of your own.

A keyword name – even when you are including a non-developed keyword – is a mandatory field.

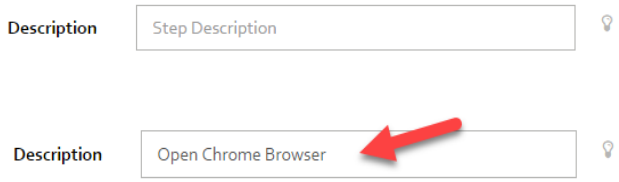
3.16. WDESCRIPTION

This field is populated when the Keyword is chosen. The description is for your reference, so that, you know exactly what this keyword does. If a non-developed keyword is chosen, this field would remain empty. This field is a non-editable field.



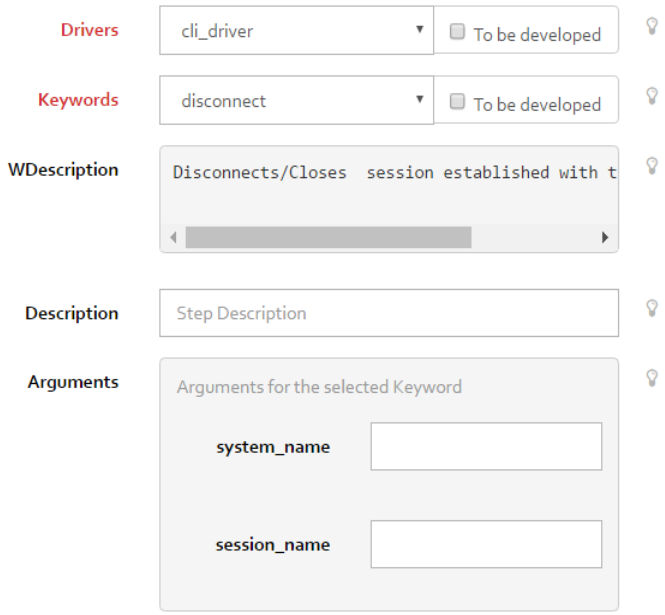
3.17. DESCRIPTION

This field is provided to you so that you can add a description of your own for the chosen Keyword. This description would be displayed when this steep is run. This is an optional field.

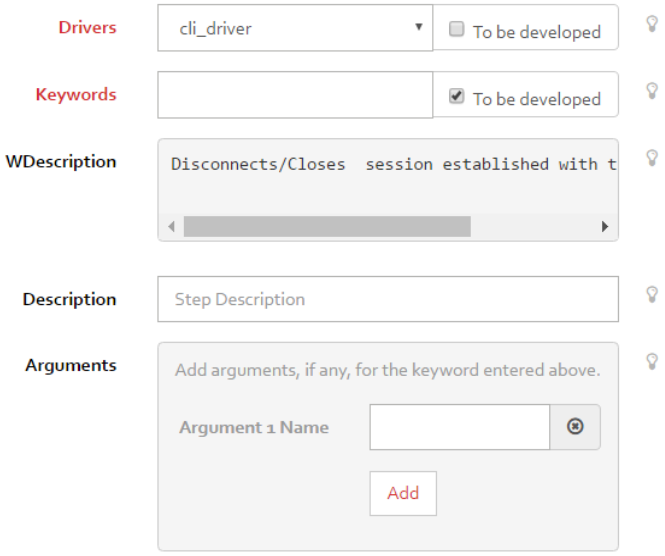


3.18. ARGUMENTS

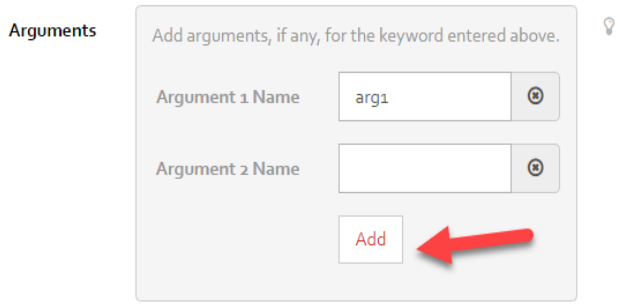
This space gets populated when a developed keyword is selected. All the arguments accepted by the Keyword show up here. You can type in the values associated with each value against that argument name. To identify which are the mandatory arguments, you can go through the next section.



When a non-developed keyword is selected, this field is populated a little differently.

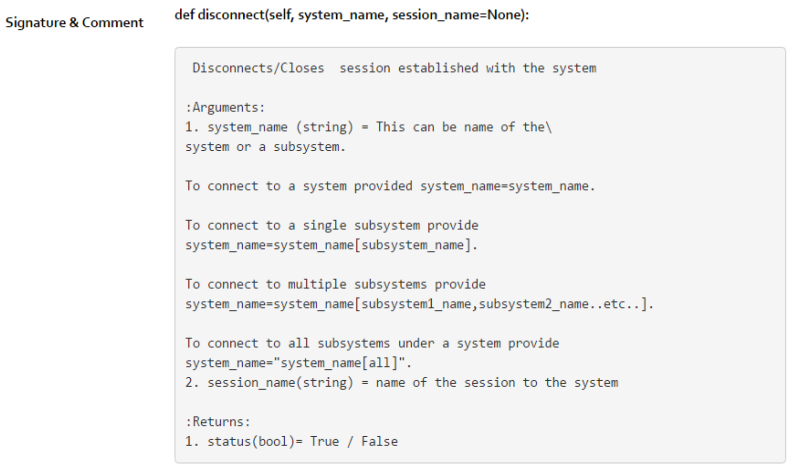


Here, in the space where Arguments for a developed keyword should show up, input fields that will let you define the arguments that you think this keyword should have would be displayed. You can then add the arguments names for the non-developed keyword.



3.19. SIGNATURE & COMMENTS

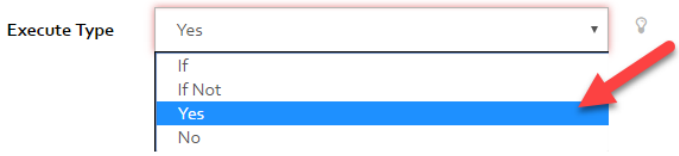
This field also appears prepopulated when a developed Keyword is selected. This is for your benefit, so that you understand what the keyword does when it runs, what kind of arguments it accepts, and what kind of values it returns.



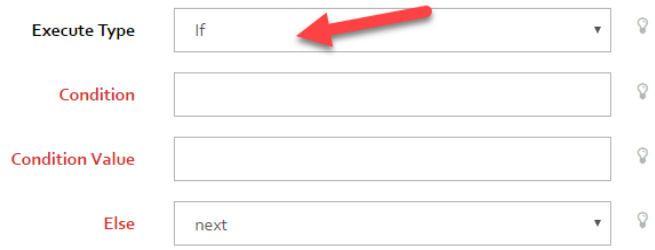
This field would remain blank when a non-developed keyword is selected.

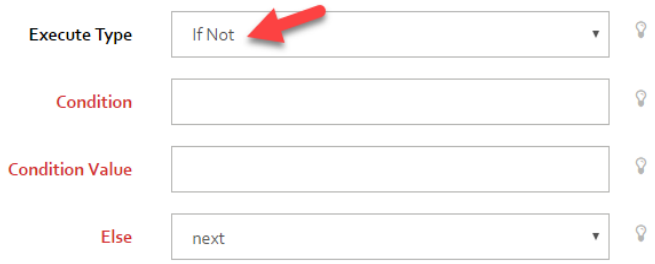
3.20. EXECUTE TYPE

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.

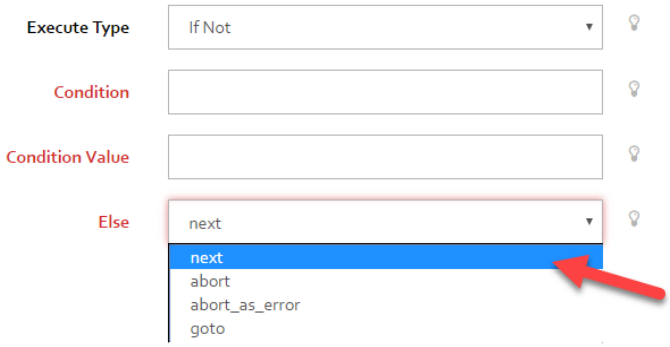


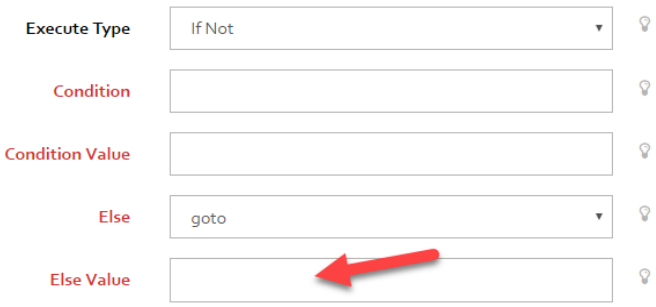
On clicking on 'If' and 'If Not', two other input boxes show up that let you add your conditions.





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.

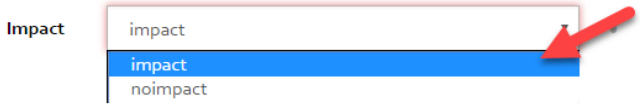




To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.1.

3.21. IMPACT ON FAILURE

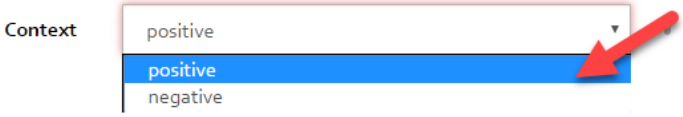
On clicking the dropdown you would be able to select from one of the options presented to you. The default is 'impact'.



To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.6 and Section 11.7.

3.22. CONTEXT

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

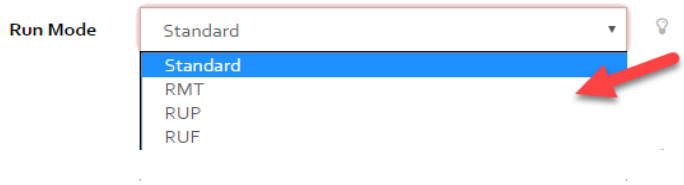


To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.8 and Section 11.9.

3.23. 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.

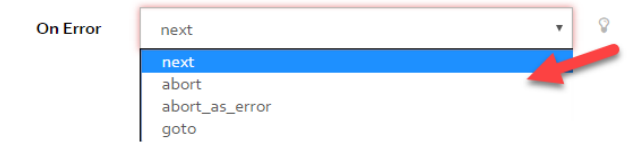
You can input the number of times you want this particular step to run here.

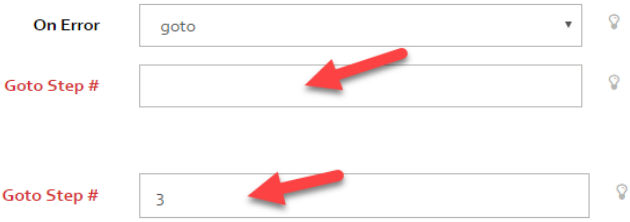
To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.2, Section 11.3, Section 11.4, and Section 11.5.

3.24. ON ERROR

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

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





To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 11.10, Section 11.11, Section 11.12, and Section 11.13.

3.25. SAVE STEP

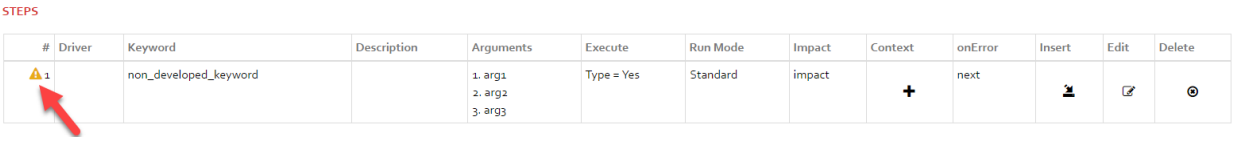
Clicking on the 'Save Step' button saves this step. This DOES NOT save the case – only that particular step. A step must be saved

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

STEPS

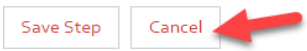
#	Driver	Keyword	Description	Arguments	Execute	Run Mode	Impact	Context	onError	Insert	Edit	Delete
1	selenium_driver	browser_launch	Browser will be launched	1. system_name = browser_1 2. browser_name = chrome_1 3. type = chrome 4. url = http://www.google.com	Type = Yes	Standard	impact	+	goto 3			

If a Case contains a step with a non-developed keyword, then, a small yellow warning sign would appear along with the step number:



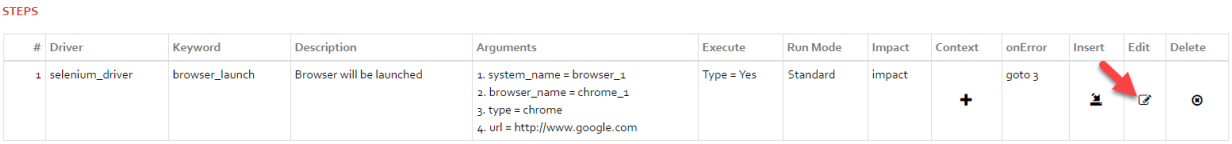
3.26. CANCEL STEP

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

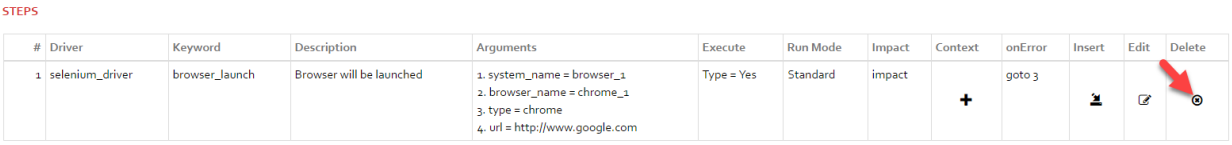


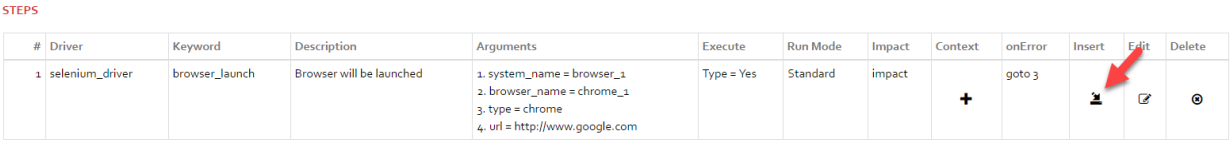
3.27. EDIT STEP

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



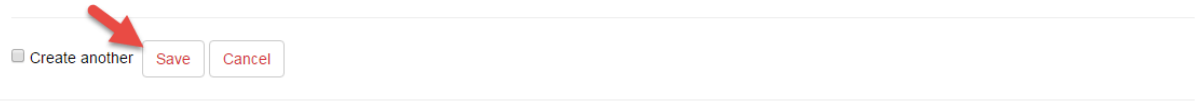
This opens up the step editor once again and you can make the desired changes.



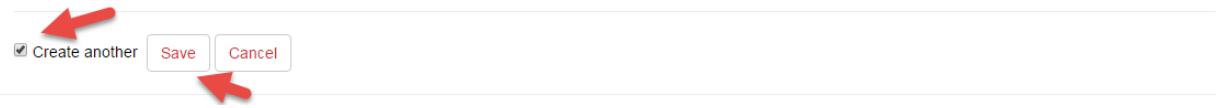


3.28. SAVE CASE

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

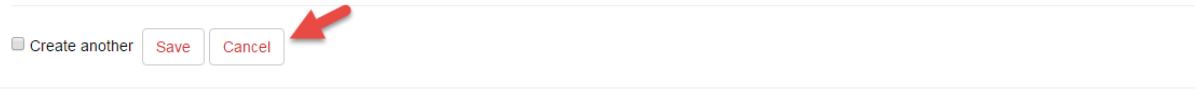


Checking the box against “Create Another” and then hitting Save, lets you save the case and directly opens up a page where you can start creating a new Case.

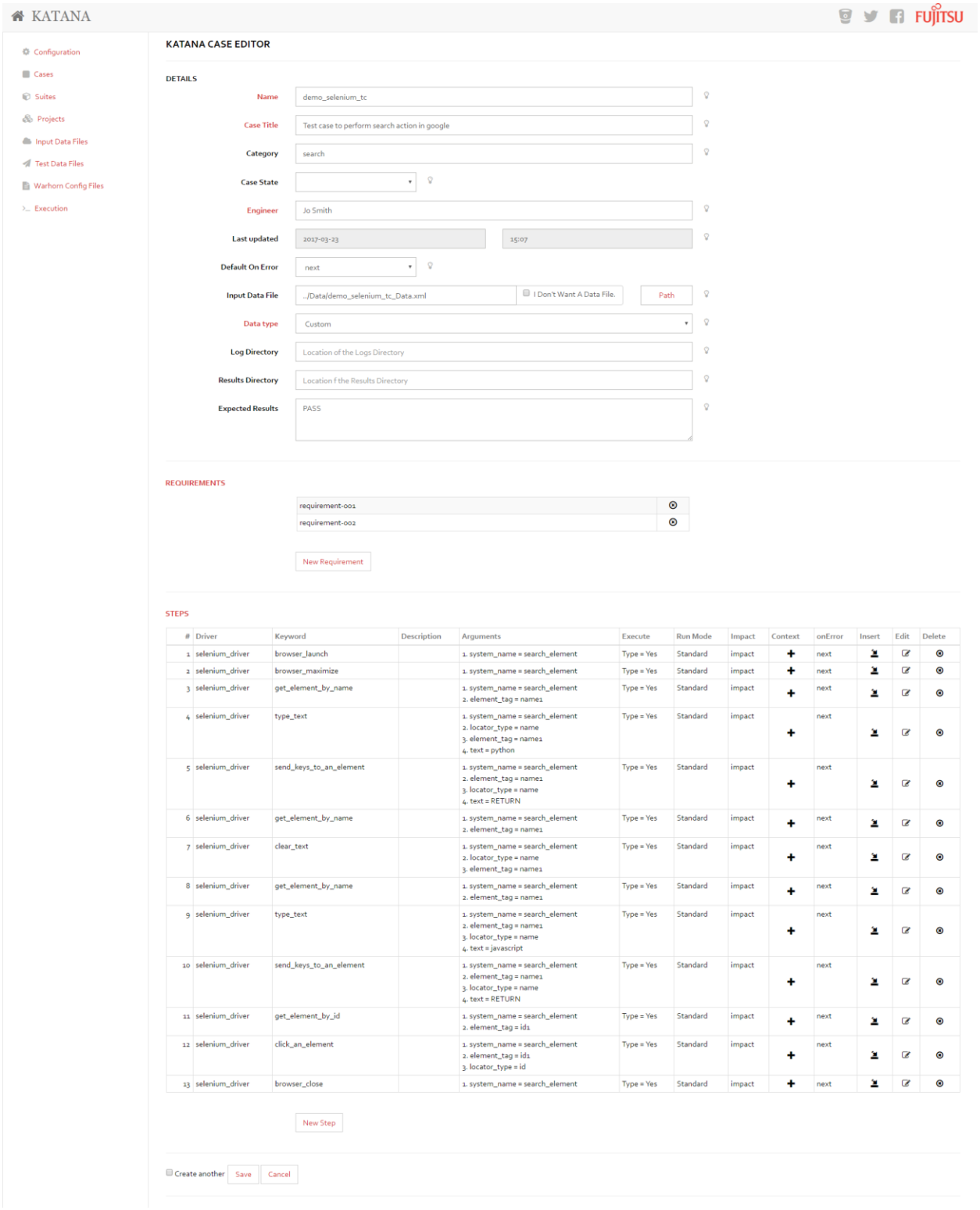


3.29. CANCEL CASE

Clicking on ‘Cancel’ discards all the changes performed.

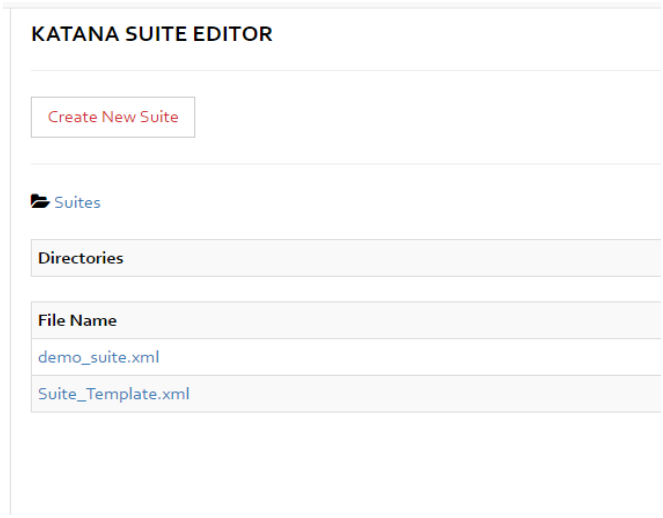


The case thus created would look something like this:



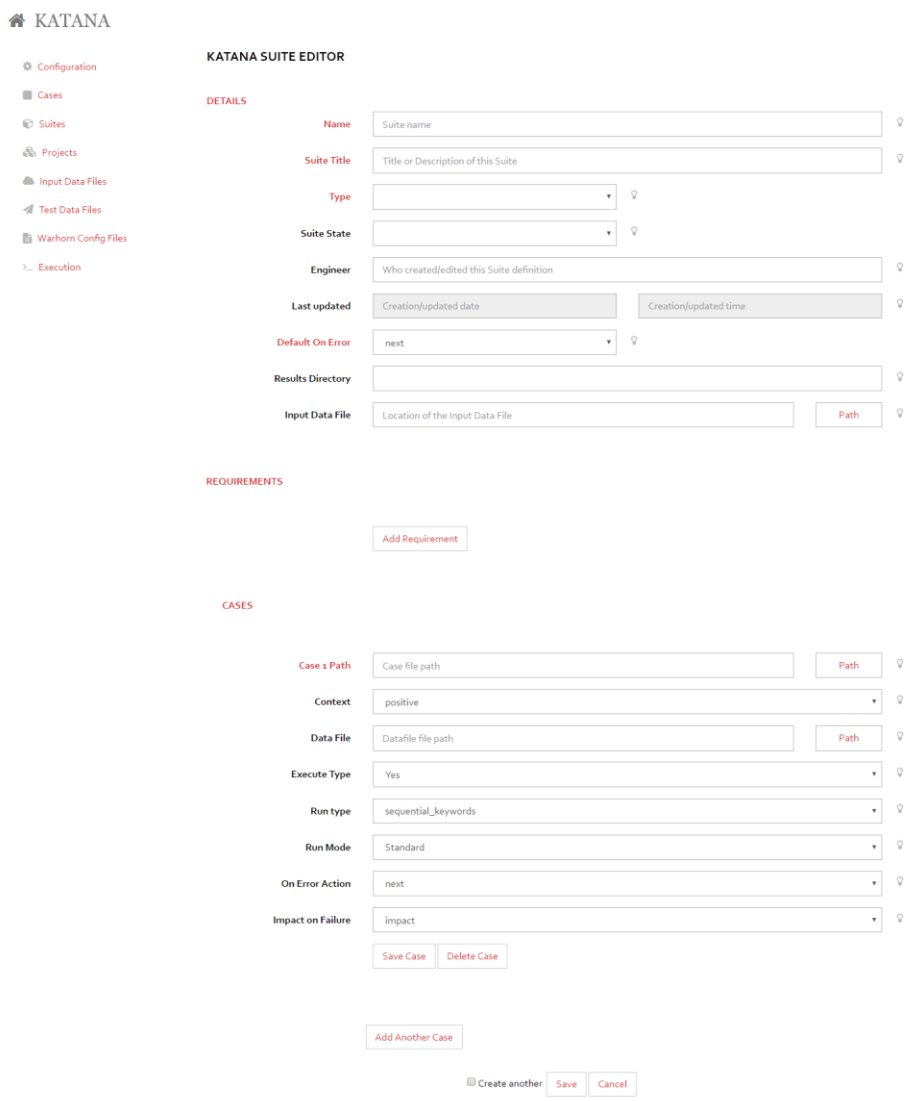
4. CREATING SUITES WITH KATANA

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



Note: The list of suites below the “New Suite” button may differ as Katana would show the Suites saved in your suite directory.

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



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

4.1. SUITE NAME

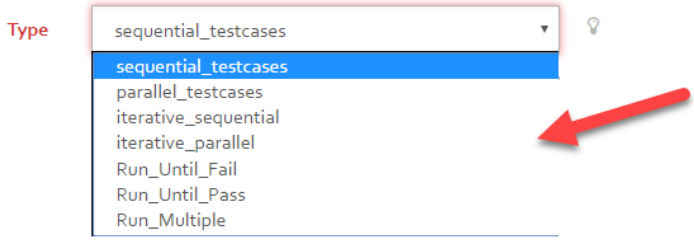
Here enter the name of the suite. A name can be anything that you recognize. This is a mandatory field.

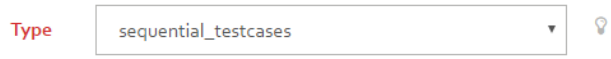
4.2. SUITE TITLE

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

4.3. TYPE

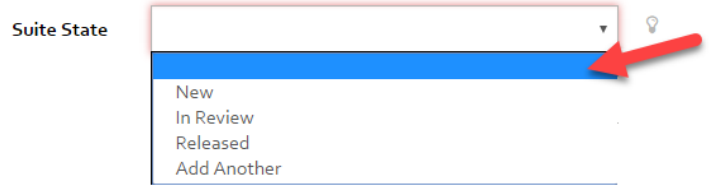
This is a mandatory field that lets you determine how the suite should run. You can click on the dropdown and select the 'type' of the suite.





4.4. 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 case state that suits the state in which your case is, such as “Test-Assigned”, and click “Done”.

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

4.5. ENGINEER

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

4.6. RESULTS DIRECTORY

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

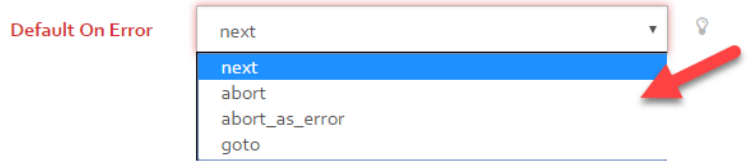
4.7. LAST UPDATED (DATE-TIME STAMP)

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

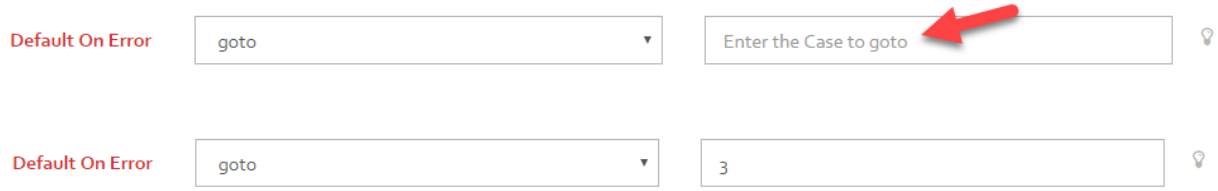
4.8. DEFAULT ON ERROR

This is field where you can specify what the Suite should do it case it errors out. By default, it is “next”, that is, if you don’t make a change, the Suite will proceed to the next Suite available for execution if the current suite 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 Case errors out. You have the ability to choose any one of them.



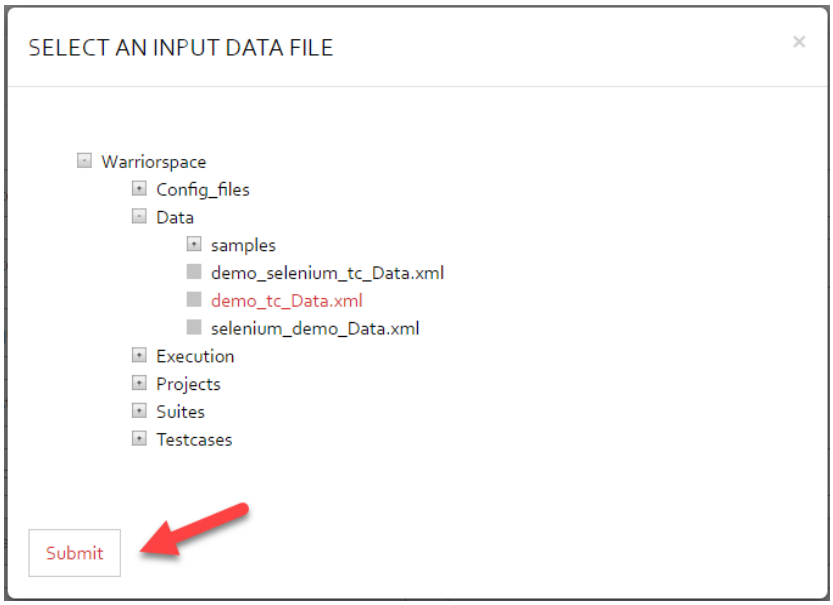
On selecting “goto”, another pops up and lets you add the Case number that Warrior should go to in case of an error.



4.9. INPUT DATA FILE

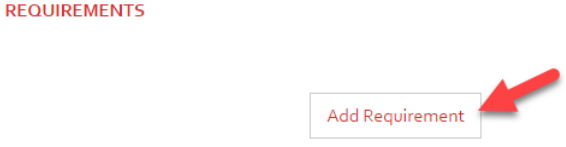
This is field wherein you can add the path to a datafile (Warrior Framework User Guide: Section 9.0) that you want this case to use. You can click on the 'Path' button to choose a path

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

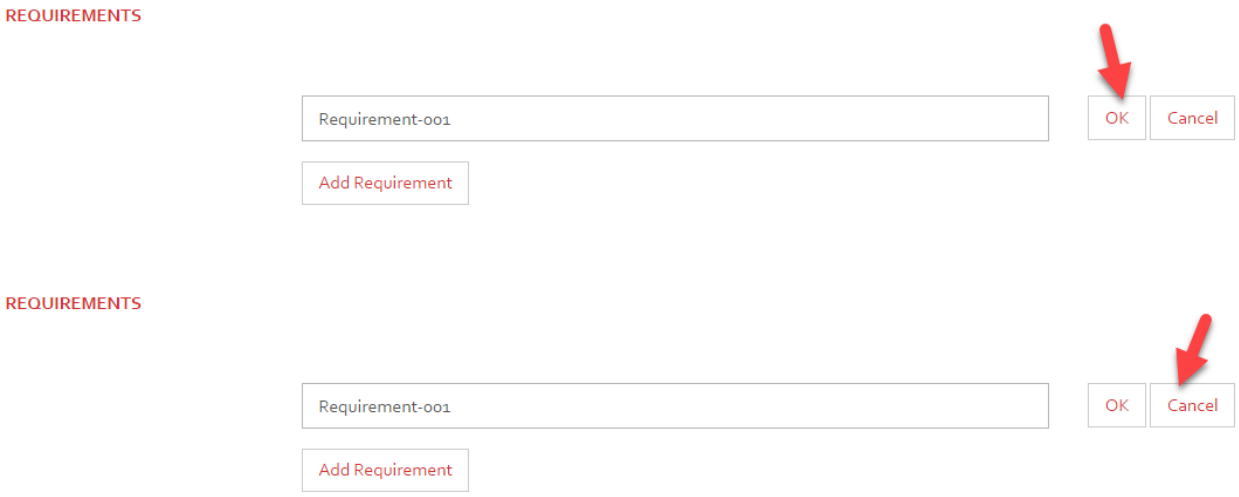


Or you can simply type in a path to your datafile.

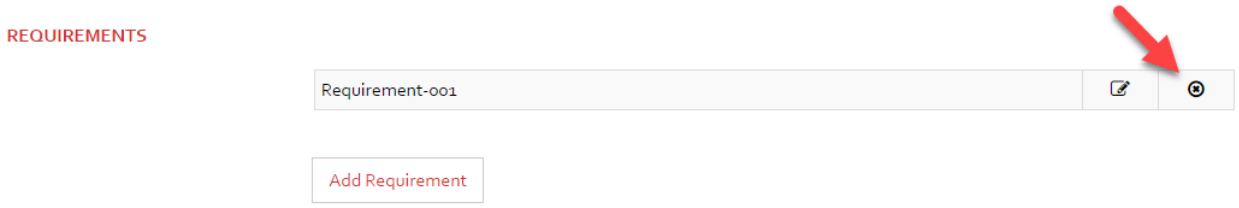
4.10. REQUIREMENTS



This field is for you add any requirements that this suite may have. This is an optional field.



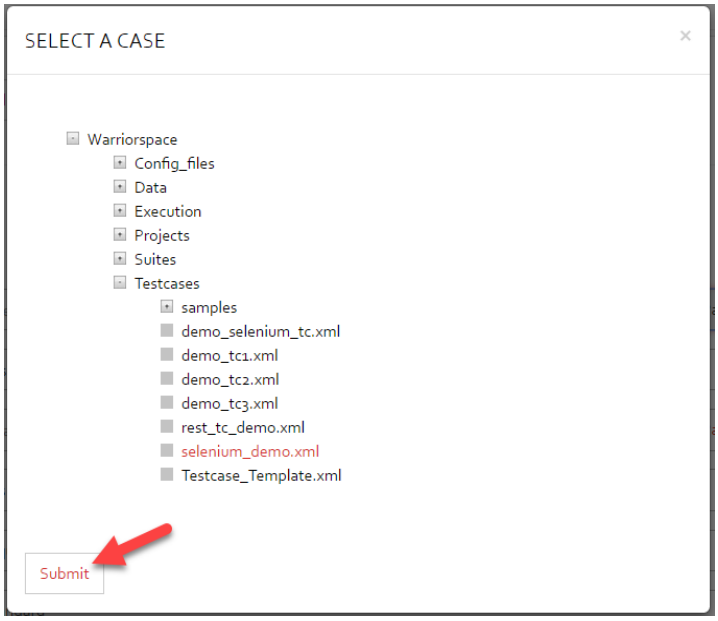
You can also delete an added requirement by clicking the 'delete' button against the Requirement that you want to delete.



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

4.11. PATH

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

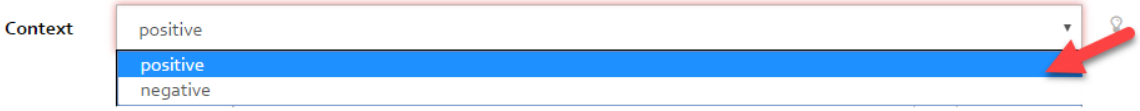


Then, you will be able to select a path to your case, and to enter this path you can hit 'Submit'.

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

4.12. CONTEXT

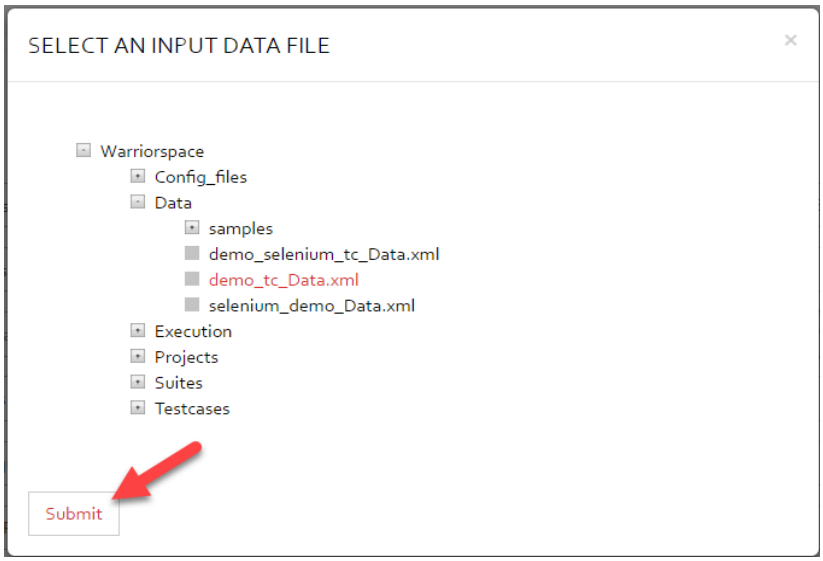
On clicking the dropdown you would be able to select from one of the options presented to you. The default is 'positive'.



4.13. DATA FILE

This field is for a path to the data file. You can enter the path by clicking on the 'Path' button.

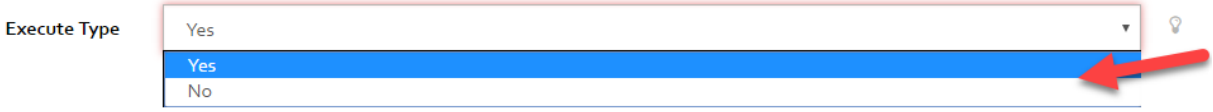
A pop up would appear so that you can select a data file.



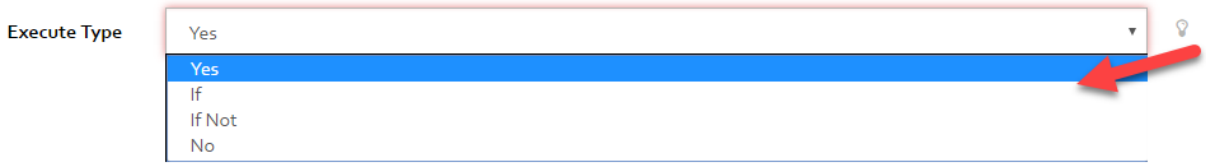
4.14. EXECUTE TYPE

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.

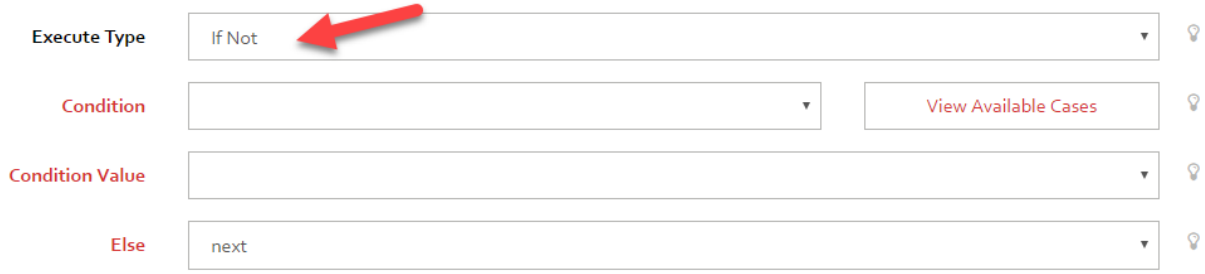
For the first Case, only "yes" or "no" would show up.



After that, for every Case, these options would be available to choose.

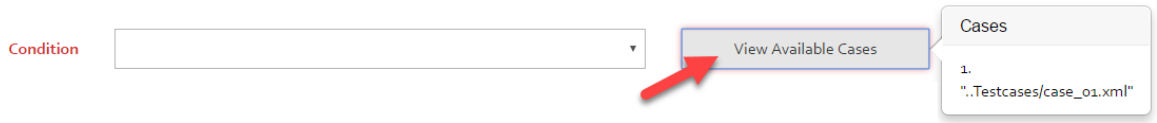


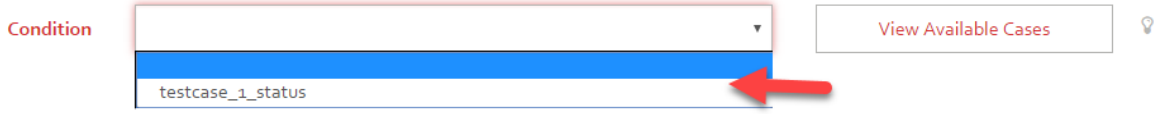
On clicking on 'If' and 'If Not', two other input boxes show up that let you add your conditions.



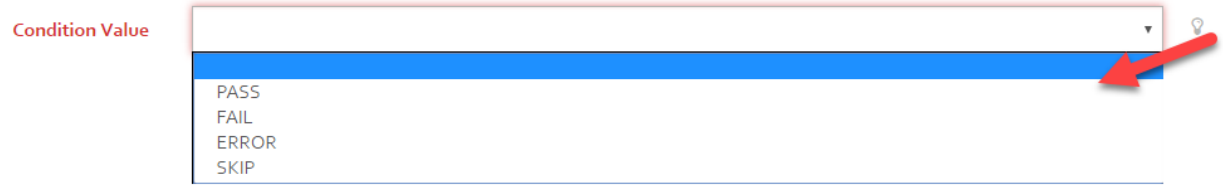
The third input box is a dropdown consisting of the condition that handles the scenario where the condition in 'If'/'If Not' is not met and hence Warrior would need to know what to do in such a case.

You would be able to enter the condition for the Execute Type "If"/"If Not":





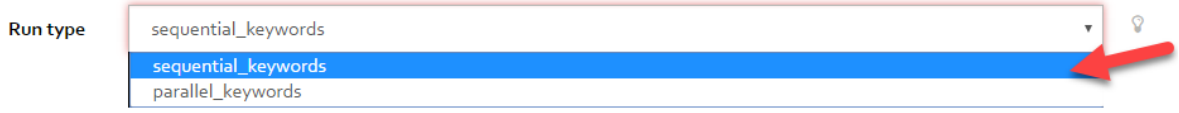
Then, you would be able to enter the condition value for the condition:



To read about how to fully utilize this feature, please refer to Warrior User Guide: Section 12.6.

4.15. RUN TYPE

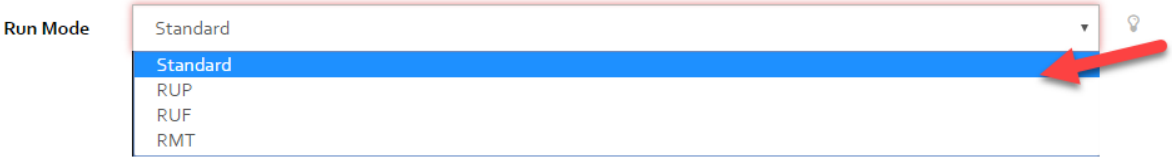
The run type dictates how the steps in a Case would run. The default is “sequential_keywords”.



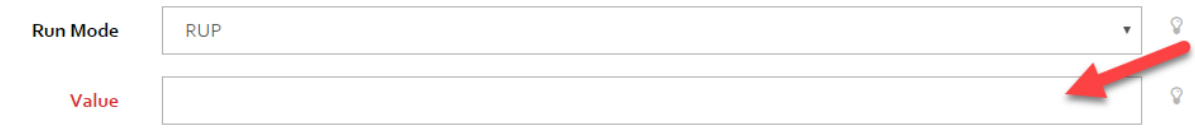
4.16. RUN MODE

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

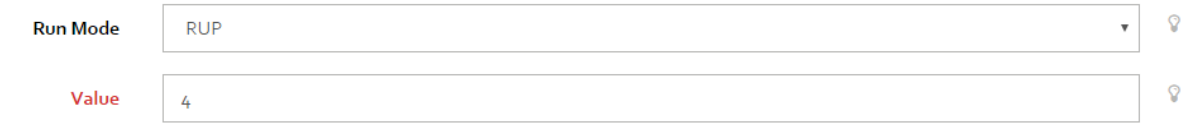
The Run Mode field is dropdown consisting of values “RUP”, “RUF”, and “RMT”.



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 case – if RUP or RUF is selected or the number of times you want to run this case if RMT is selected.

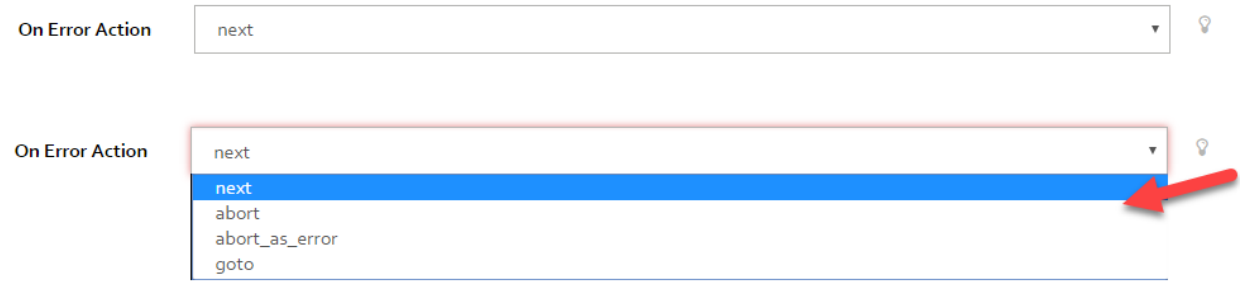


You can input the number of times you want this particular case to run here.



4.17. ON ERROR ACTION

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

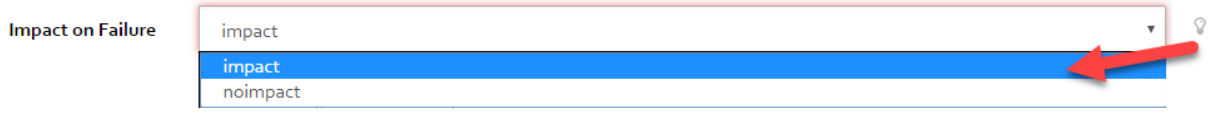


If goto is selected, then, another input box appears that lets you add the number of case to which you want to jump in case of failure.

4.18. IMPACT ON FAILURE

The Impact on Failure checkbox is checked by default.

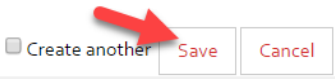
You can uncheck it if you want the result of this particular case to not affect the entire suite.



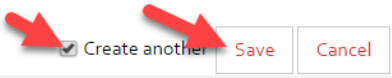
4.19. ADD CASES

4.20. SAVE SUITE

Clicking on the 'Save' button saves this suite.



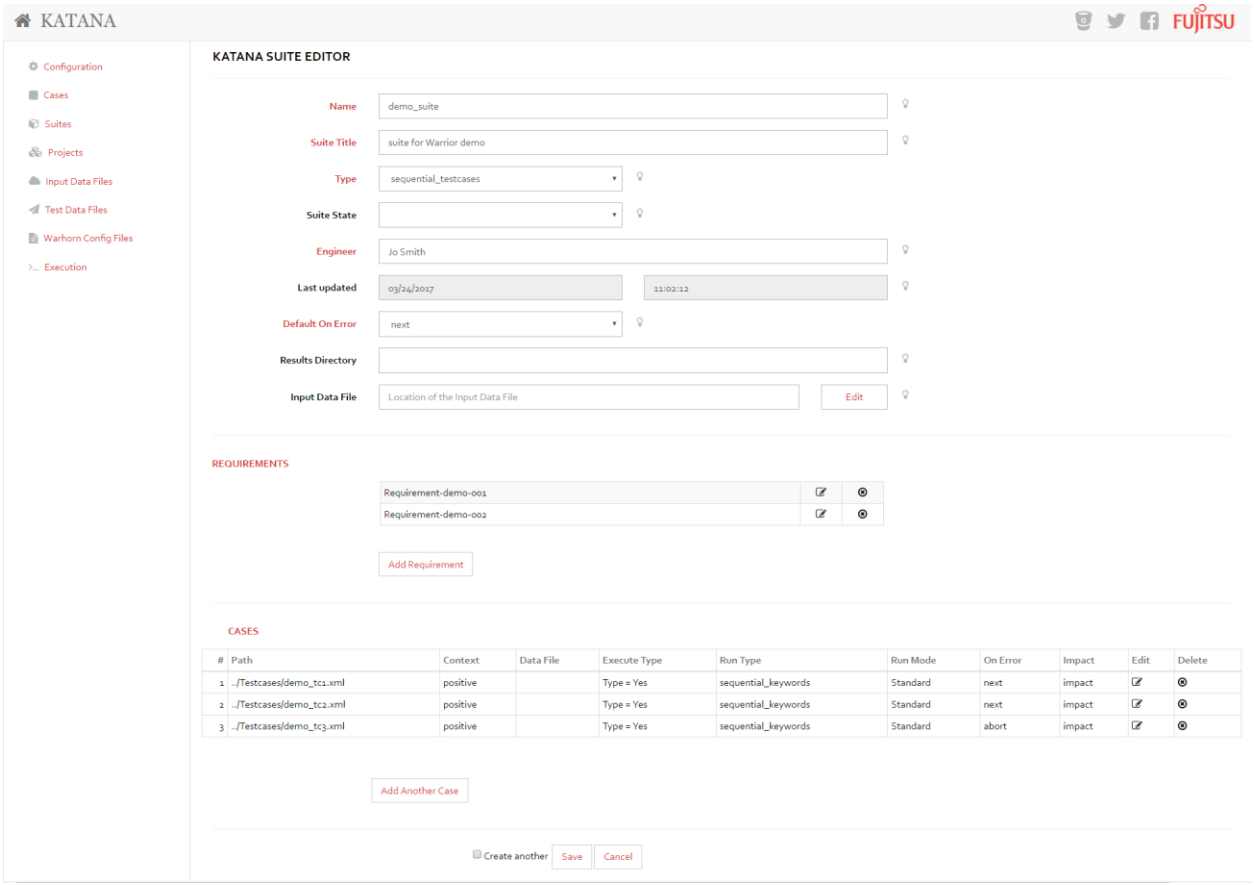
Checking the 'Create Another' checkbox, lets you save the suite and then open a new fresh suite page for you to start creating a new suite.



4.21. CANCEL SUITE

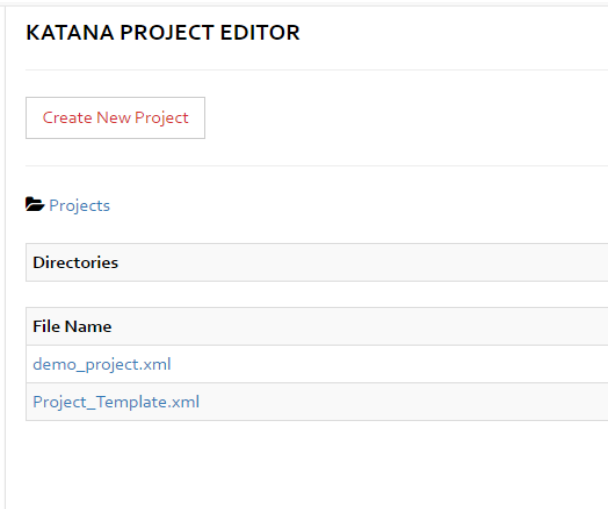
On the other hand, clicking on 'Cancel' discards all the changes made to this suite.

The suite thus created looks something like this:



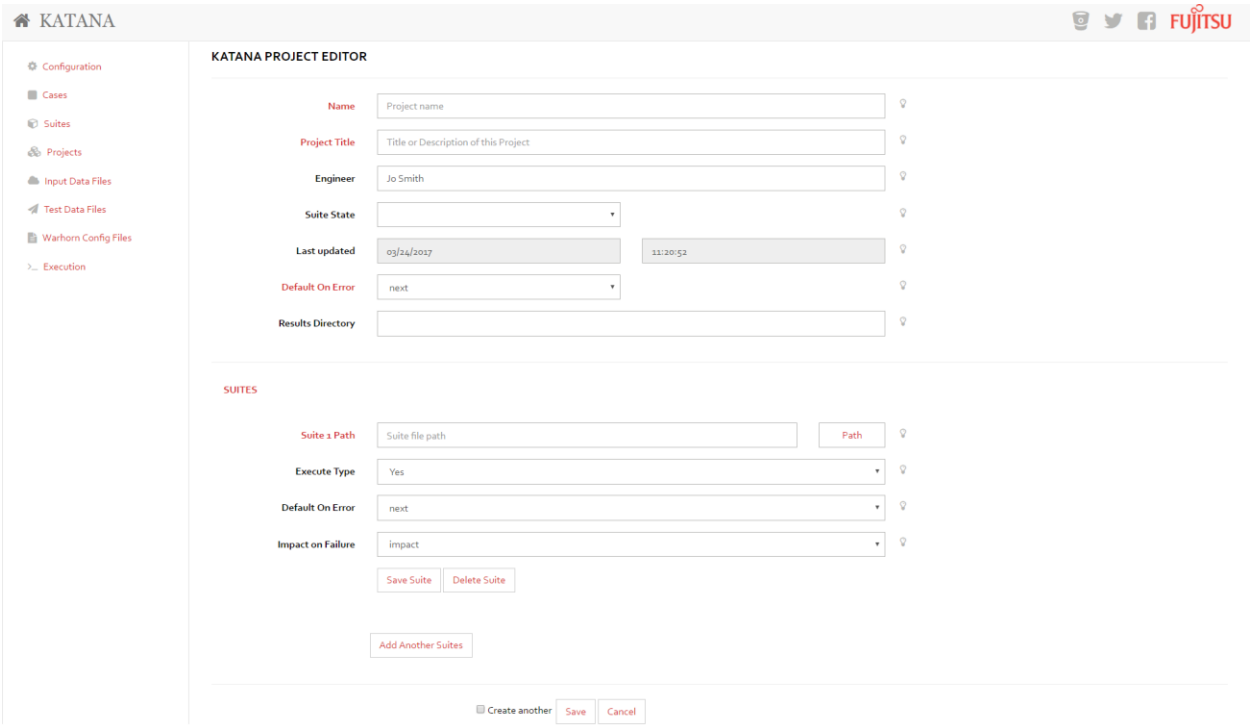
5. CREATING PROJECTS WITH KATANA

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:

5.1. PROJECT NAME

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

5.2. PROJECT TITLE

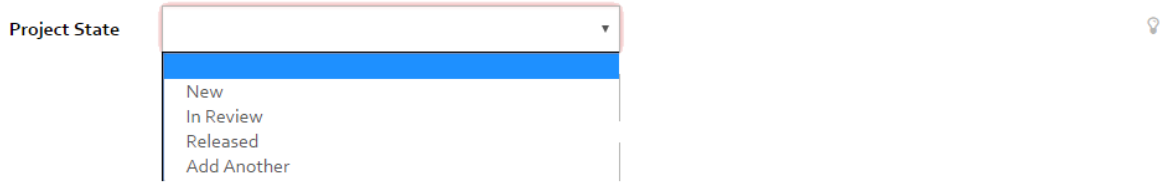
A suite Title (or description) is a field where you can add a descriptive title to identify what this suite does. This is a mandatory field.

5.3. ENGINEER

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

5.4. 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.

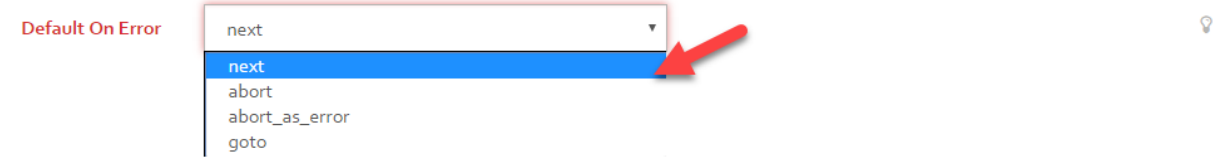
5.5. LAST UPDATED (DATE-TIME STAMP)

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

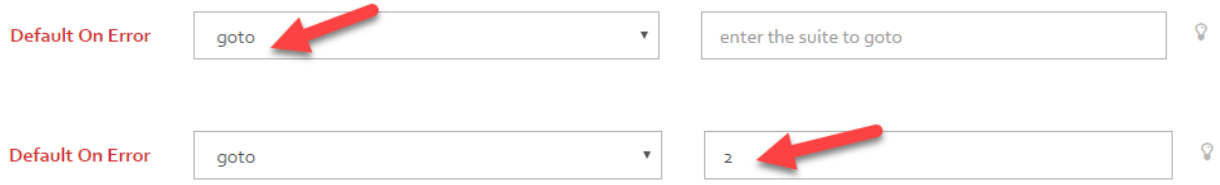
5.6. DEFAULT ON ERROR

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 suite 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.



5.7. RESULTS DIRECTORY

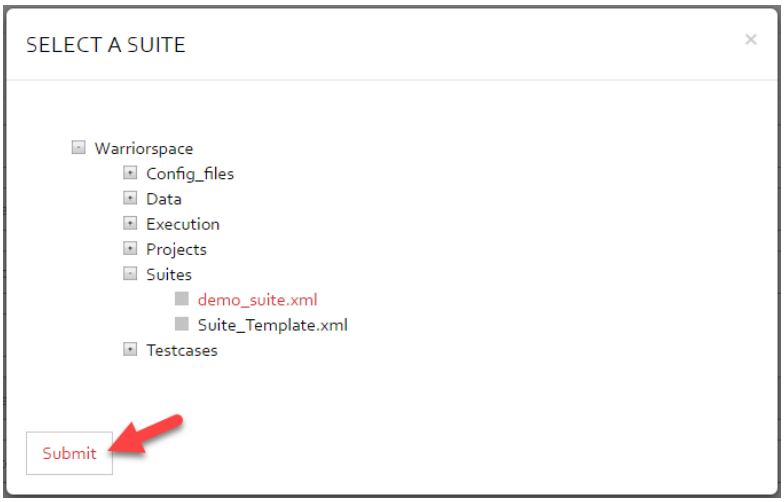
This field lets you specify the path to the Results directory – this directory is where all the results returned by Warrior would be stored. This is an optional field.

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

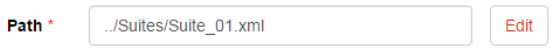
This button lets you delete a Suite that has been added to the project. But the Project needs at least one suite in it.

5.8. PATH

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



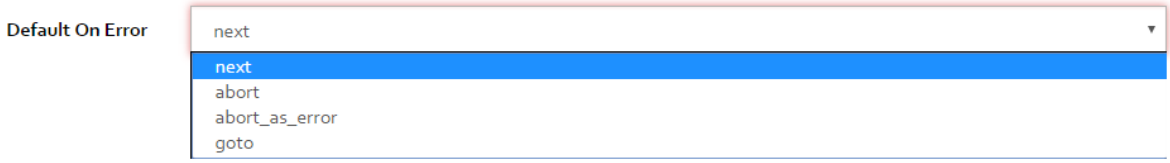
Then, you will be able to select a path to your case, and to enter this path you can hit 'Submit'.

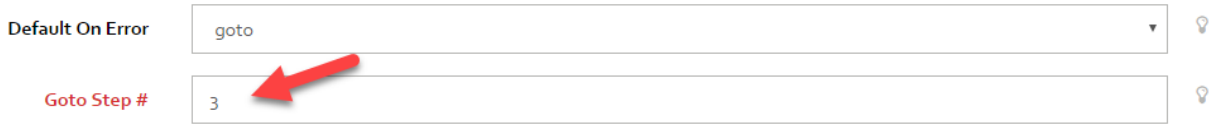


You can also manually add a path to your case. This is a mandatory field.

5.9. 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 suite to which you want to jump in case of failure.

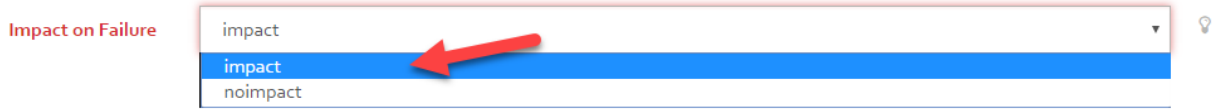




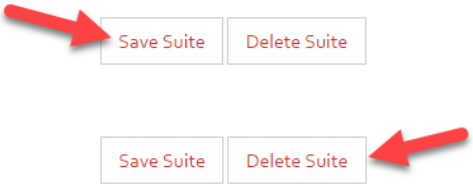
5.10. IMPACT ON FAILURE

The Impact on Failure checkbox is checked by default.

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



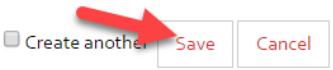
5.11. ADD SUITES



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

5.12. SAVE PROJECT

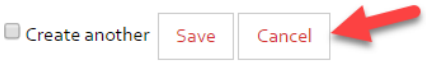
1. Clicking on the 'Save' button saves this project.



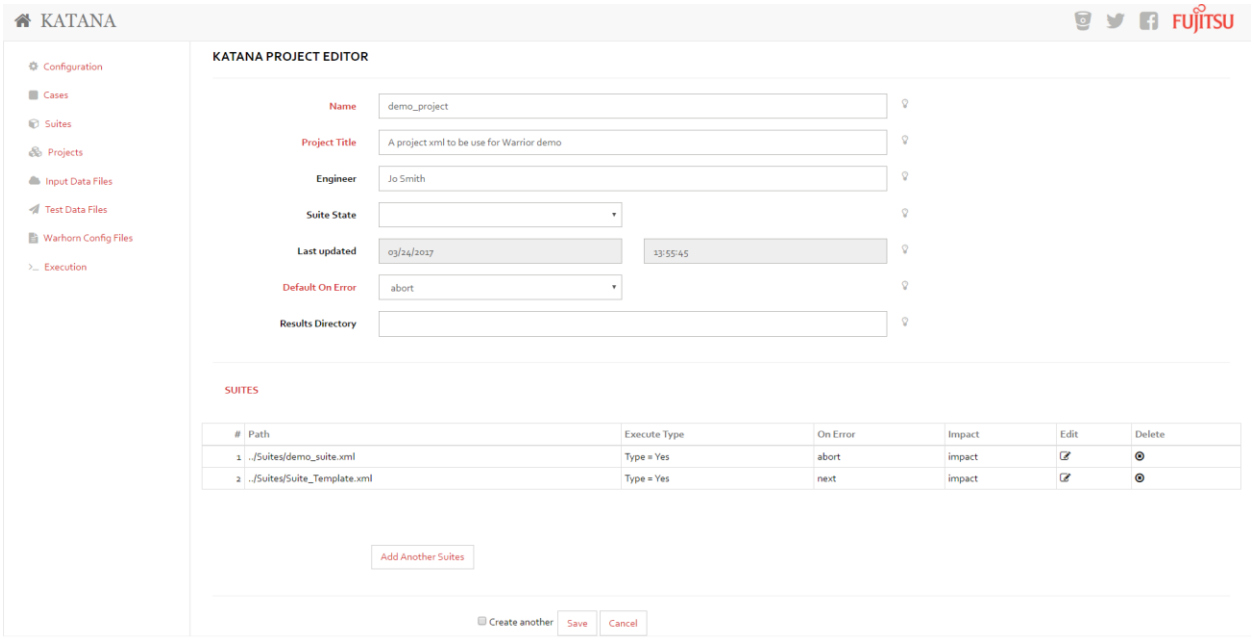
Checking the 'Create Another' checkbox, lets you save the project and then open a new fresh project page for you to start creating a new project.

5.13. CANCEL PROJECT

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

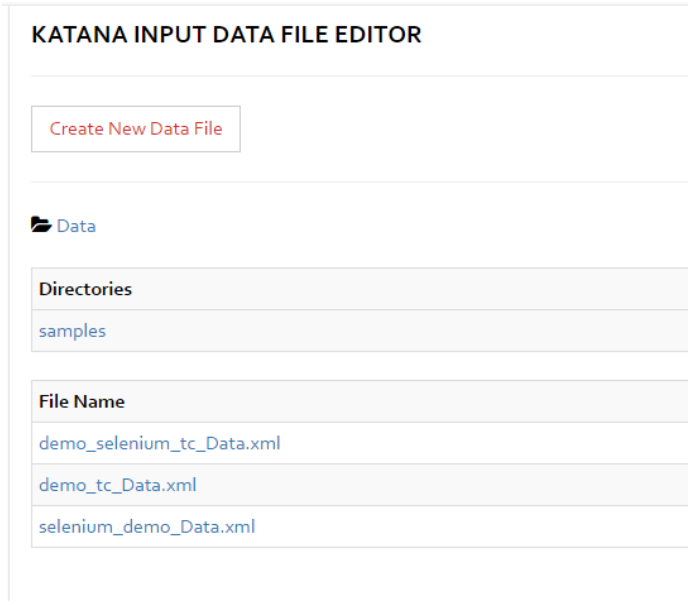


The Project thus created looks something like this:



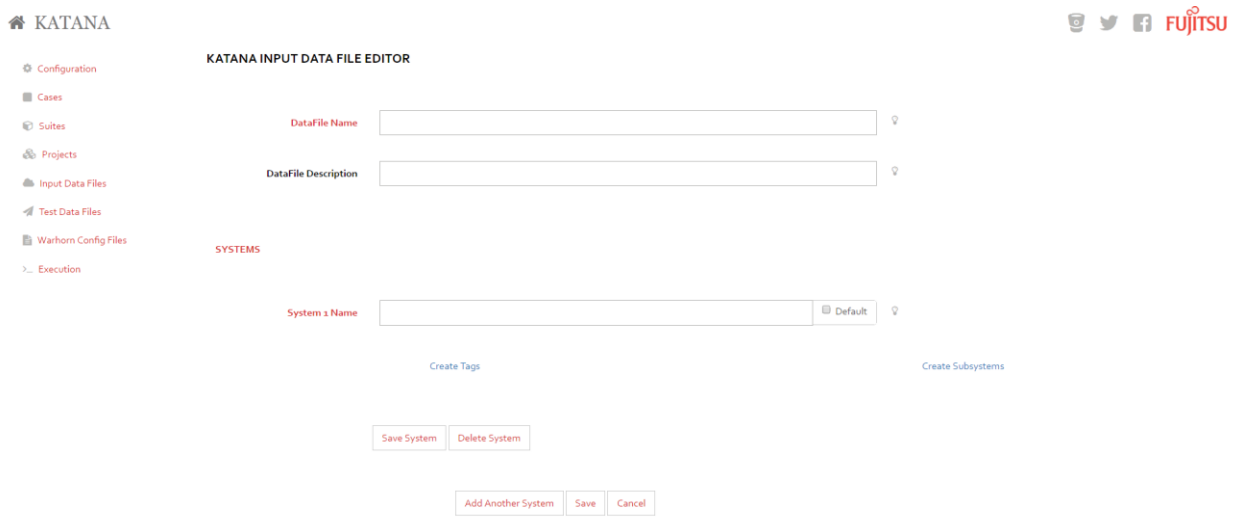
6. CREATING INPUT DATA FILES WITH KATANA

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:

6.1. 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

6.2. DATAFILE DESCRIPTION

You can add a description for the data file here.

6.3. SYSTEMS

You can start creating systems inside this Data file

SYSTEMS

6.4. SYSTEM NAME

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

6.5. DEFAULT SYSTEM

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

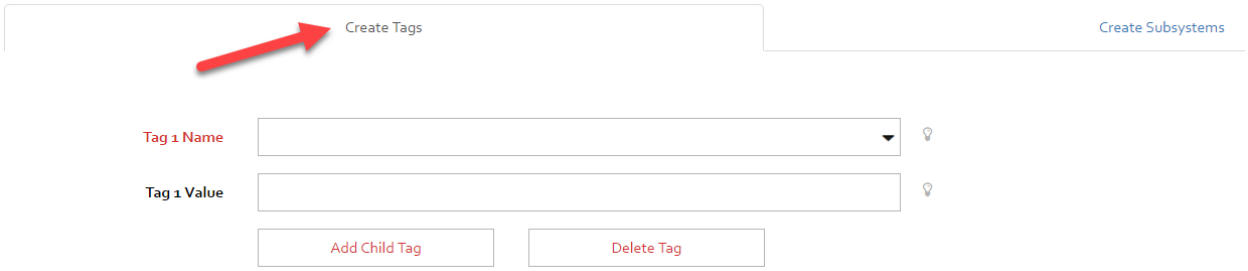
If no system has been set as default then the first system would be treated as the default.

6.6. CREATE TAGS AND SUBSYSTEMS

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

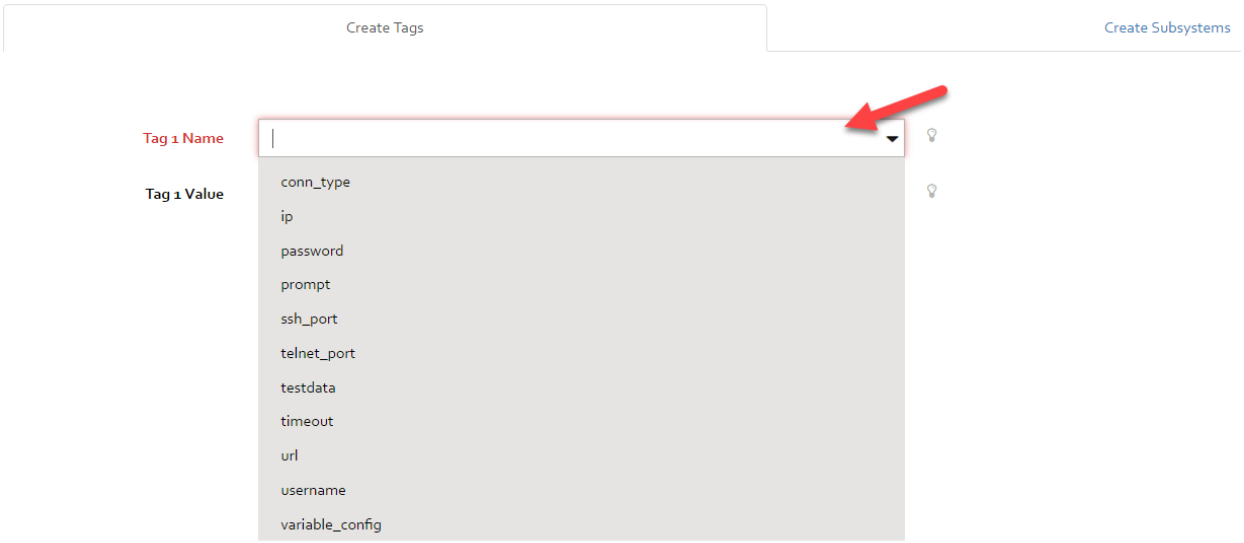
6.7. CREATING TAGS

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

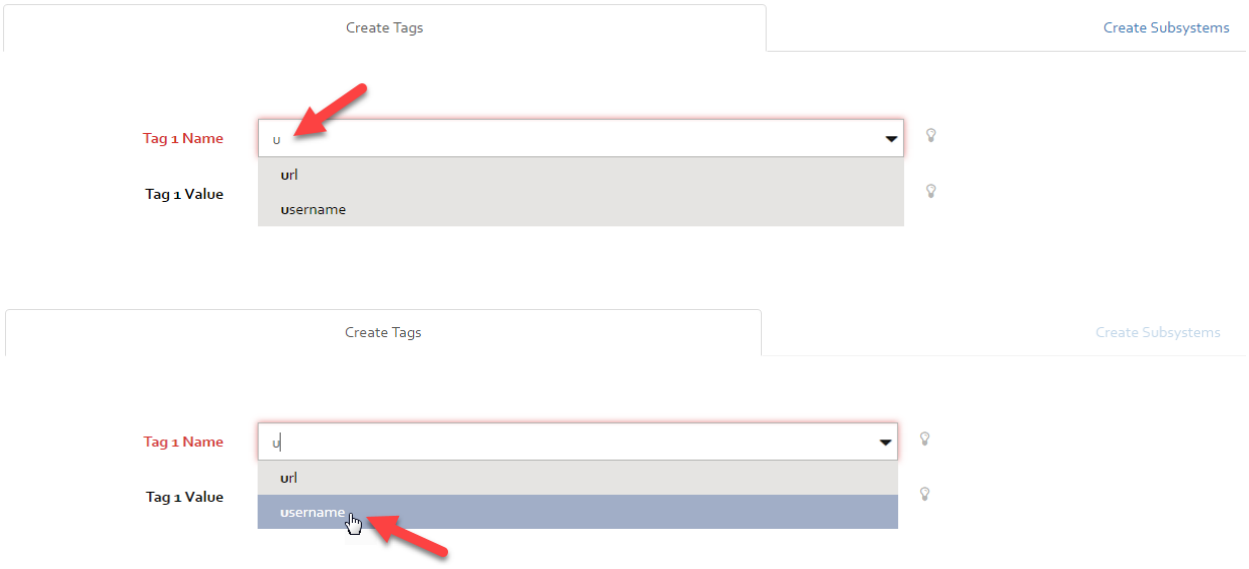


6.8. TAG NAME

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.



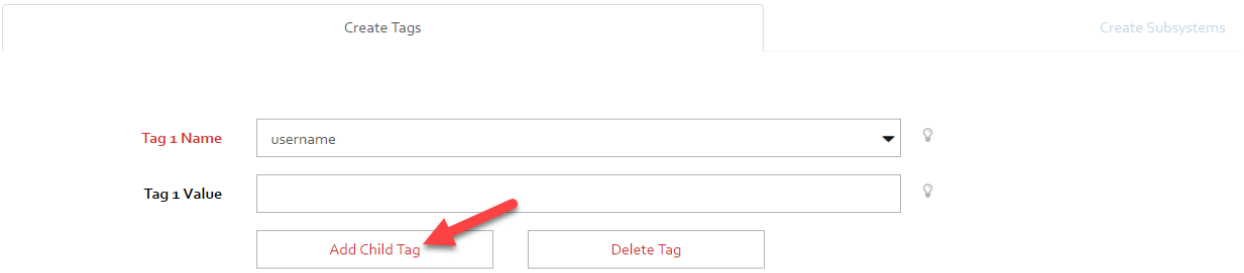
This is a mandatory field.

6.9. TAG VALUE

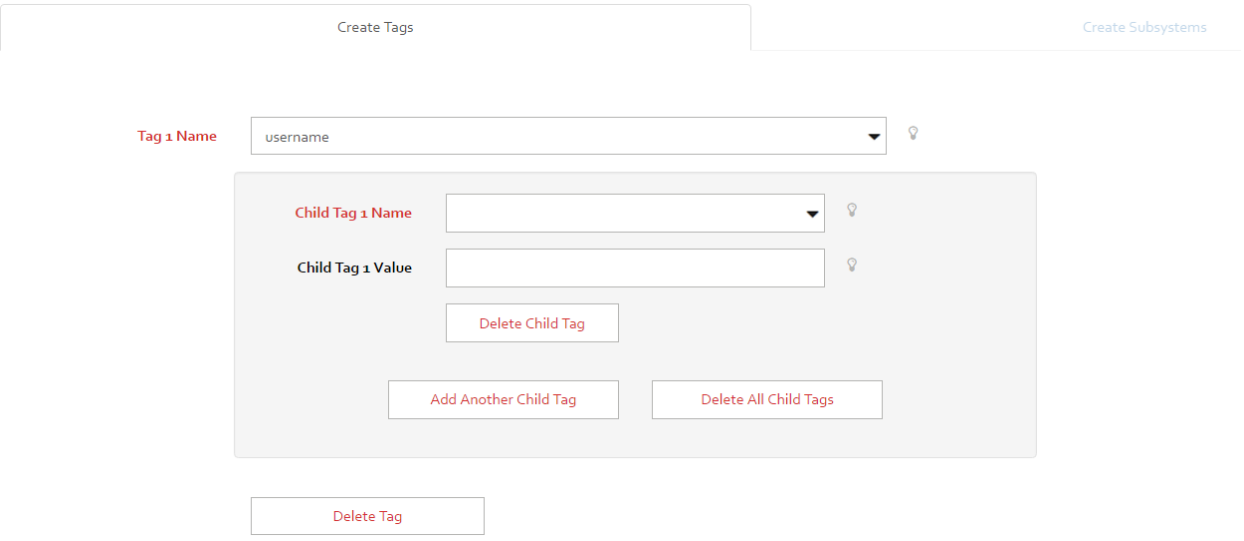
A corresponding value to the tag created above can be added here.

6.10. ADD A CHILD TAG TO TAG

A child tag can be added to each tag created by clicking the “Add Child Tag” button.



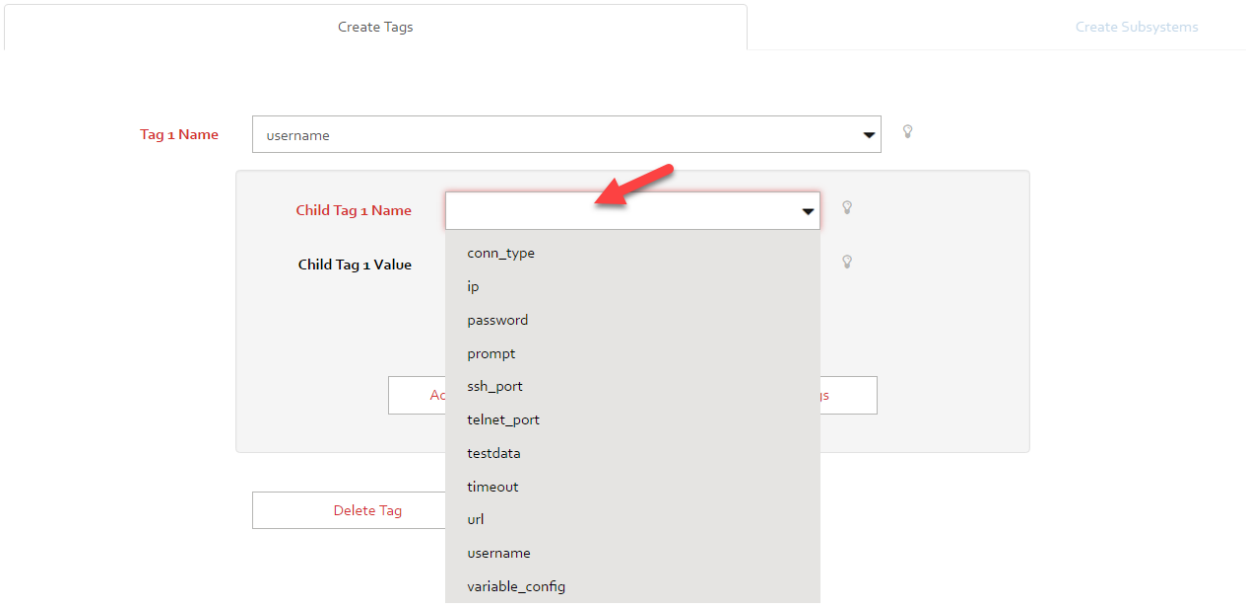
A section will emerge that you let you add a child tag to the tag.



6.11. CHILD TAG NAME

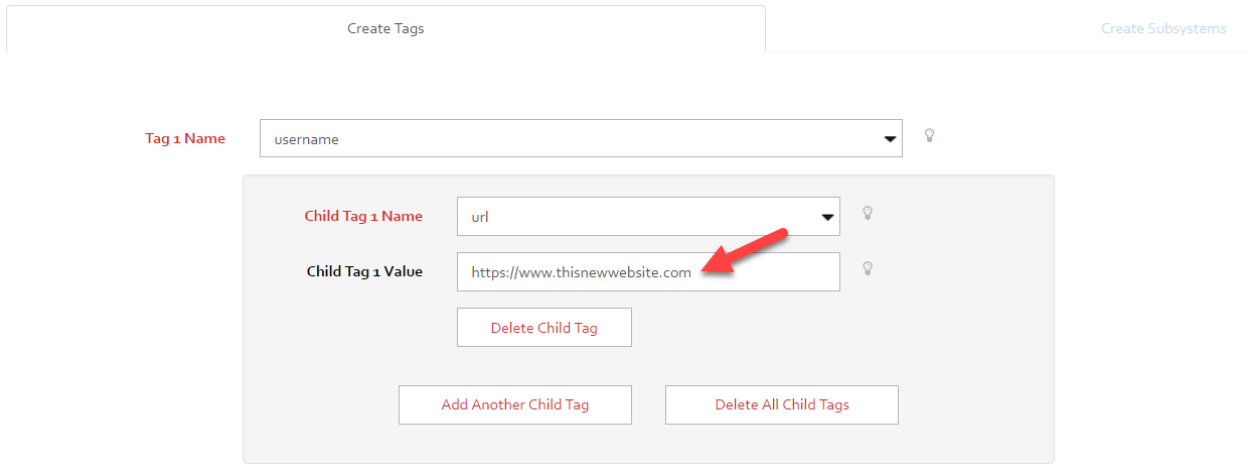
Similar to tag name, you can add a child tag name. On clicking inside the Tag Name input box, you would be able to view all the tag names that are already available to choose from. If the tag name already exists in that list, you can select it from the dropdown list; or else you can add a tag name that you want.

This is a mandatory field.



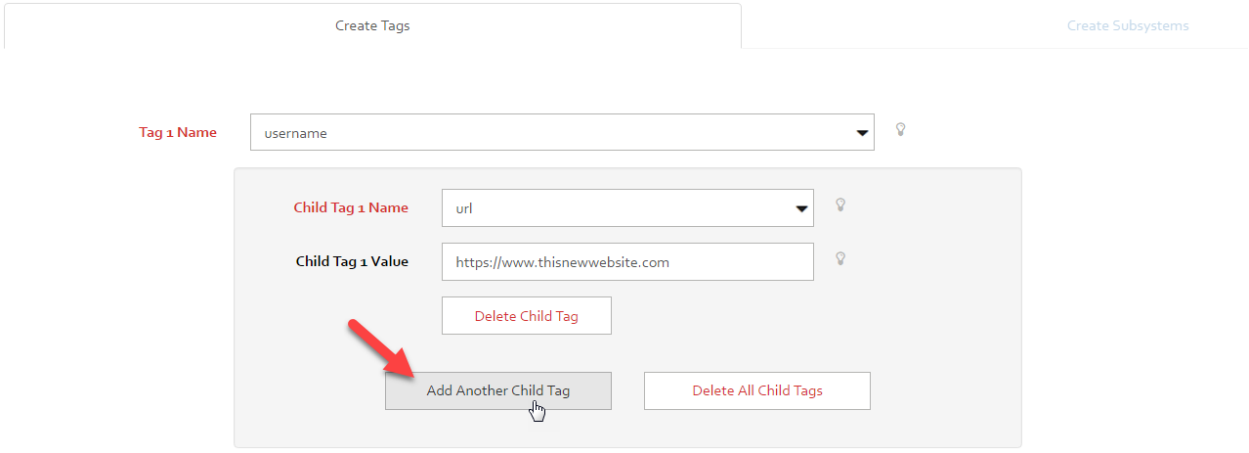
6.12. CHILD TAG VALUE

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

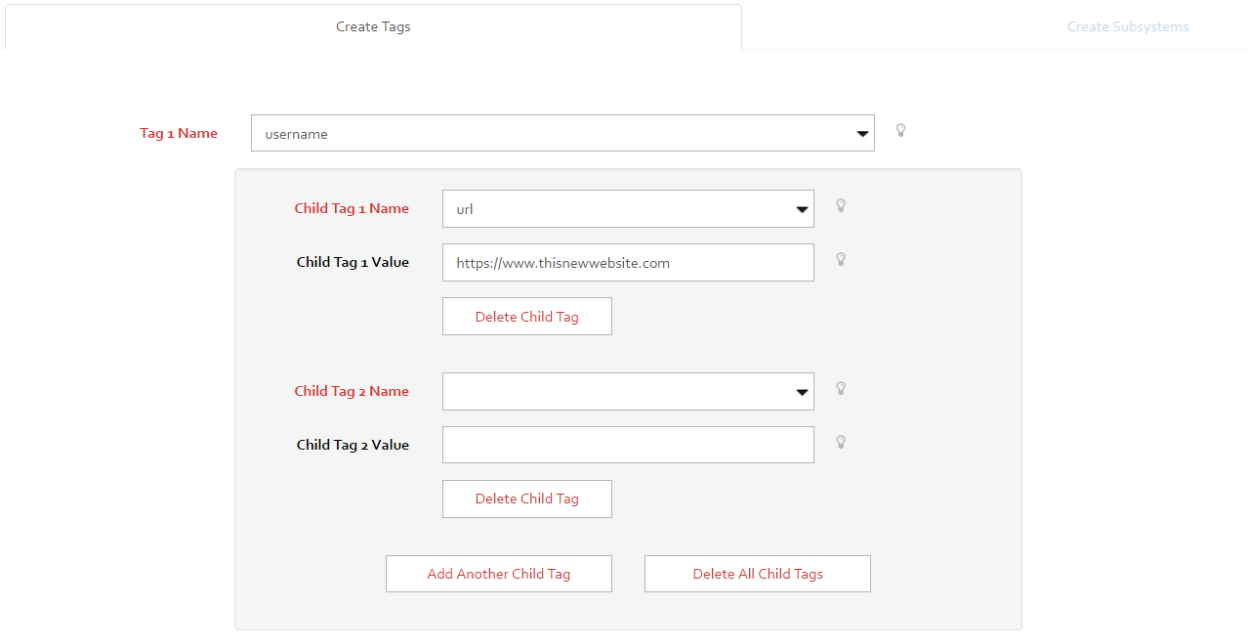


6.13. ADD ANOTHER CHILD TAG

You can add another child tag by clicking the “Add Another Child Tag” button

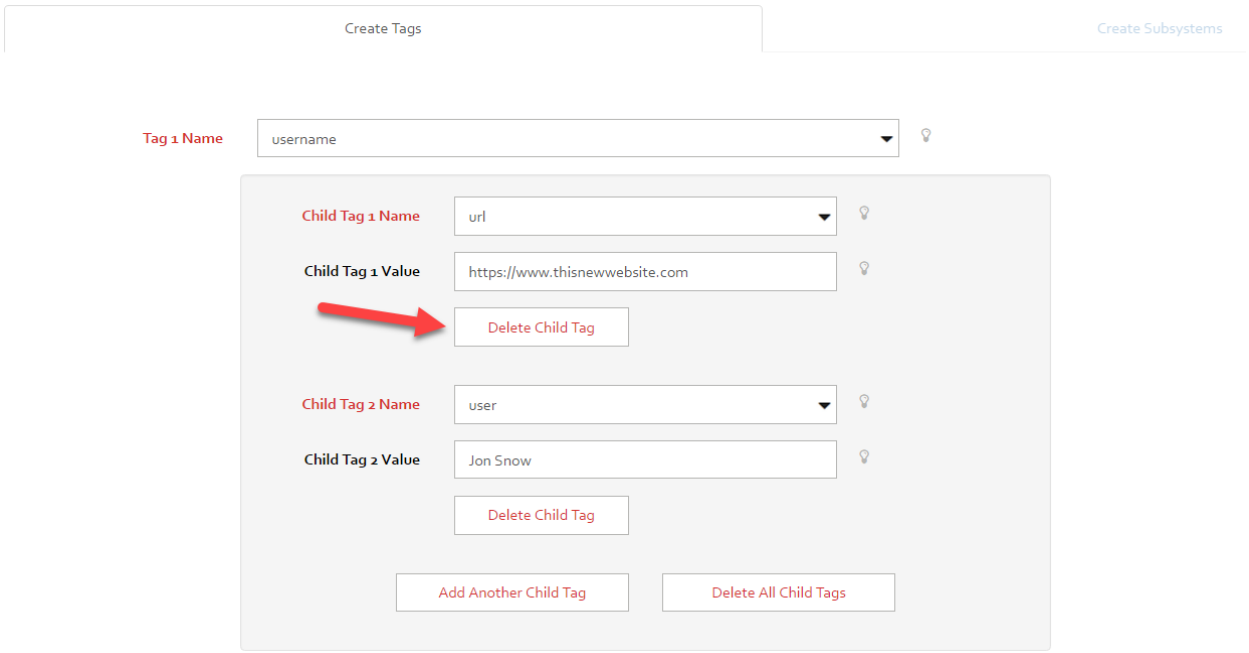


Clicking it would create another child tag filed for you.



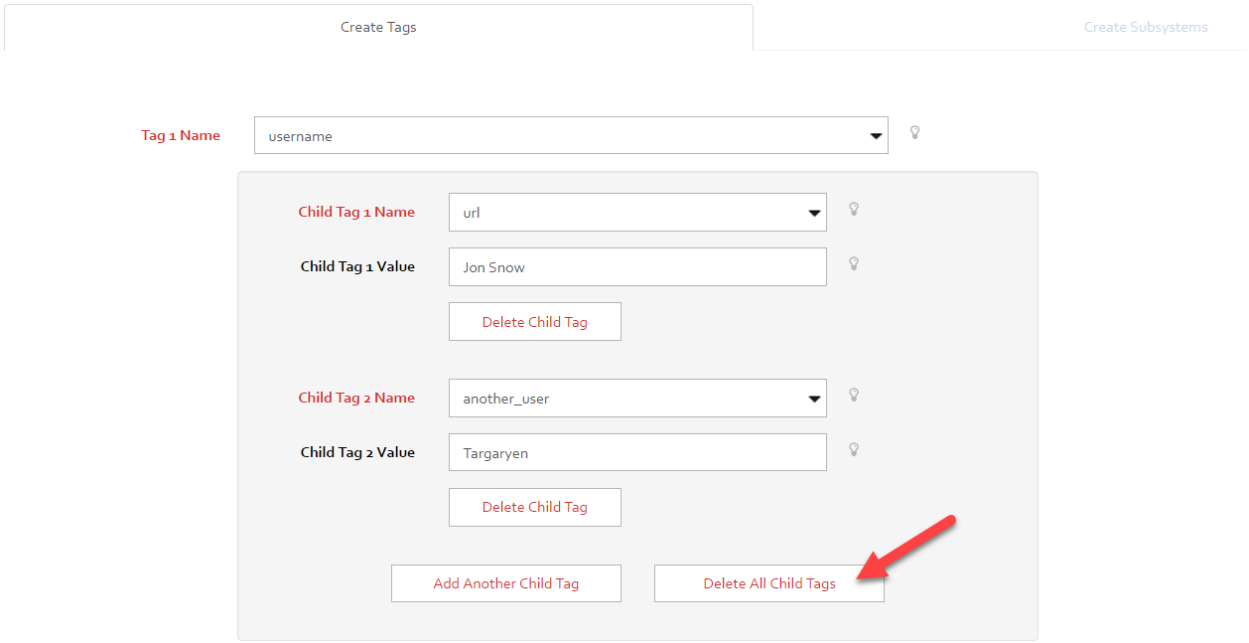
6.14. DELETE CHILD TAG

You can delete a child tag by clicking the “Delete Child Tag” button

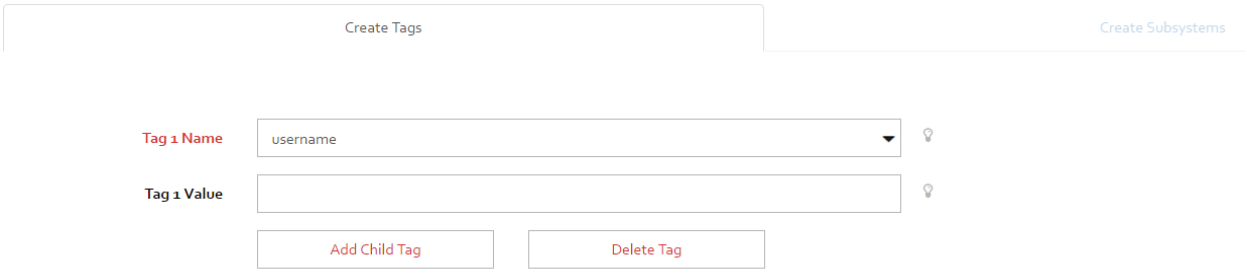


6.15. DELETE ALL CHILD TAGS

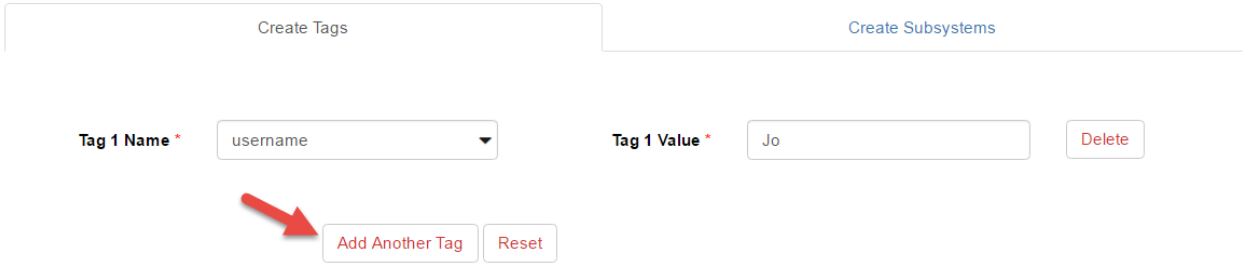
You can delete all child taga by clicking the “Delete All Child Tags” button



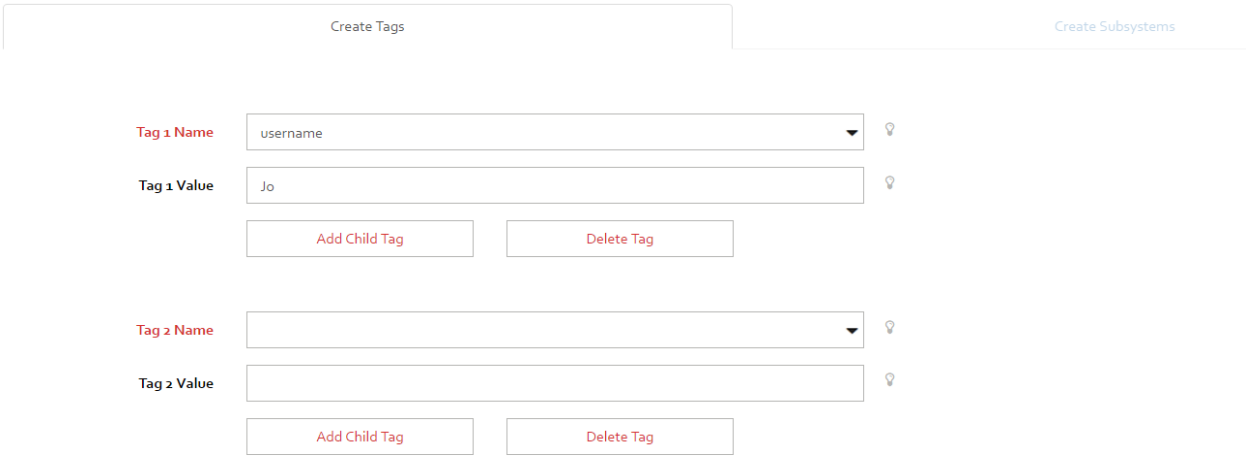
This would delete all the child tags and you would again see the Tag Value field.



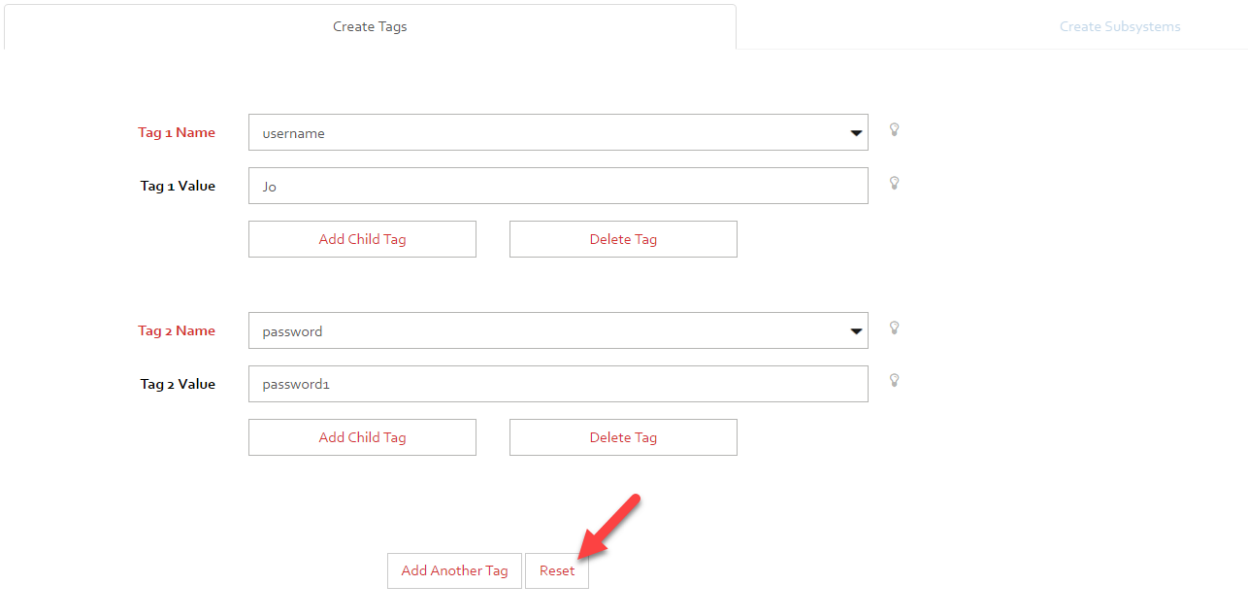
6.16. ADD ANOTHER TAG



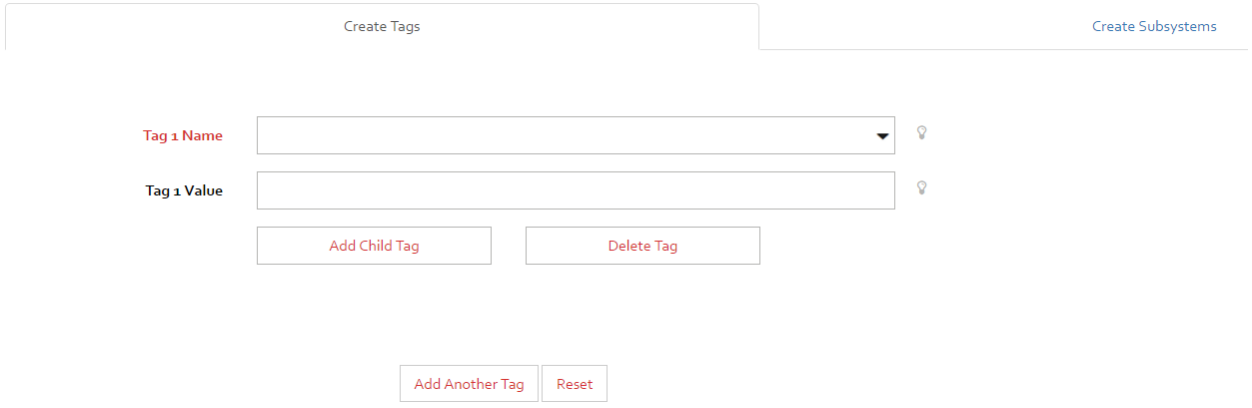
Clicking the “Add Another Tag” lets you create another tag for the same system.



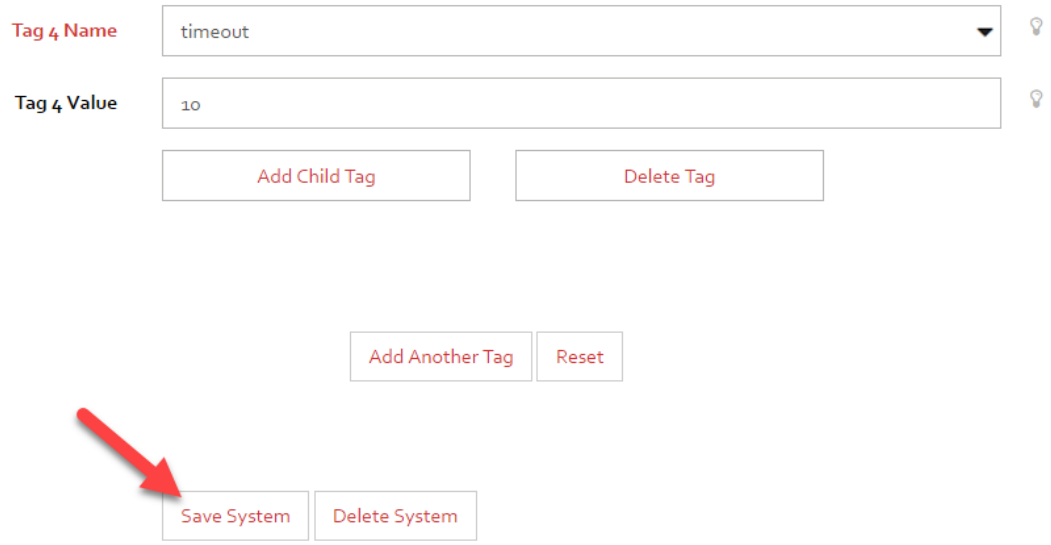
6.17. RESET



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



6.18. SAVE SYSTEM



Clicking on Save System, saves the system for you.

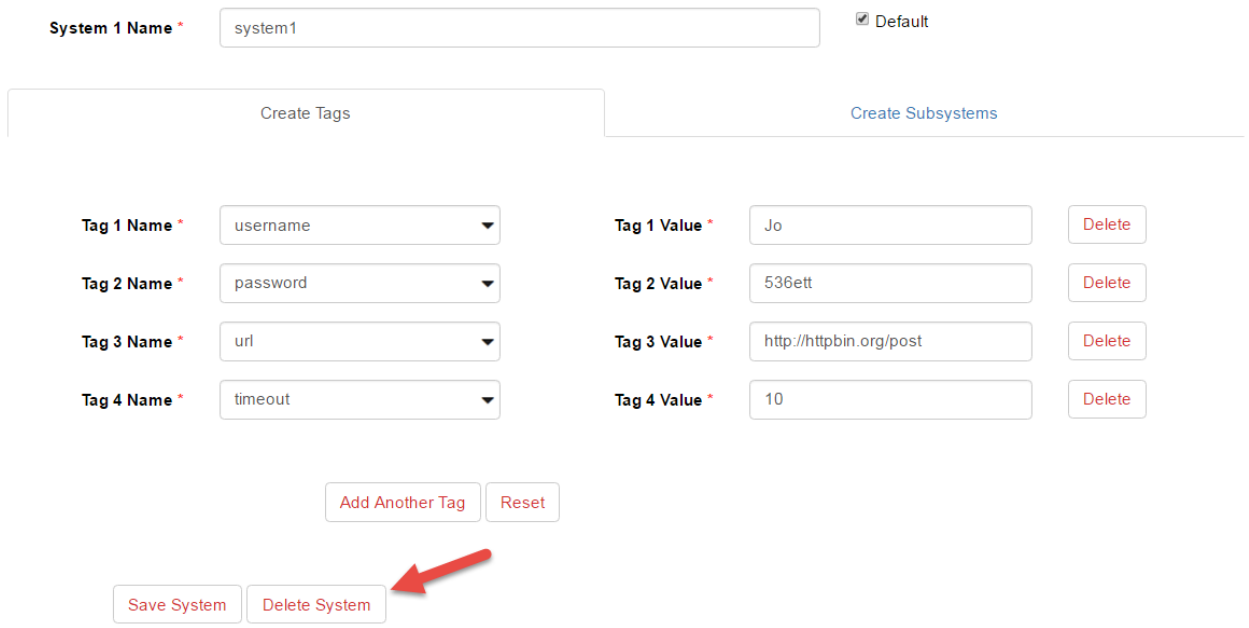
6.19. EDIT SYSTEM

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

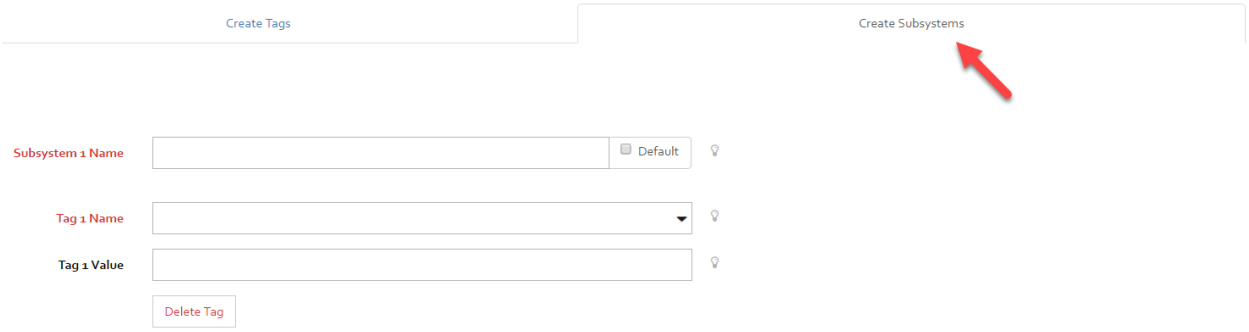
6.20. DELETE SYSTEM

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

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



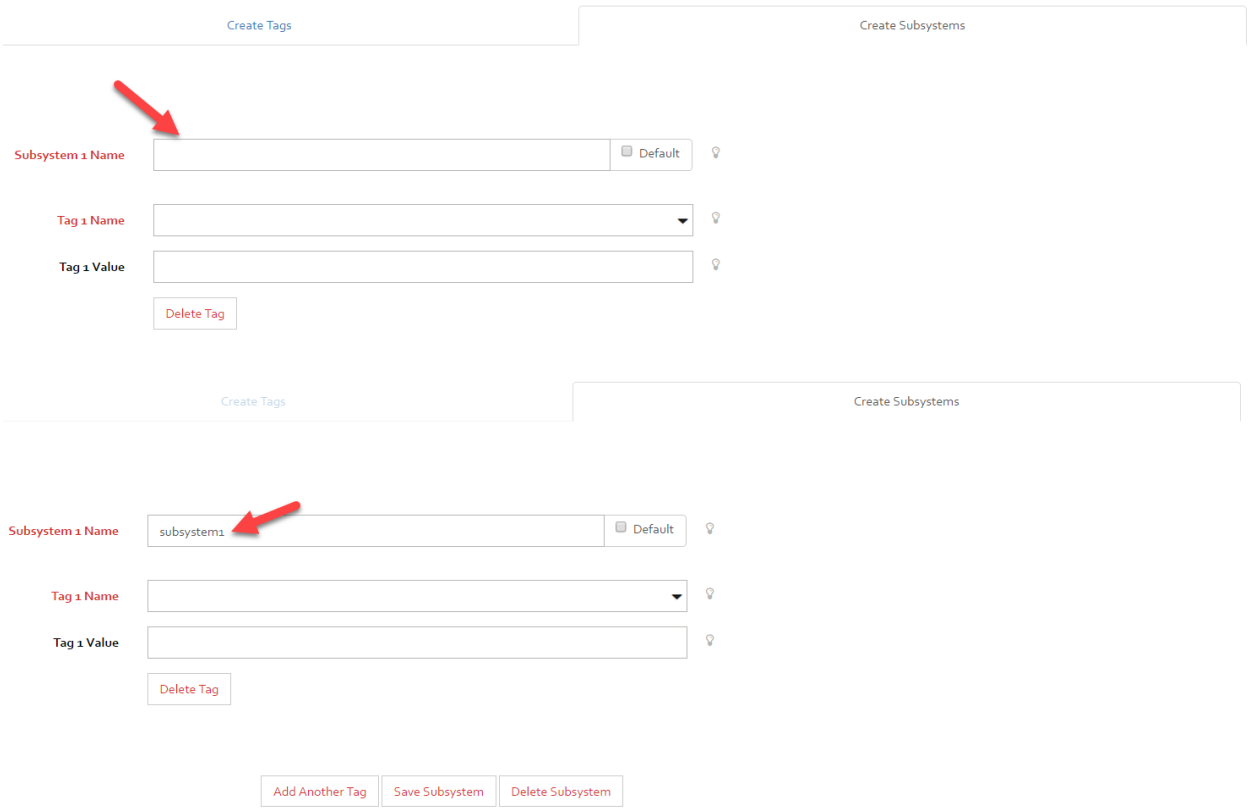
6.21. CREATE SUBSYSTEMS



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

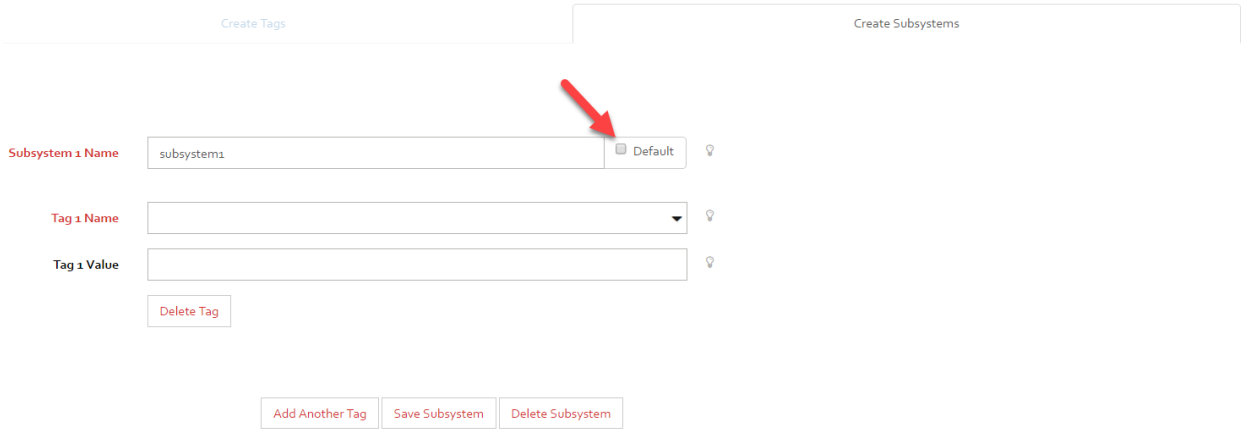
6.22. SUBSYSTEM NAME

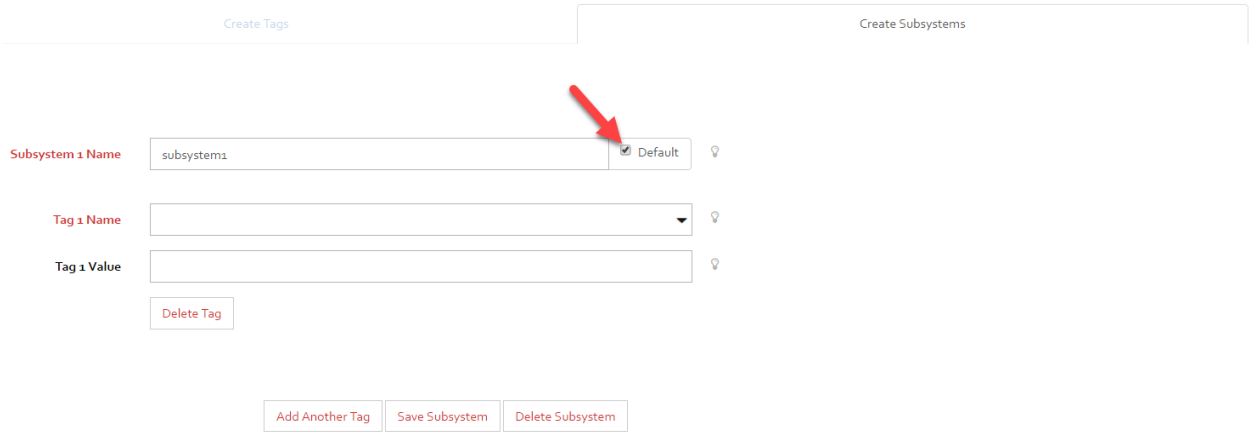
You can add a name to the subsystem in the “Subsystem Name” field. This is a mandatory field.



6.23. 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.

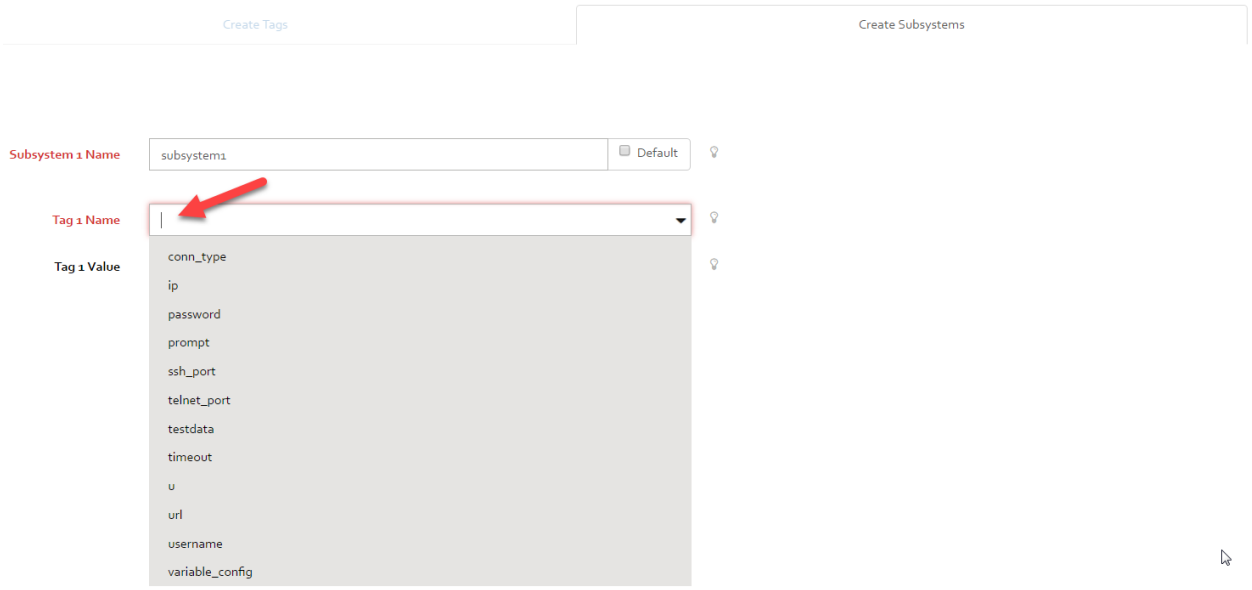




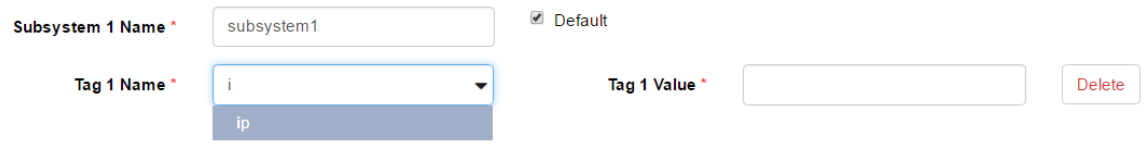
By default, the subsystem is always the default subsystem.

6.24. SUBSYSTEM 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.



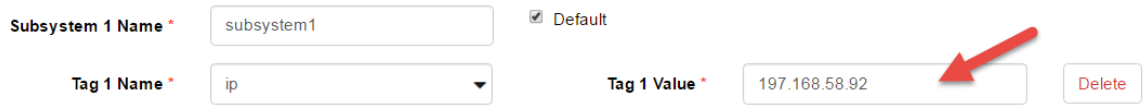
If the tag name already exists in that list, you can select it from the dropdown list; or else you can add a tag name that you want.



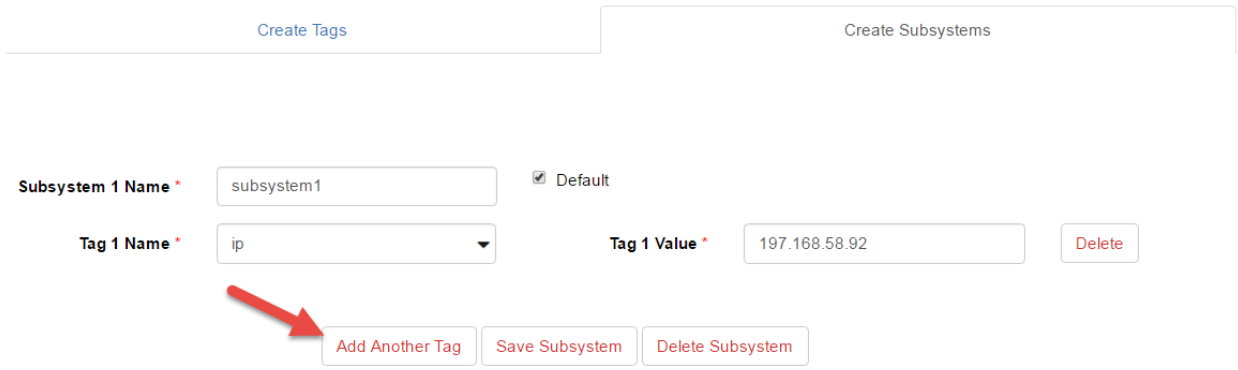
This is a mandatory field.

6.25. SUBSYSTEM TAG VALUE

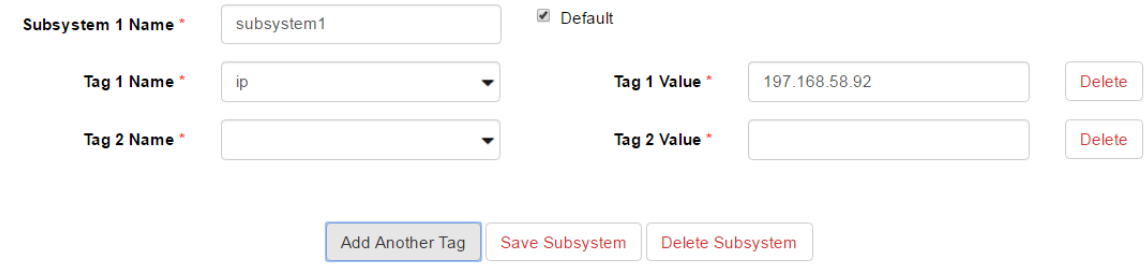
A corresponding value to the tag created above can be added here.



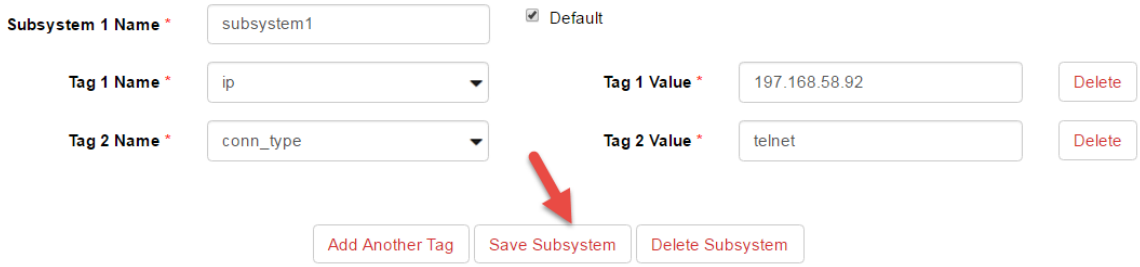
6.26. ADD ANOTHER TAG FOR SUBSYSTEM



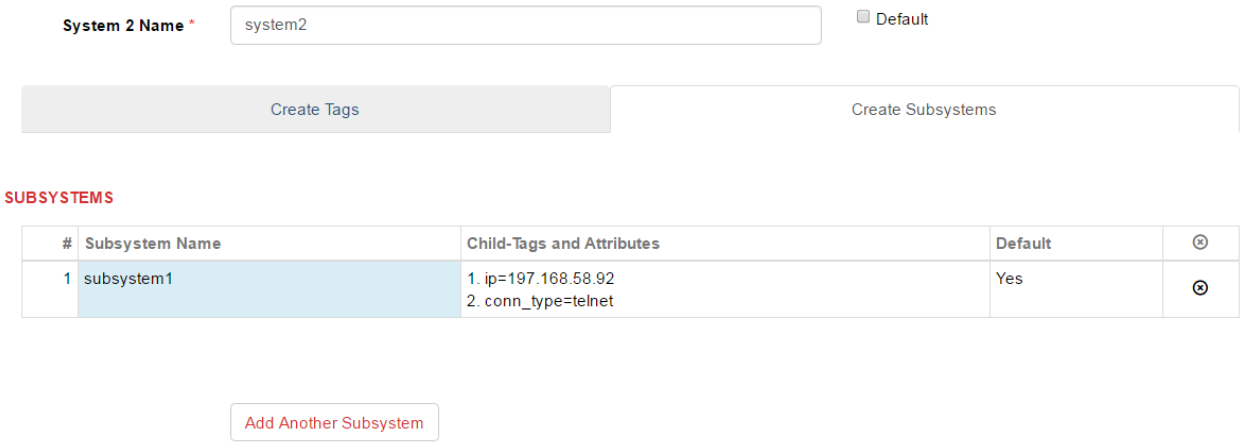
Clicking this button, lets you add another tag to the same subsystem.



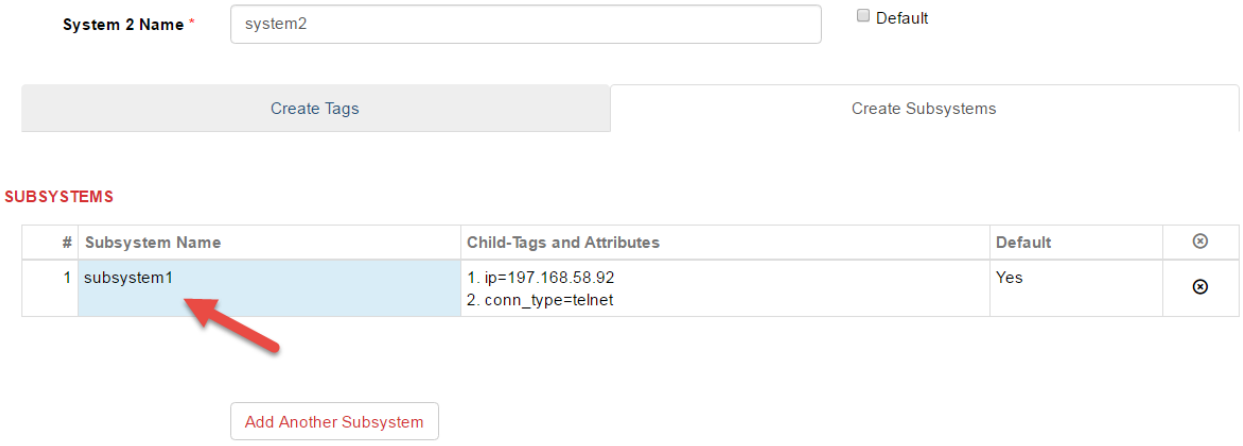
6.27. SAVE SUBSYSTEM



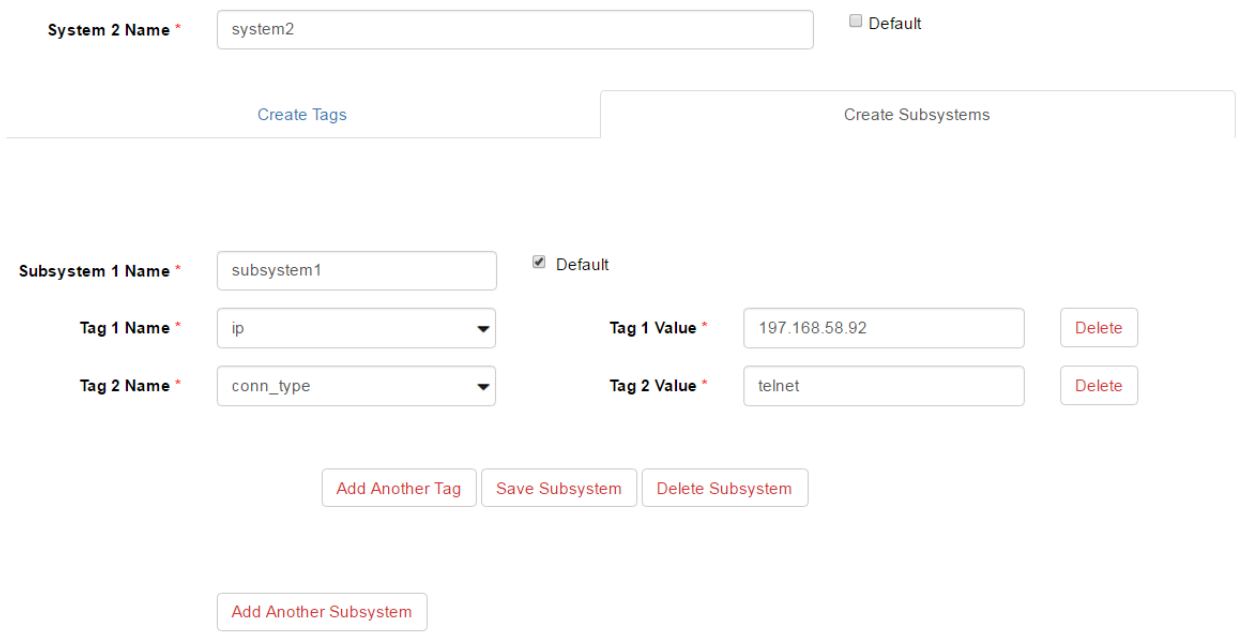
"Save Subsystem" saves the subsystem in Katana



6.28. EDIT SUBSYSTEM

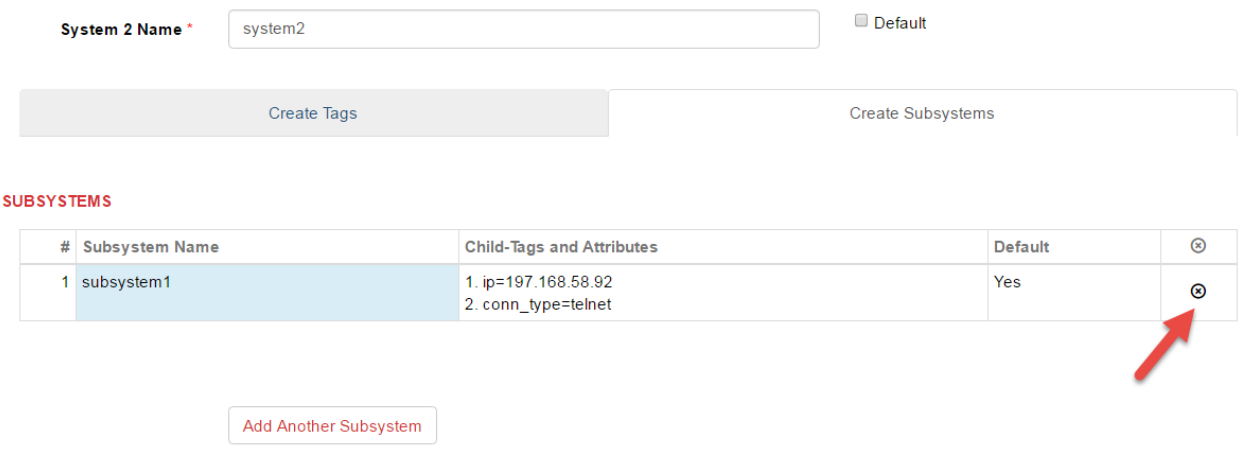


A saved subsystem can be edited by clicking on the subsystem name.

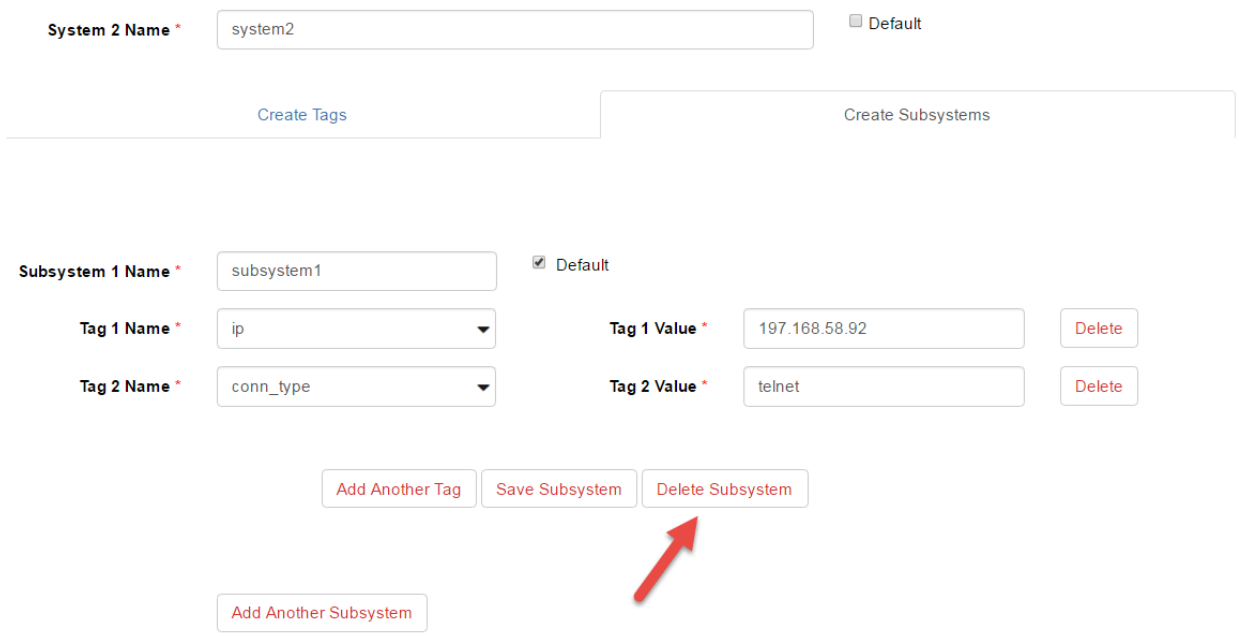


6.29. DELETE SUBSYSTEM

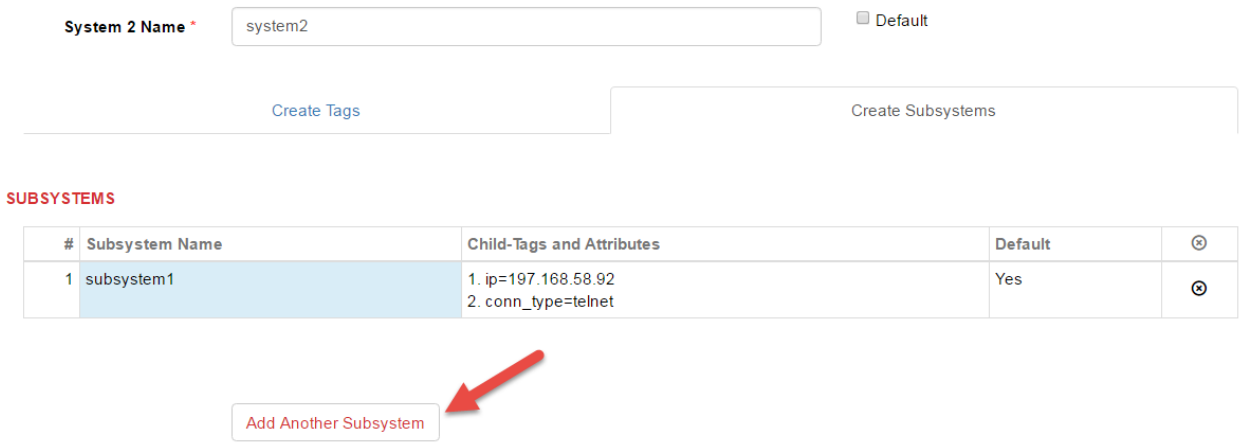
A saved subsystem can be deleted by clicking on the delete icon against it.



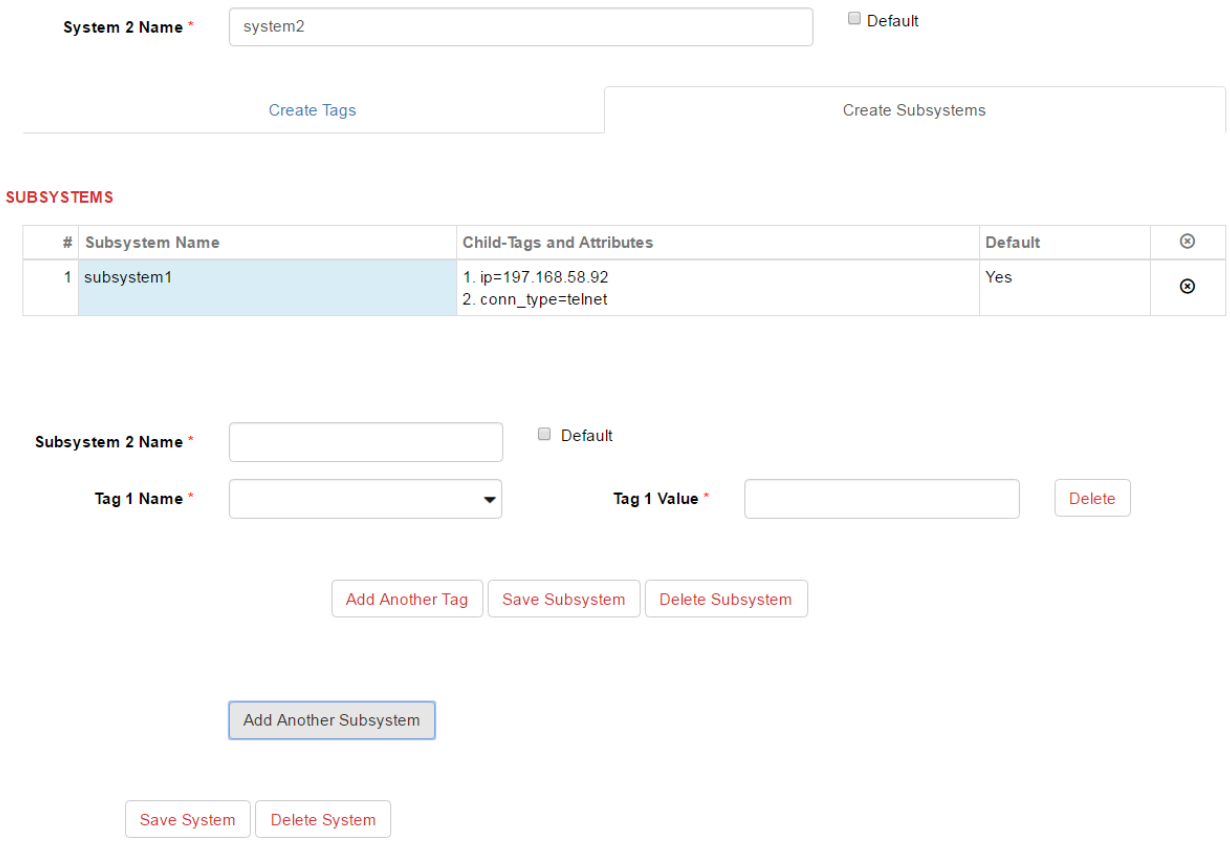
An unsaved subsystem can be deleted by clicking the “Delete subsystem” button.



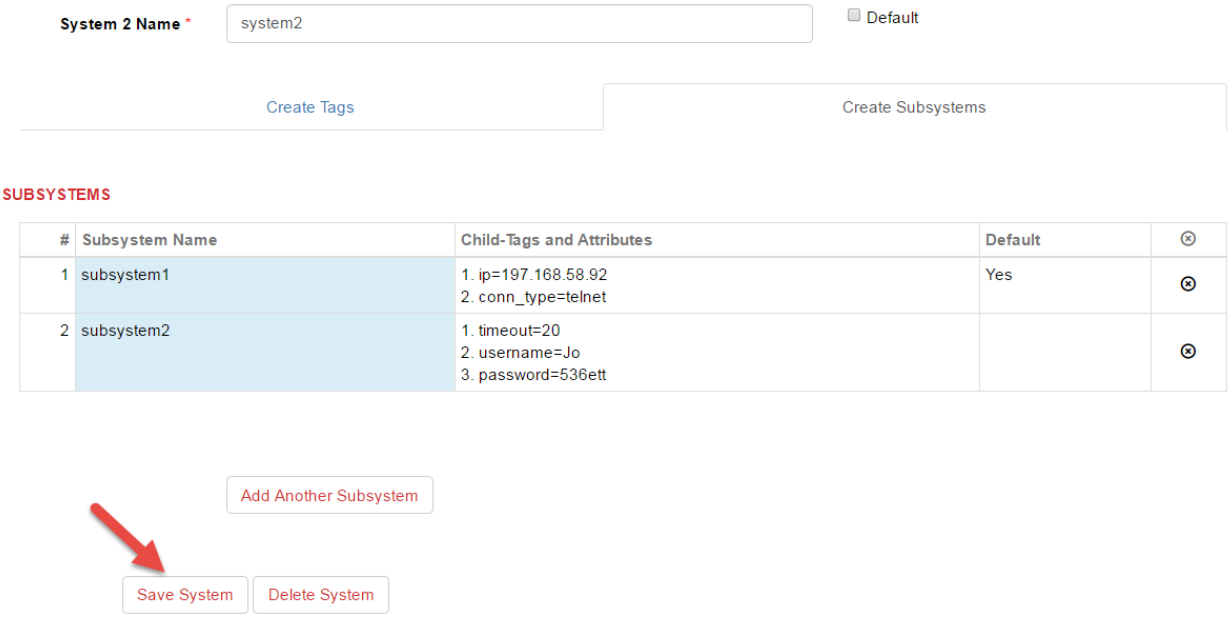
6.30. ADD ANOTHER SUBSYSTEM



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



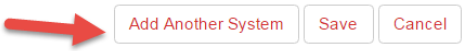
6.31. SAVE SYSTEM



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

6.32. ADD ANOTHER SYSTEM

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

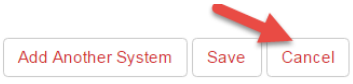


6.33. SAVE DATA FILE

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

6.34. CANCEL DATA FILE CREATION

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

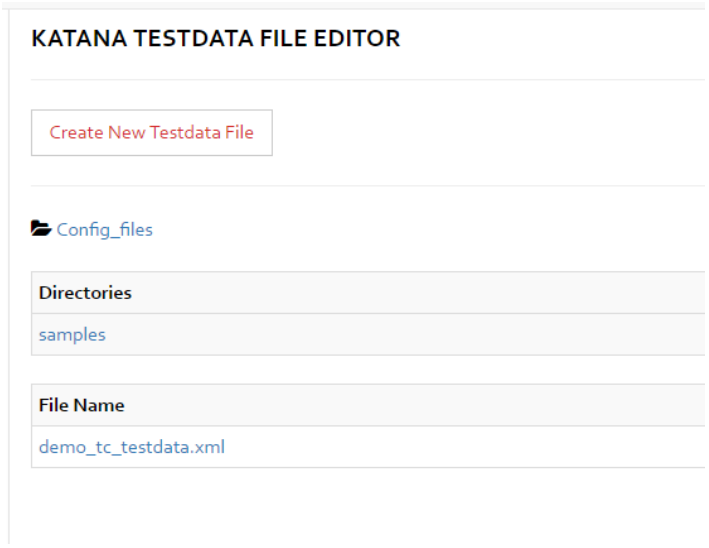


The final Data File thus created looks something like this:

```
<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. THE TESTDATA FILE

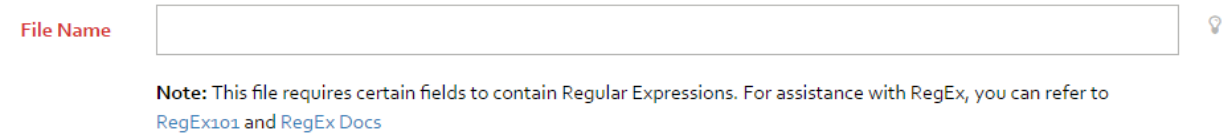
Katana allows you to create TestData Files through its user interface. You have to start up Katana and go to its TestData Files tab. There, you will see a “Create New Testdata File” button:



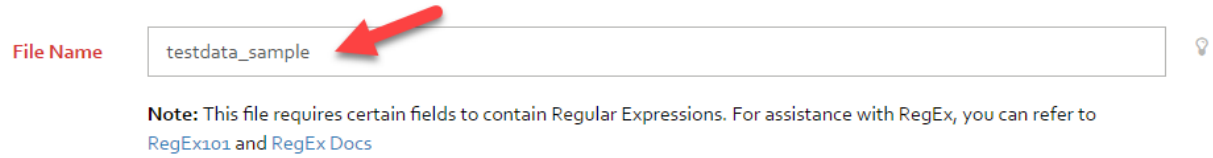
Note: The list of testdata files below the “Create New Testdata Files” button may differ as Katana would show the input data files saved in your Config_files directory.

On clicking that button, you will be directed to a new page where you will see the following fields.

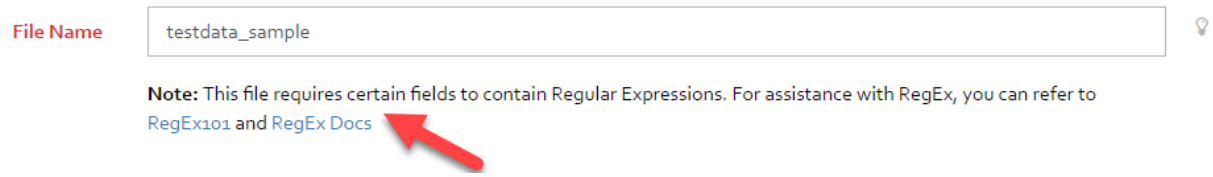
7.1. TESTDATA FILE NAME



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



A note exists below this field that redirects you to a Regular Expression page – this note is here for reference as this form has some fields that may require a regular expression as an input.



7.2. GLOBAL SECTION

This is the global section of the testdata file. This section is the 'default' section, that is, if some information is not given in the testdata block, that information would be read from this global section.

7.3. GLOBAL SYS

This is the system and/or subsystem on which the command should be executed. Connection to the system/subsystem should have been established earlier.

1. system = provide the system name directly

   Example: "NE1", "server1"

2. system + subsystem = provide system+subsystem combination, only a single subsystem is supported

   Example: "NE1[cli]", sever1["interface1"]

3. subsystem only = to refer only a subsystem provide the subsystem name enclosed by square brackets, In this case the system name provided in the testcase keyword will be used.

Example: [cli], [dip], [interface1]

7.4. GLOBAL SESSION

This is session name of the system or subsystem to connect to.

7.5. GLOBAL START

This is starting prompt of the command. The default is ".*". Using the default ignores the check for start prompts. Variable substitution is supported for this parameter.

7.6. GLOBAL END

This is ending prompt of the command. The command is considered to have completed successfully if the end prompt is found in the command's response. Variable substitution is supported for this parameter.

7.7. GLOBAL TIMEOUT

This is the time to wait to receive the end prompt. If a command times out, then the command status will be marked as error. After timeout, Warrior will wait for an additional 60 seconds to receive the end prompt, this is to provide an extra buffer to check if it is an intermittent delay or a long delay. Irrespective of whether or not the end prompt is received during this extra time the command will be marked as error. The default is 60 seconds.

7.8. GLOBAL SLEEP

This is the time to wait in seconds after completion of a command.

7.9. GLOBAL VERIFY

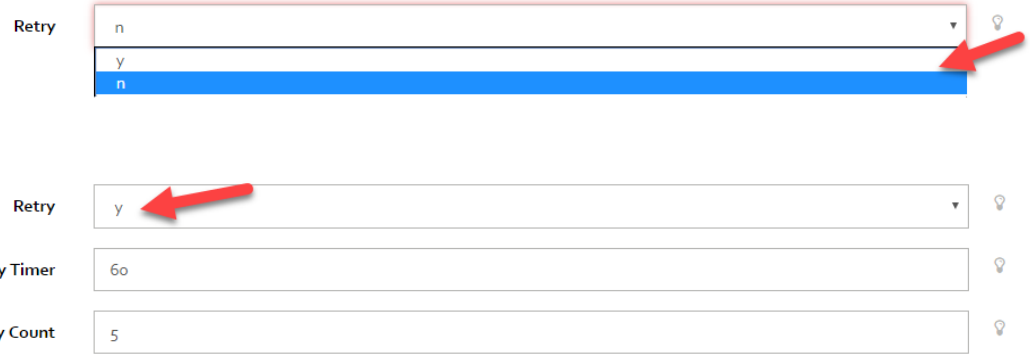
This field takes in the verification tags created in the verification section.

If multiple verification are required for a command, provide tag names separated by commas. Example: "v1,v2,v3"

7.10. GLOBAL RETRY, RETRY TIME, AND RETRY COUNT

When retry is set to “y”(yes), Warrior will retry sending a command if the command fails.

By default, retry is set to “n” (no). You can click on the drop down and change its value to “y” to reveal the fields for Retry Timer and Retry Count.



Retry Timer is the time interval in seconds between subsequent re-trails.

Retry Count is number of attempts to keep on re-sending the command. If during one of the attempts the command passes, re trials will be passed.

7.11. GLOBAL RETRY ONMATCH

This can be a text or a regular expression. If provided and Retry is set to “y”, then re-trials will be attempted only when the response of the command has the provided text/regular expression.

7.12. GLOBAL RESPONSE REQUIRED

This saves the response of the command in the framework's data repository. This feature is always enabled. You can turn it off by clicking on the dropdown and setting its value to "n"

7.13. GLOBAL RESPONSE PATTERN REQUIRED

This saves only a particular text or regular expression from the response of the command into the framework’s datarepository.

7.14. GLOBAL RESPONSE REFERENCE

You can provide a reference text for the response of this command. The responses will be saved in the framework's data repository using this text as the key. If you do not provide response reference, then, framework automatically assigns this value based on the position of the command in the testdata section, that is, if the command is the second command in the testdata then resp_ref="2" will be used and so on.

7.15. GLOBAL MONITOR

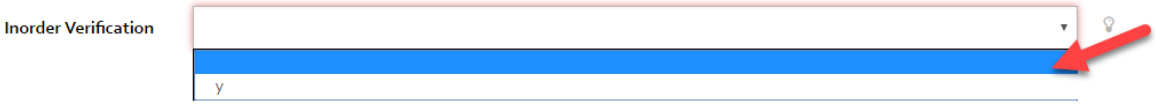
Responses from the system names obtained from this parameter are printed out on the console.

Example: comma separated system names - NE1[cli].session, NE2, [dip] - where NE1, NE2 are the system names, [cli], [dip] are the subsystem names and .session is the session name.

7.16. GLOBAL IN-ORDER VERIFICATION

When set to “y” (yes), in-order verification of response will happen.

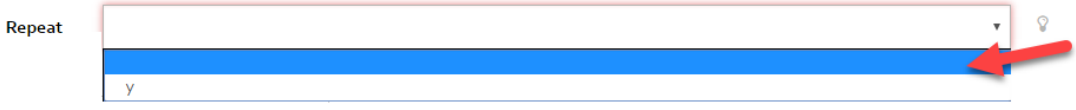
This feature is always disabled. You can turn it on selecting "y" from the dropdown. If enabled, command passes only when the received response order matches with the order given in the 'verify' tag



7.17. GLOBAL REPEAT

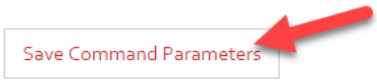
when iter_type=‘per_td_block’ with repeat='y', corresponding command in testdata block will be executed in each iteration.

This feature is always disabled, user can turn it on by selecting "y" from the dropdown and this tag is applicable only for the testdata commands without any iteration pattern in it.



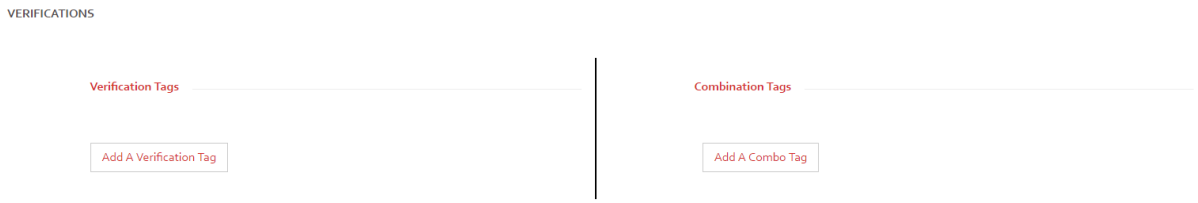
7.18. SAVE GLOBAL COMMAND PARAMETERS

You can save the global command parameters by clicking on this button:



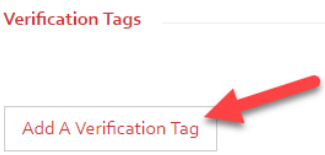
7.19. GLOBAL VERIFICATIONS

These are the global level verification and combination tags

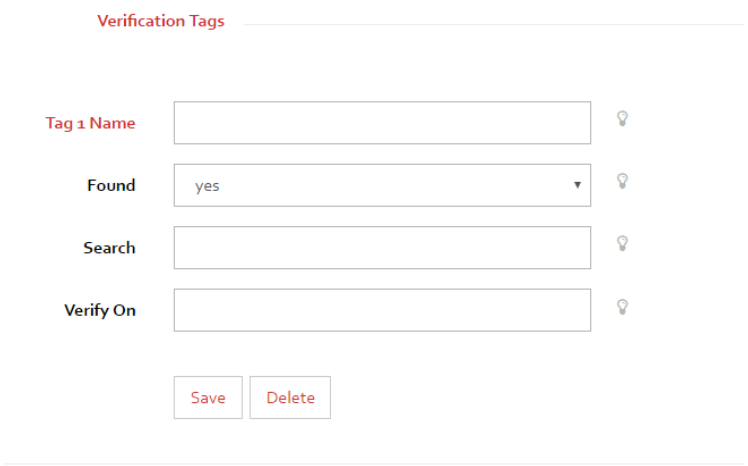


7.20. GLOBAL VERIFICATION TAGS

You can add a verification tag by clicking on the “Add a Verification Tag” button



It would display the block below:

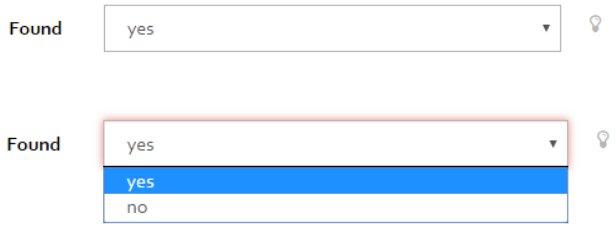


7.21. GLOBAL VERIFICATION TAG NAME

Enter the tag name here. This is a mandatory field.

7.22. GLOBAL VERIFICATION FOUND

“Yes” will search for the presence of the search string the command response. “No” will search for the absence of the search string the command response



7.23. GLOBAL VERIFICATION SEARCH

This is the search string that will be searched for in the command response

7.24. GLOBAL VERIFICATION VERIFY ON

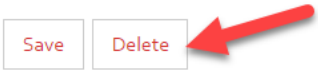
This would contain the systems on which the verification would be performed.

7.25. SAVING GLOBAL VERIFICATION

Save the verifications by clicking this button:

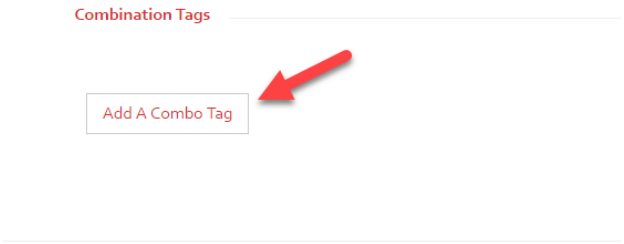
7.26. DELETING GLOBAL VERIFICATION

Delete the verifications by clicking this button:

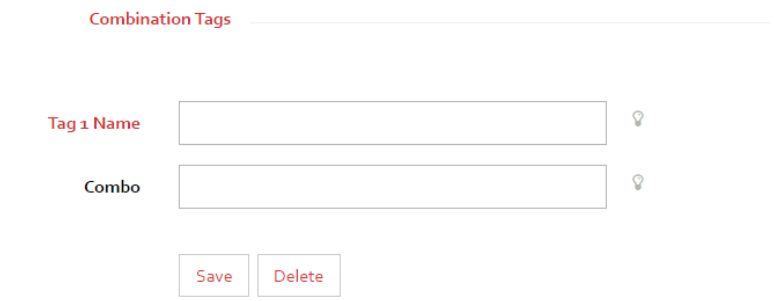


7.27. ADD GLOBAL COMBINATION TAG

You can add a combination tag by clicking this button.



This would reveal the block below:



7.28. ADD GLOBAL COMBINATION TAG NAME

You can add the tag name here. This is a mandatory field.

7.29. ADD GLOBAL COMBINATION TAG VALUE

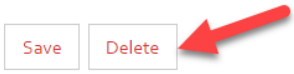
Add comma separated verification tags here.

7.30. SAVE GLOBAL COMBINATION TAG

Save the combination tags by clicking this button

7.31. DELETE GLOBAL COMBINATION TAG

Delete the combination tags by clicking this button

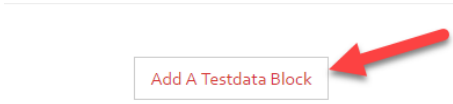


7.32. TESTDATA BLOCKS

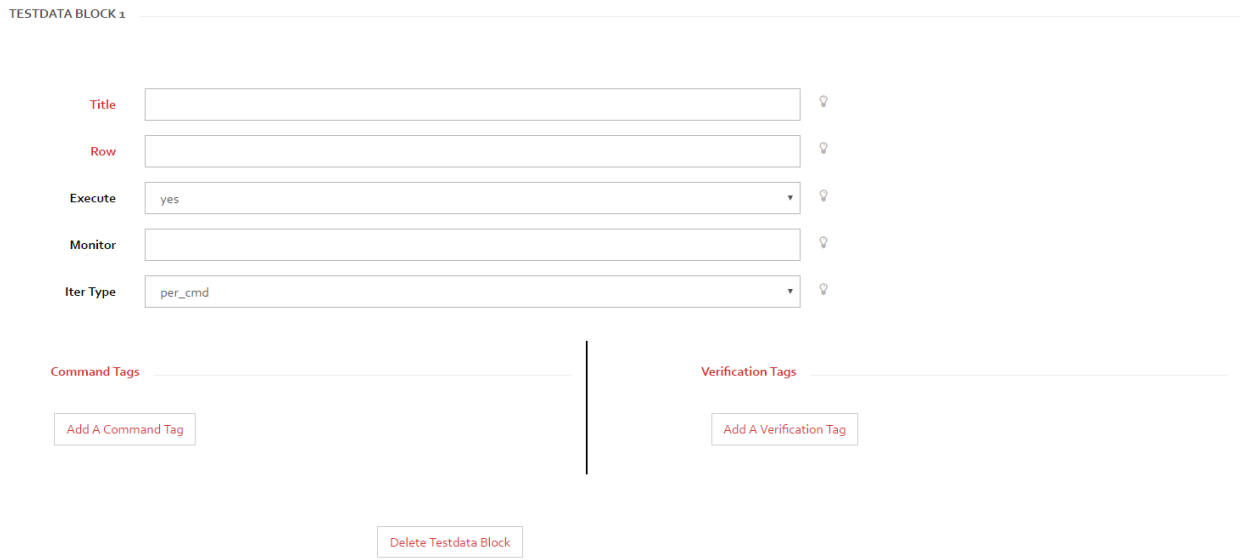
The testdata blocks start from here:

7.33. ADD A TESTDATA BLOCK

You can add a testdata block by clicking on this button:



A form like below will be displayed:



7.34. TESTDATA TITLE

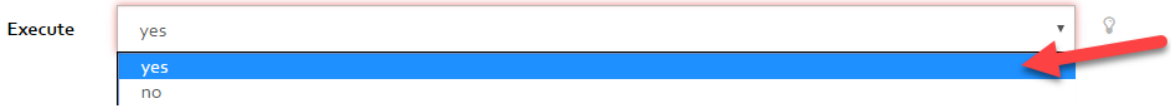
This would be the title of the testdata block

7.35. TESTDATA ROW NUMBER

This would be the row number of the testdata block

7.36. TESTDATA EXECUTE

This would indicate if this testdata block should be executed. The default is “yes”.

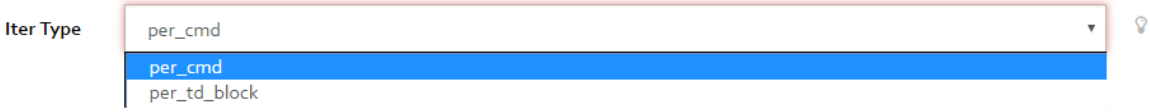


7.37. TESTDATA MONITOR

This is the testdata block level monitor attribute.

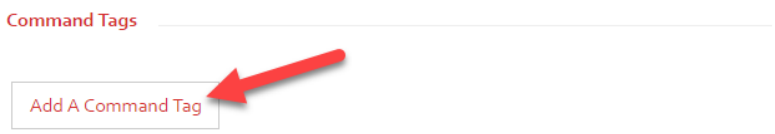
7.38. TESTDATA ITERATION TYPE

This is the testdata iteration type. The default is “per_cmd”.

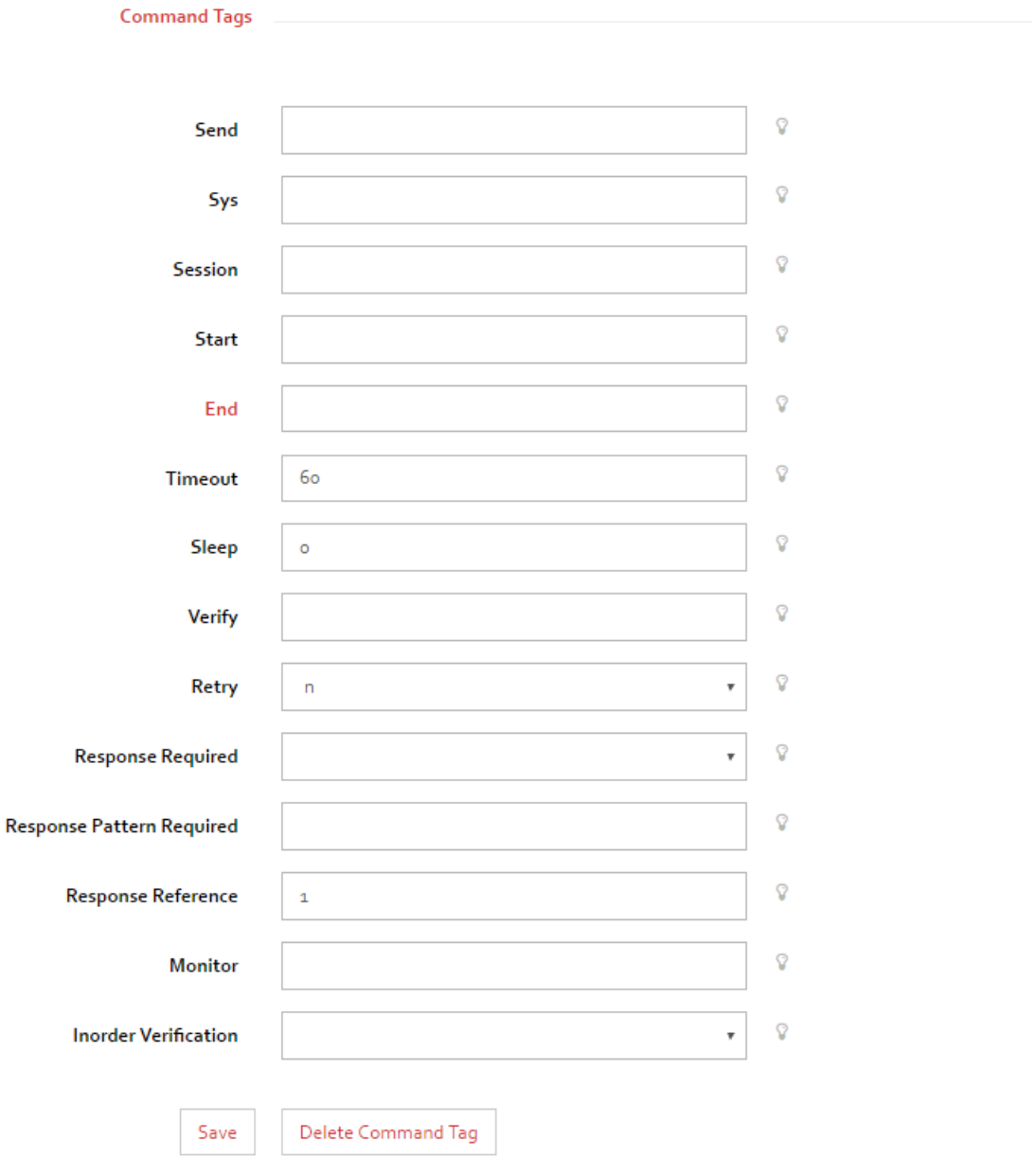


7.39. ADD TESTDATA COMMAND

Add a testdata level command by clicking on this button:



A form like below will be displayed:



7.40. TESTDATA COMMAND SEND

This is the command for the testdata block.

7.41. OTHER TESTDATA COMMAND ATTRIBUTES

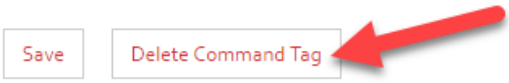
Please refer to the global command parameters section for the details.

7.42. SAVE TESTDATA COMMAND

Save the testdata command block

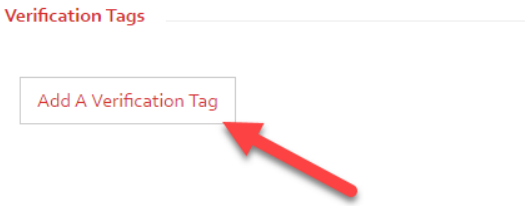
7.43. DELETE TESTDATA COMMAND

Delete the testdata command block

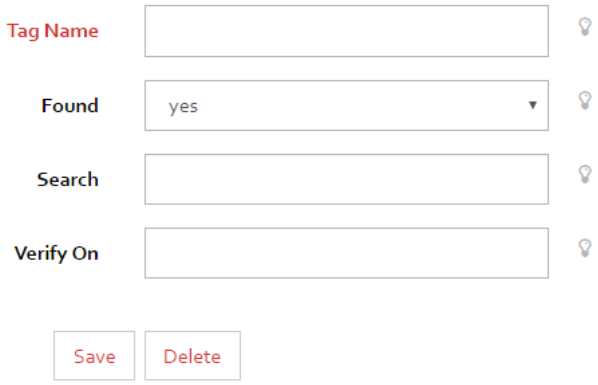


7.44. ADD TESTDATA VERIFICATION TAG

You can add testdata level verification tags by clicking on this button:



A form like below would be displayed.



7.45. TESTDATA VERIFICATION TAG DETAILS

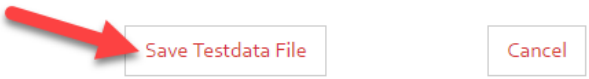
Refer to global verification tags for details on this section.

7.46. DELETE TESTDATA BLOCK

You can delete the testdata block here:

7.47. SAVE TESTDATA FILE

You can save the testdata file here:

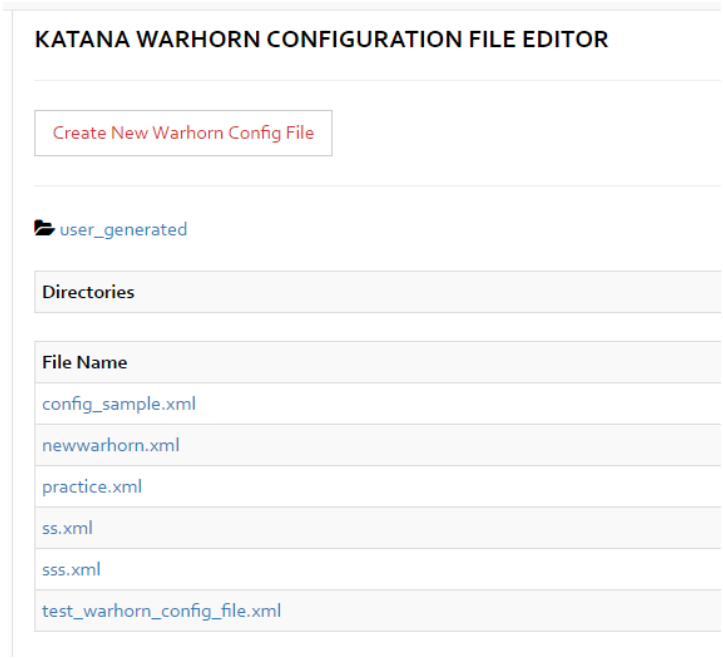


7.48. CANCEL TESTDATA FILE CREATION

You can cancel the testdata file creation here:

8. THE WARHORN CONFIGURATION FILE

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

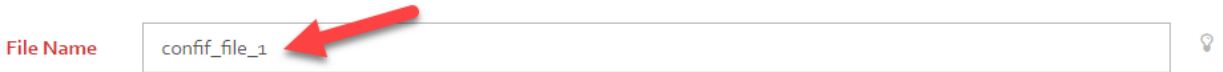


Note: The list of warhorn config files below the “Create New Warhorn Config Files” button may differ as Katana would show the input data files saved in your Warhorn Config Files directory.

On clicking that button, you will be directed to a new page where you will see the following fields.

8.1. WARHORN CONFIG FILE NAME

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



8.2. DEPENDENCIES

These are the dependencies recommended by Warrior for you to install in the environment where you will execute Warrior. The dependencies which are already checked are the one which would get installed when you run warhorn. You can check or uncheck the dependencies to suit your needs.

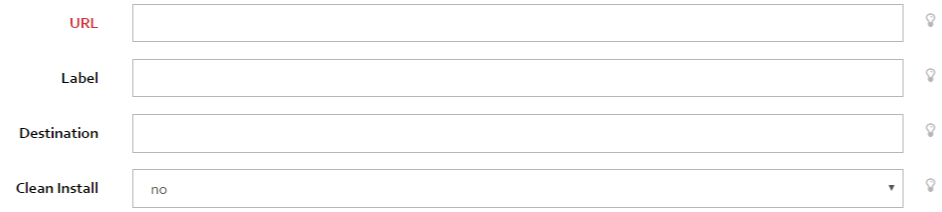
DEPENDENCIES

#	Dependency	Version That Will Be Installed	Install?
1	jira	1.0.3	☐
2	lxml	3.5	☑
3	ncclient	0.4.6	☐
4	paramiko	1.16.0	☑
5	pexpect	3.1	☑
6	pysnmp	4.3.2	☑
7	requests	2.9.1	☑
8	selenium	2.48.0	☑
9	xlrd	1.0.0	☑
10	cloudshell-automation-api	7.1.0.34	☑

8.3. WARRIOR INSTALLATION DETAILS

This section contains details about Warrior.

WARRIOR INSTALLATION DETAILS



8.4. WARRIOR URL

Type the URL of the Warrior repository here.

8.5. WARRIOR LABEL

Type in the branch name, commit-id or tag that you want you Warrior to be at

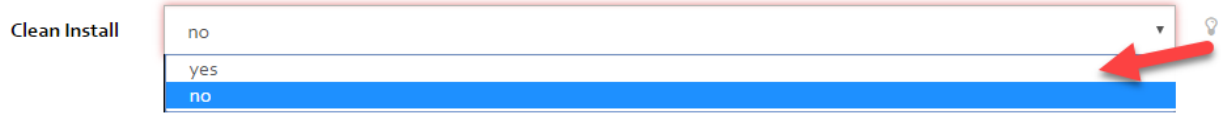
8.6. WARRIOR DESTINATION

Type in a directory path where you want Warrior to get installed

8.7. WARRIOR CLEAN INSTALL

Clean install indicates if you want to delete existing Warrior and replace it with a brand new one. By default, this is always set to “no”

You can click on the dropdown and select “yes”.



8.8. KATANA INSTALLATION DETAILS

This section contains details about Katana

8.9. KATANA URL

Type the URL of the Katana repository here.

8.10. KATANA LABEL

Type in the branch name, commit-id or tag that you want you Katana to be at.

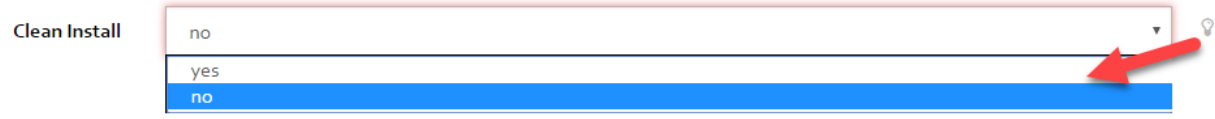
8.11. KATANA DESTINATION

Type in a directory path where you want Katana to get installed

8.12. KATANA CLEAN INSTALL

Clean install indicates if you want to delete existing Katana and replace it with a brand new one. By default, this is always set to “no”

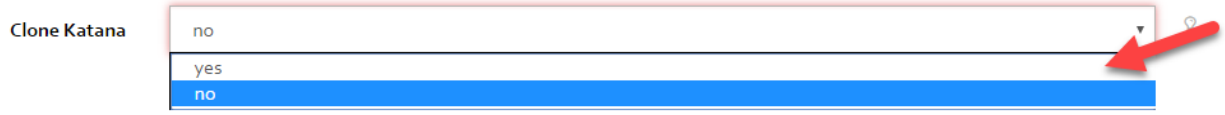
You can click on the dropdown and select “yes”.



8.13. KATANA CLONE

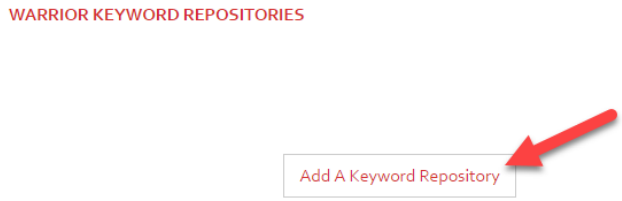
Clone indicates if you don’t want to clone Katana at all. By default, this is always set to “no”

You can click on the dropdown and select “yes”.

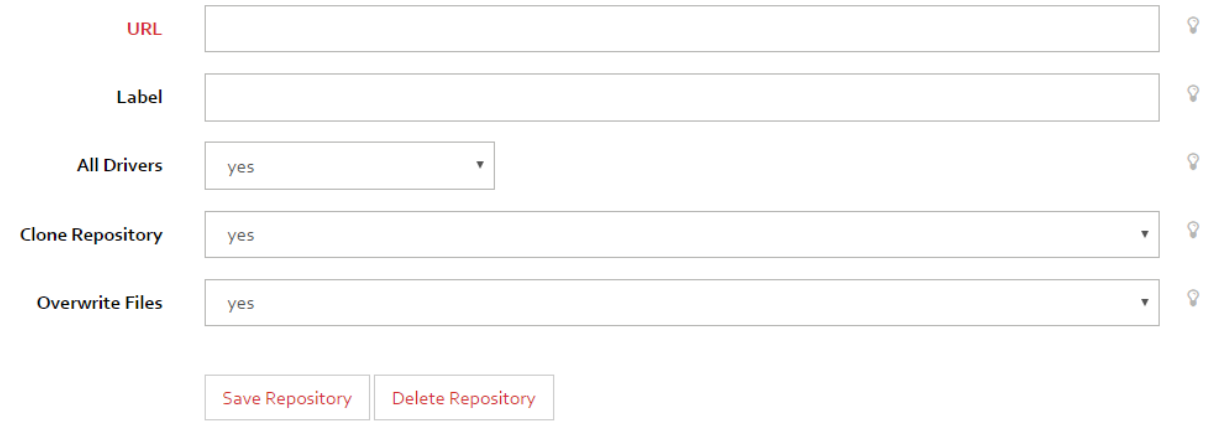


8.14. ADD A KEYWORD REPOSITORY

You can add a Keyword repository here.



On clicking “Add a keyword repository”, this block will be displayed:



8.15. KEYWORD REPOSITORY URL

This is the keyword repository URL

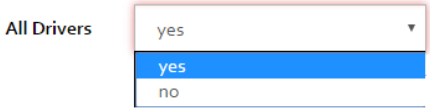
8.16. KEYWORD REPOSITORY LABEL

Type in the branch name, commit-id or tag that you want the keyword repository to be at.

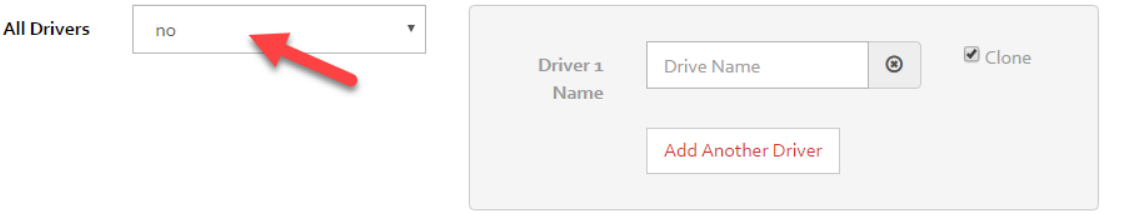
8.17. KEYWORD REPOSITORY ALL DRIVERS

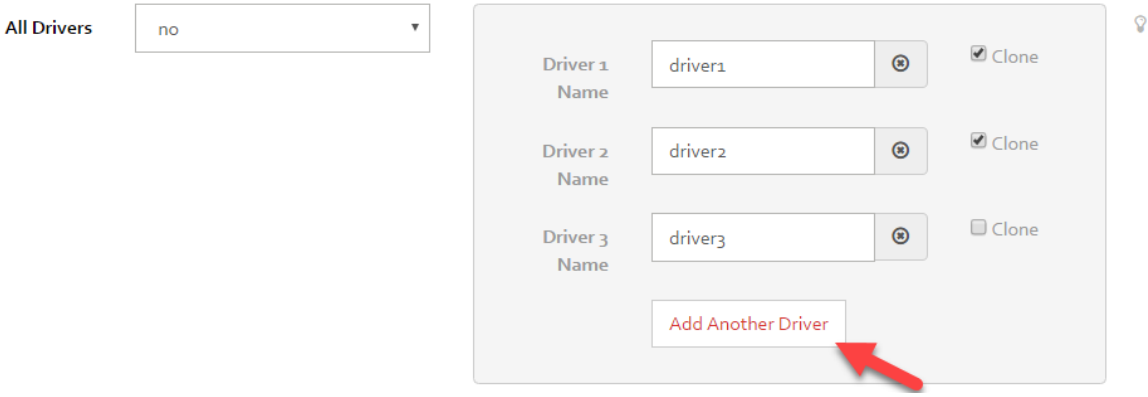
You can either clone all the drivers in the repository.

Or you can choose which drivers to clone by selecting “no” from the dropdown:

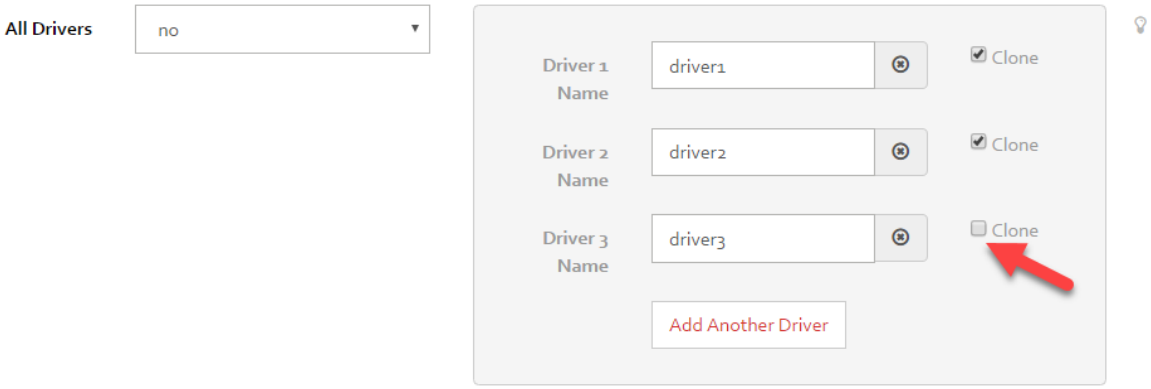


You can then add the driver names that you want to clone.

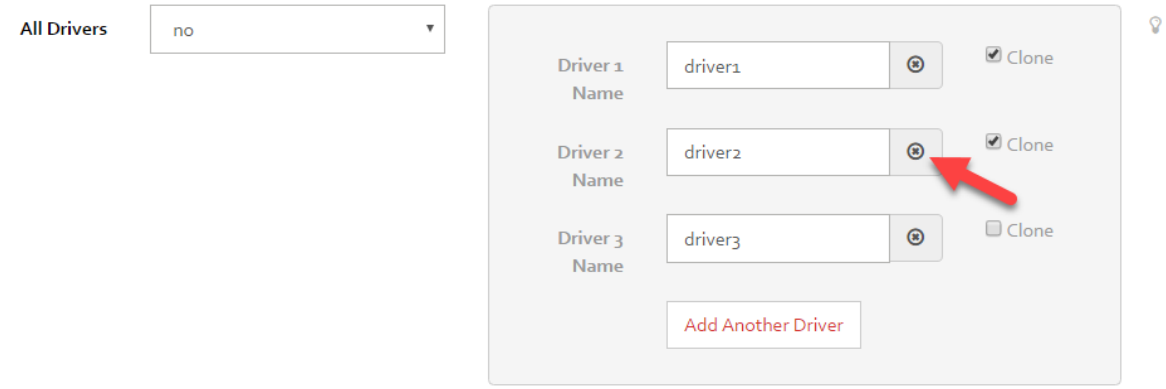


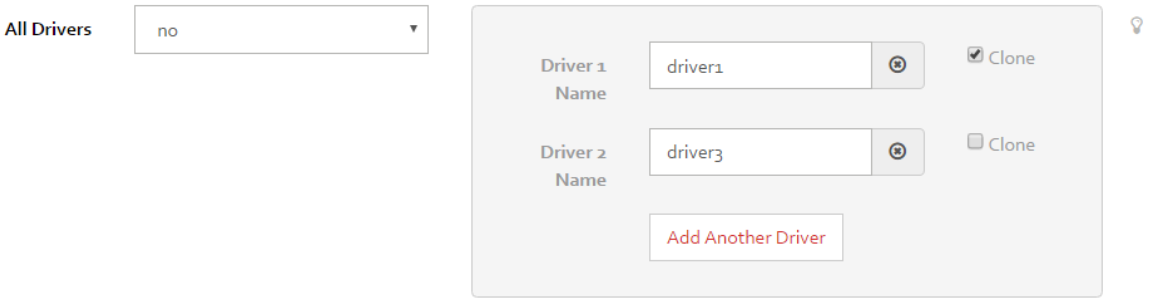


You can not clone added drivers by unchecking the “Clone” checkbox



You can delete existing drivers like this:





8.18. KEYWORD REPOSITORY CLONE

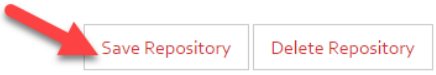
This indicates if you want to clone this keyword repository. By default, it is always set to “yes”.

8.19. KEYWORD REPOSITORY OVERWRITE FILES

This indicates if you want to overwrite existing files. By default it is always set to “yes”.

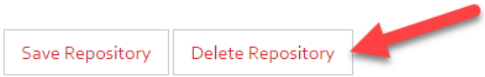
8.20. SAVE KEYWORD REPOSITORY

You can save the keyword repository by clicking on the “Save Repository” button.



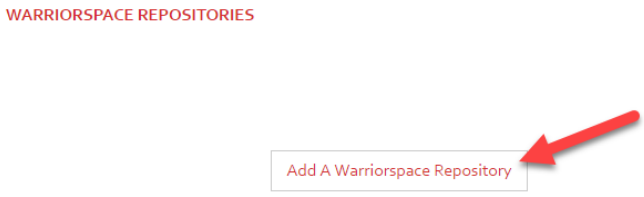
8.21. DELETE KEYWORD REPOSITORY

You can delete the keyword repository by clicking on the “Delete Repository” button.



8.22. ADD A WARRIORSPACE REPOSITORY

You can add a Warriorspace repository here.



On clicking “Add a Warriorspace repository”, this block will be displayed:

8.23. WARRIORSPACE REPOSITORY URL

This is the Warriorspace Repository URL

8.24. WARRIORSPACE REPOSITORY LABEL

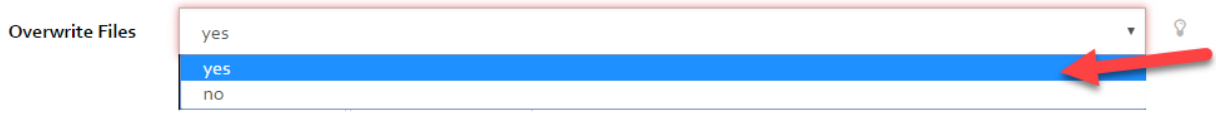
Type in the branch name, commit-id or tag that you want the Warriorspace repository to be at.

8.25. WARRIORSPACE REPOSITORY CLONE

This indicates if you want to clone this repository. By default, it is always set to “yes”.

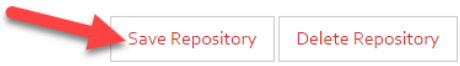
8.26. WARRIORSPACE REPOSITORY OVERWRITE

This indicates if you want to overwrite existing files. By default, it is always set to “yes”.



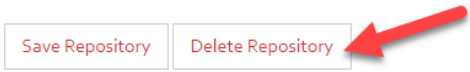
8.27. SAVE WARRIORSPACE REPOSITORY

You can save this Warriorspace repository by clicking on the “Save Repository” button



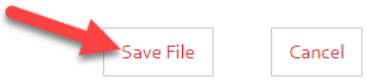
8.28. DELETE WARRIORSPACE REPOSITORY

You can delete this Warriorspace repository by clicking on the “Delete Repository” button



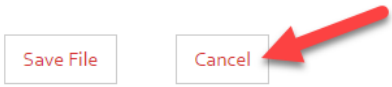
8.29. SAVE WARHORN CONFIG FILE

You can save this Warhorn Config File by clicking on the “Save File” button



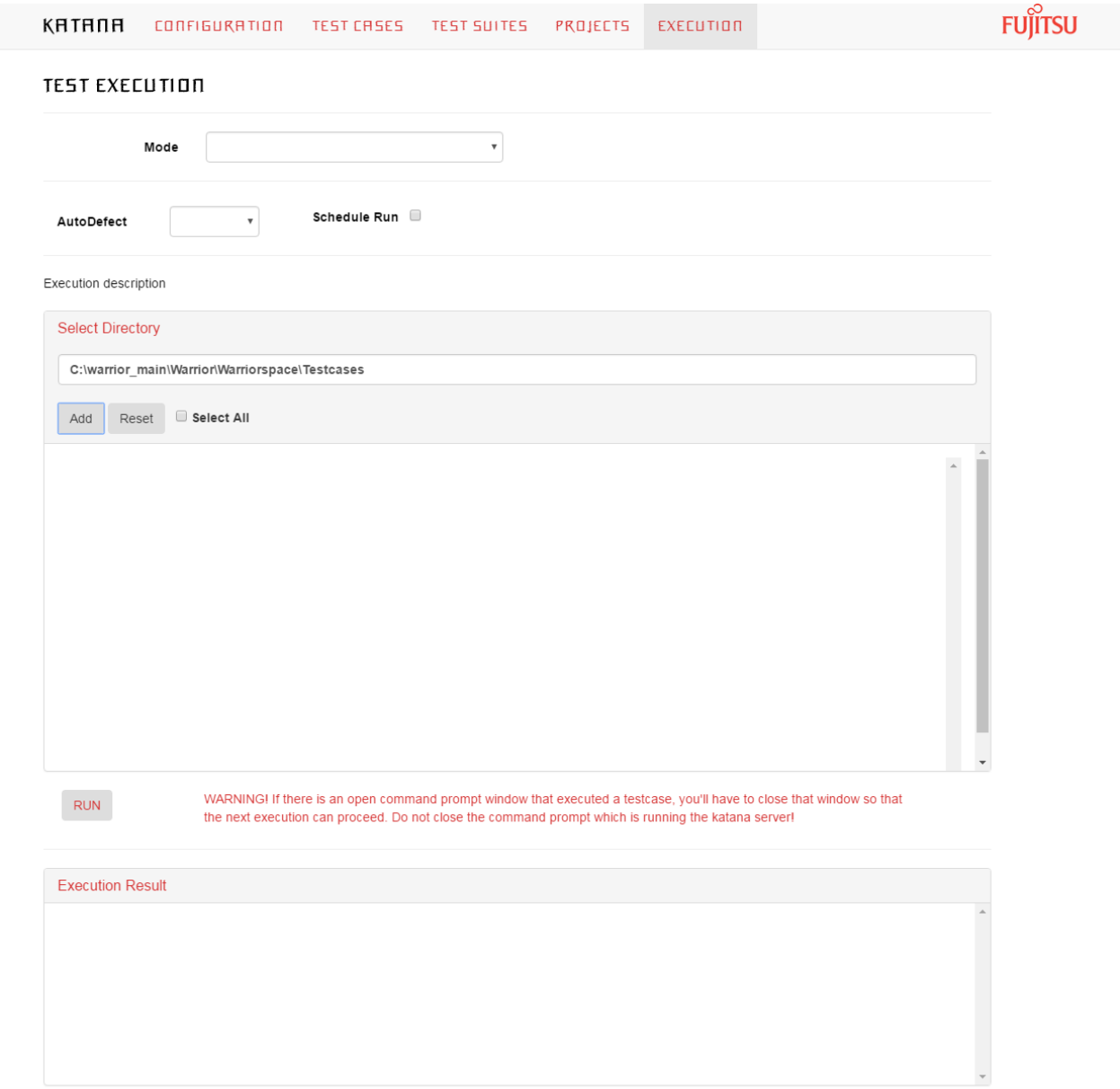
8.30. CANCEL WARHORN CONFIG FILE CREATION

You can cancel this Warhorn Config File creation by clicking on the “Cancel” button

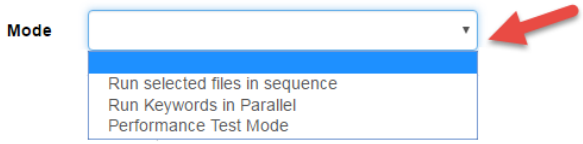


9. EXECUTING WARRIOR THROUGH KATANA

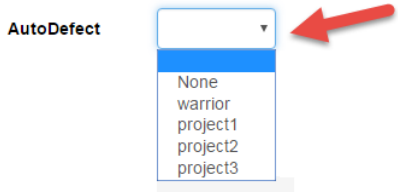
Warrior can also be executed through Katana. On clciking the 'Execution' tab, you will arrive at this page:



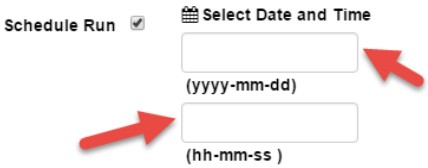
You will be able to select the 'Mode' by clicking on the dropdown:



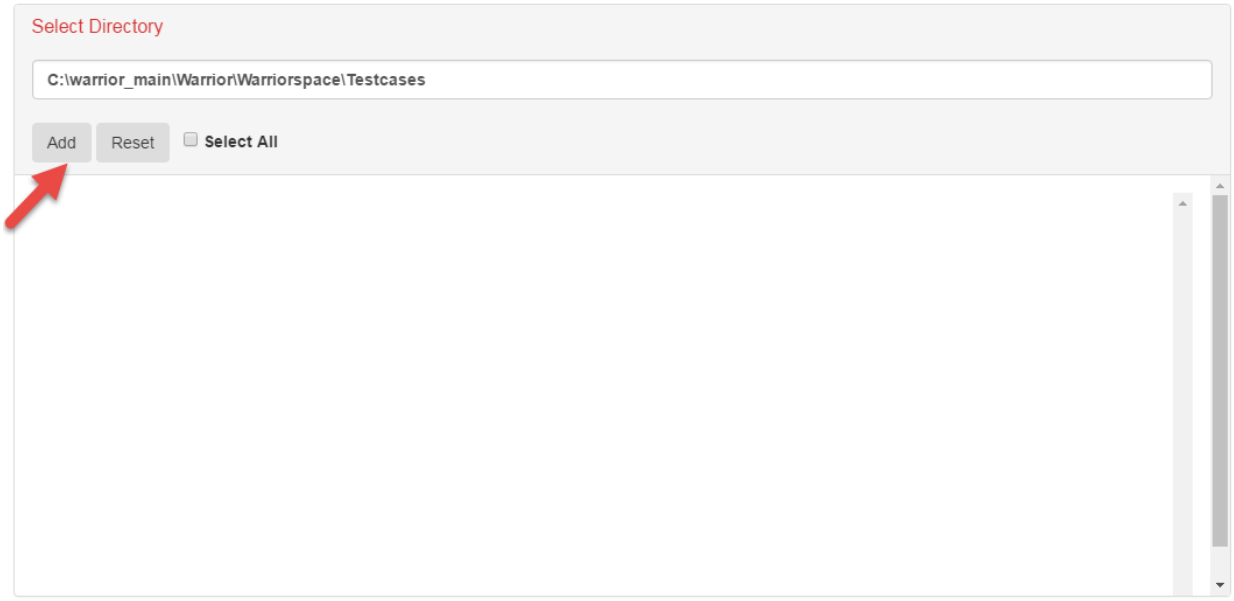
For JIRA auto defect creation, you can select your project from the 'AutoDefect' dropdown:



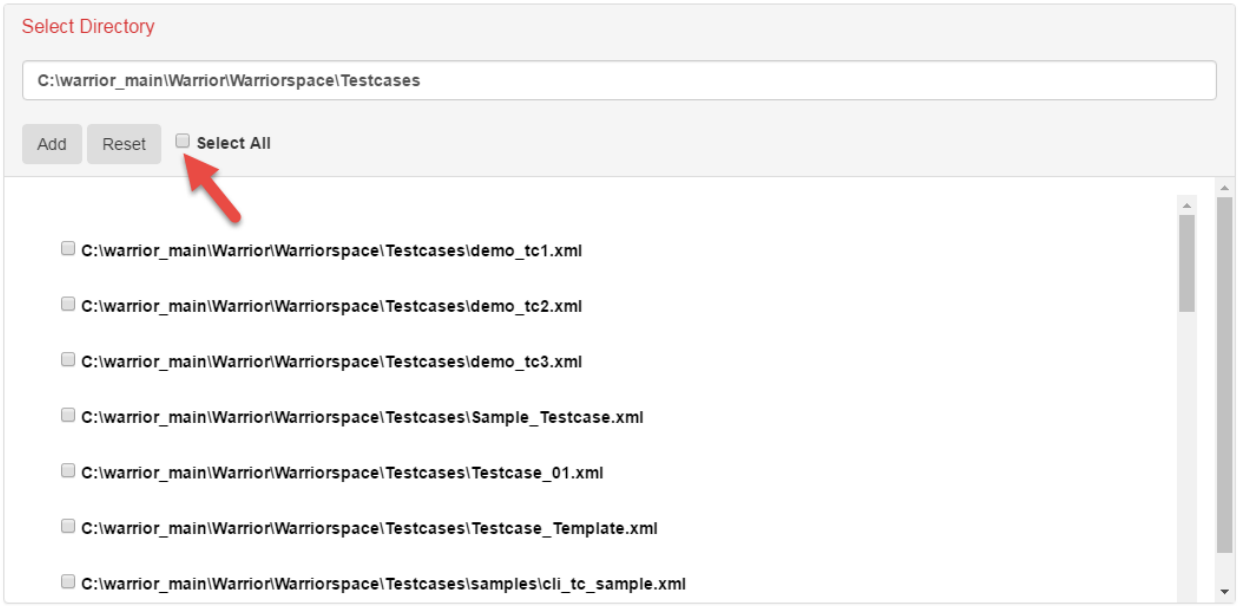
You can schedule the Case run by checking the 'Schedule Run' dropdown and then adding the date and time



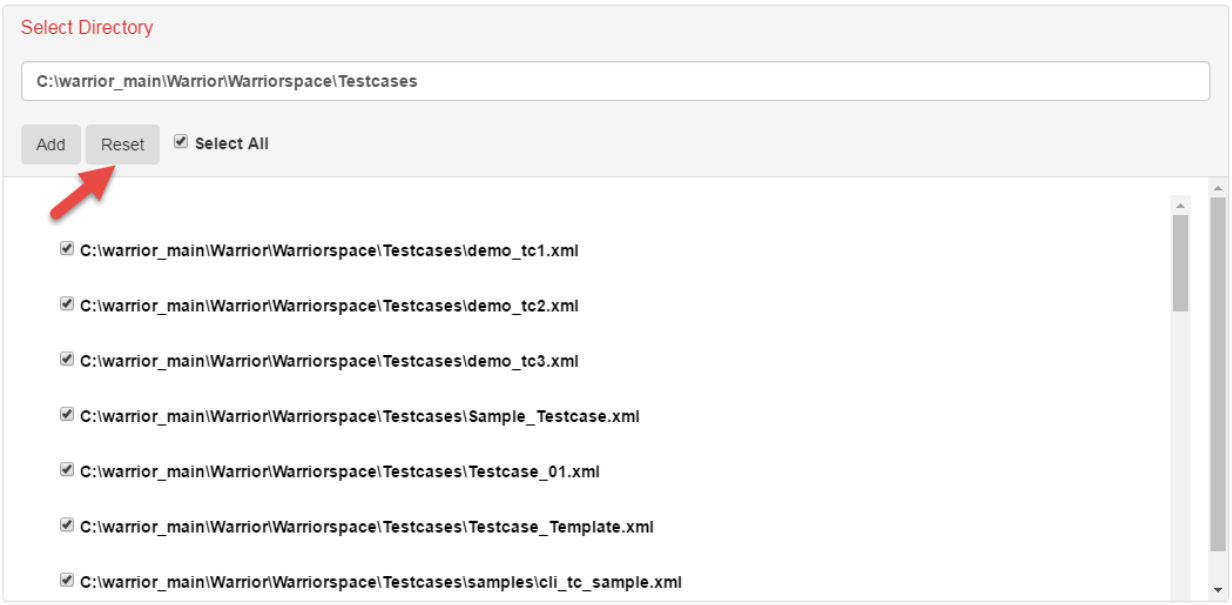
Then, You can click on add, to add all the cases available in the directory showing up in the input box beneath the 'Select Directory', You can also edit that directory if you want to run Cases from a different directory.

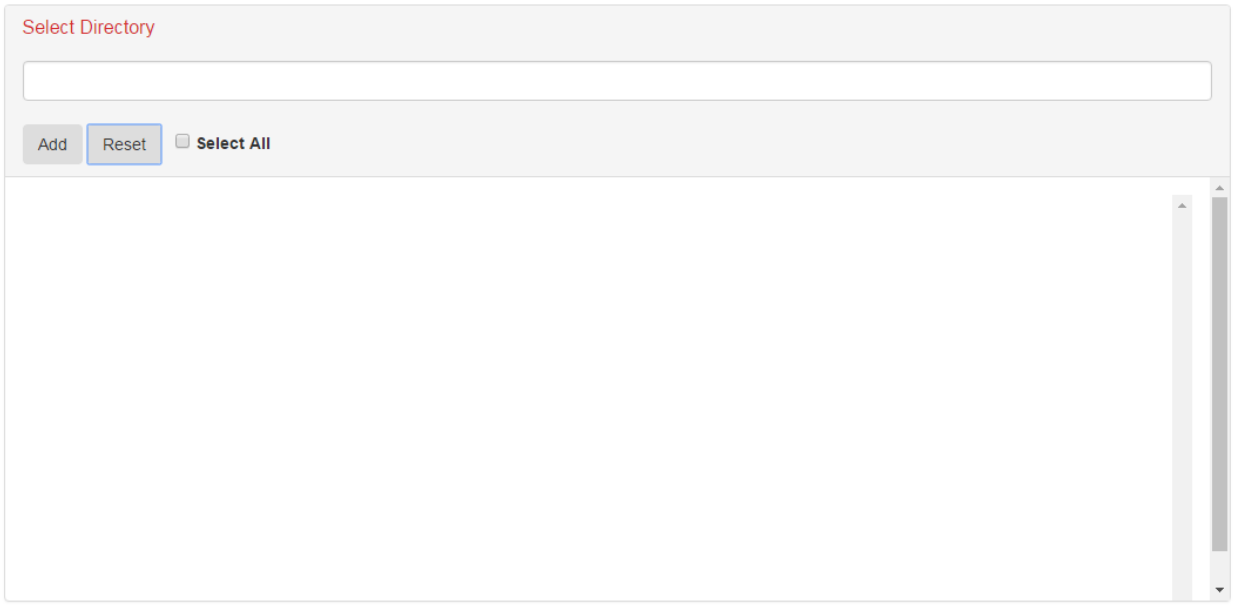


Checking 'Select All' lets you select all cases.

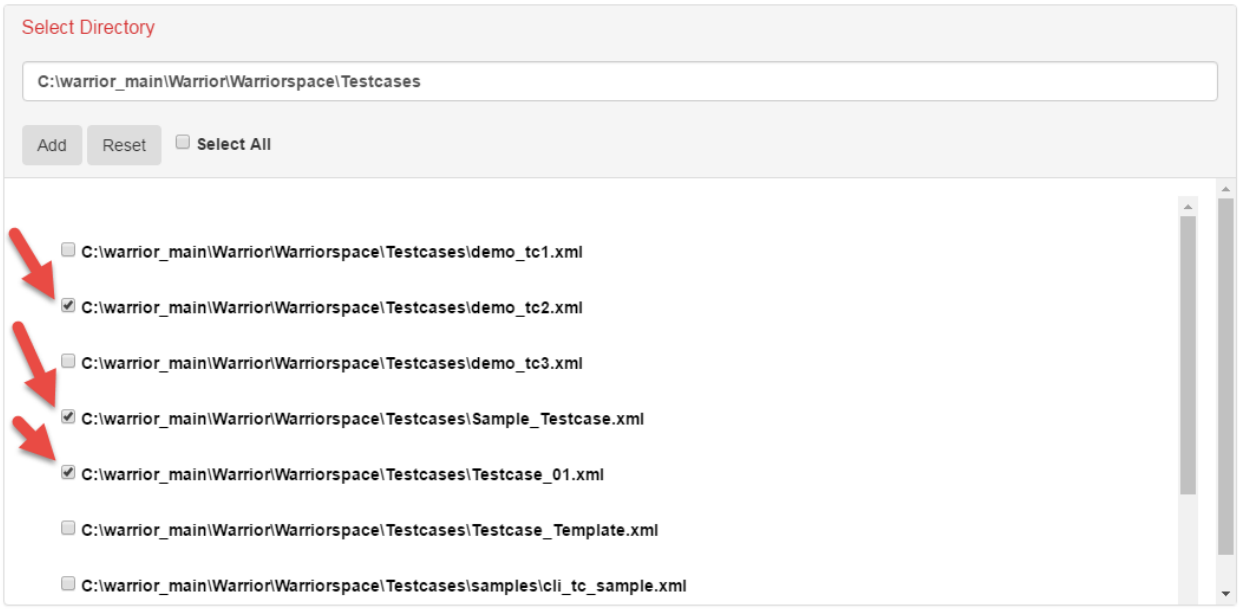


Clicking 'Reset', resets the entire page.



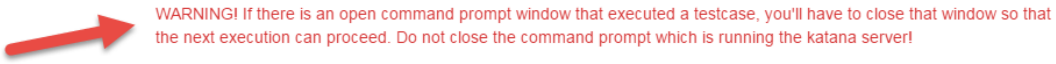


If you want to run a few, particular cases, you can check the checkboxes against those cases. The cases would run in the order in which to check the checkboxes.

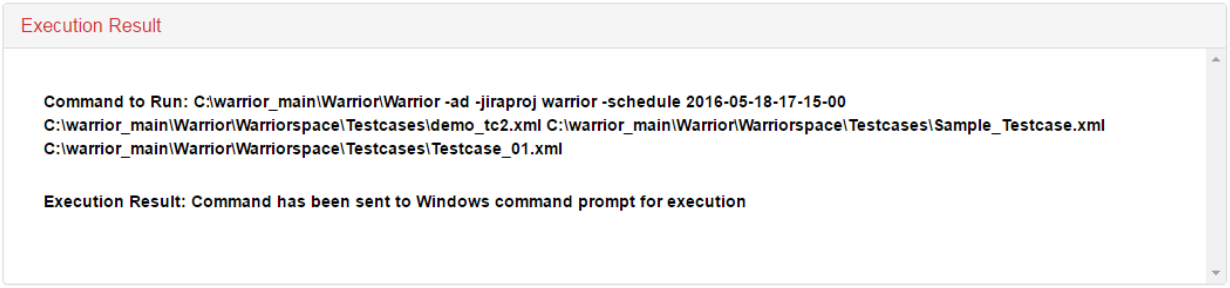


Clicking 'Run' would run the command in gnome terminal – if you running Katana on Linux, or the Command Prompt – if you are running Katana on Windows.

This Warning appears only if Katana is being run on Windows



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



You would be able to view the execution on the console

WARHORN

USER GUIDE

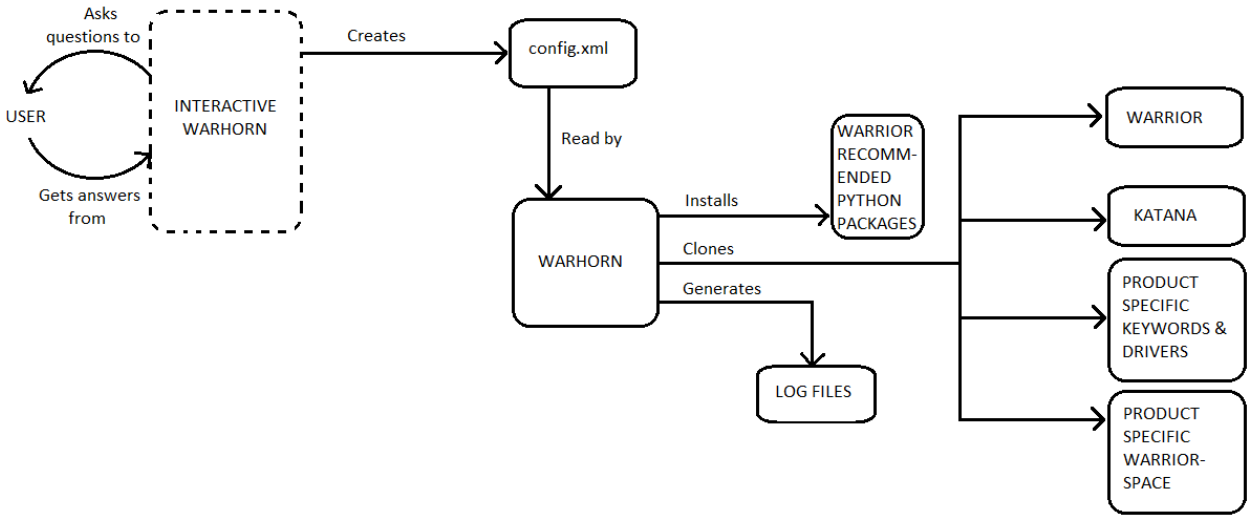
1. WHAT IS WARHORN?

Warhorn is Warrior Framework'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 pre-requisites 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 2.7.0+ till the latest in the 2.7 family
3. Git

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

4. HOW TO INSTALL WARHORN?

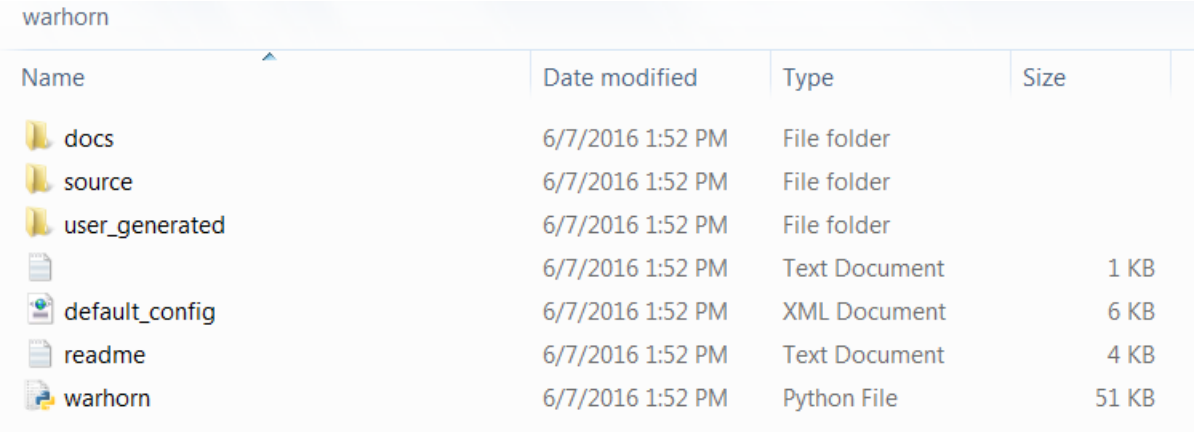
Warhorn already comes with Warrior. If you do not have Warrior cloned on your system, you can find the instructions to clone Warrior here.

The Warhorn directory can be found right inside the root directory. You can go to the warhorn directory with this command:

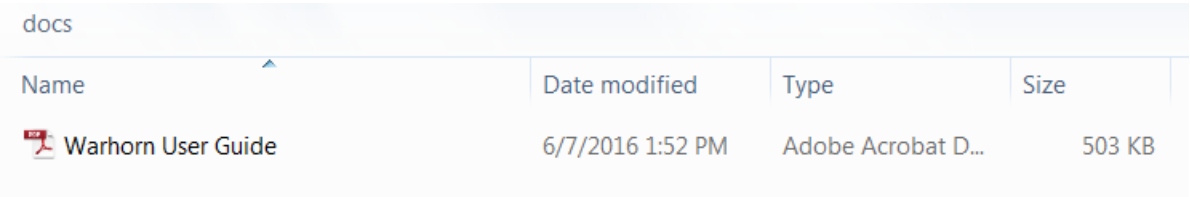
```
cd warriorframework/warhorn
```

5. THE DIRECTORY STRUCTURE

The current Warhorn directory structure looks something like this:

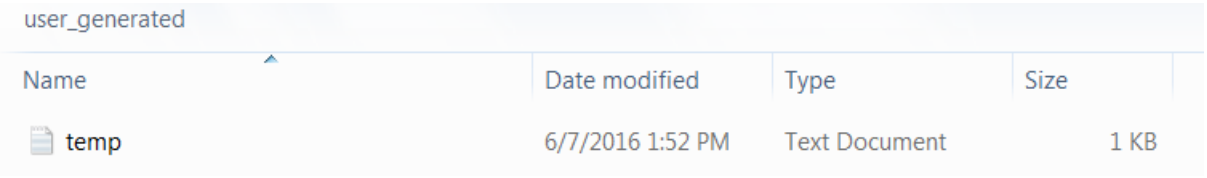


There are three folders and four files inside. The first is the Docs directory which stores the Warhorn User Guide.



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. The configuration file can be created using Katana or via an xml editor.

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 pre-requisites 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 pre-requisites 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 pre-requisites recommended by Warrior have their own <dependency></dependency> tags. The pre-requisites name is given as value to the name attribute and whether you want to install that pre-requisite– 'yes' or 'no' is given as value to the install attribute. The pre-requisite 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 Framework.

```
<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 Framework 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.