Warrior Framework

FUJITSU

Release 4.6.0
May 2021

Warrior User Guide

Contents

1 Overview

In this chapter:

Warrior is a generic automation framework that automates software tasks and processes, in addition to functional, performance and solutions testing. The framework 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 is designed to allow users to effectively create reusable generic keywords and leave the execution intelligence out to be handled by Warrior. The keyword execution is handled by the framework and designed in the cases, suites, and projects. 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 creating XML cases, suites, and projects. Katana provides keyword searching, keyword documentation review, and keyword case and project execution.

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 is designed for automating end-to-end solutions. The framework architecture allows 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 allows the integration of test sets, for example, Wireshark, Spirent, and others within the case. In addition, Warrior framework can integrate cases created in other frameworks and execute them in conjunction with Warrior cases. Warrior framework is able to integrate multiple systems, test sets, and frameworks within a single execution to simplify the design and execution of end-to-end testing.

1.1 Warrior Framework Components

Warrior framework structure can be essentially broken down into the following components:

- Framework
- Keywords
- Cases (Suites and Projects)
- Data Files

The following figure shows the diagrammatic representation of Warrior framework.

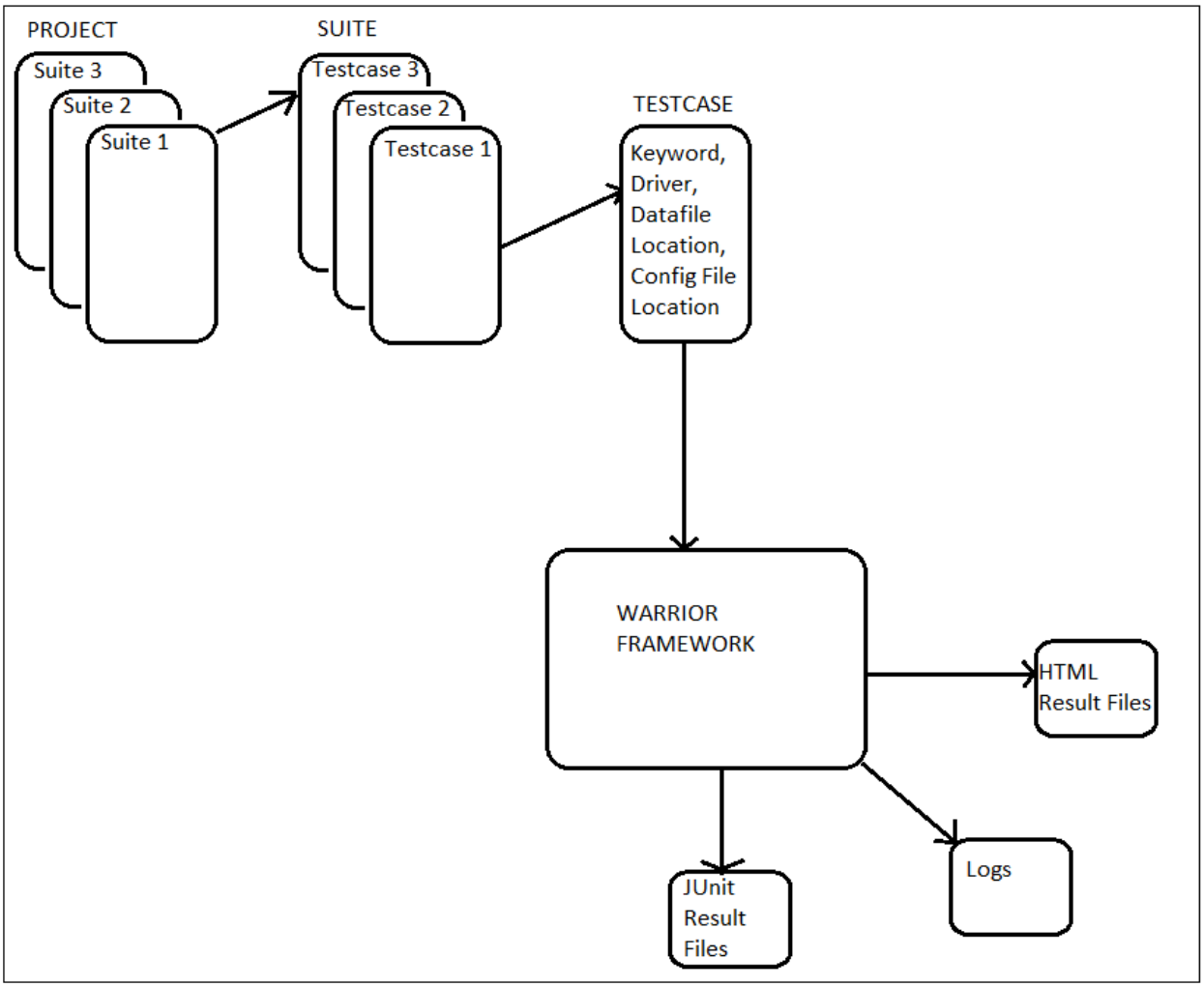


Figure 1
Warrior Framework

The preceding figure provides 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, and whether a particular keyword impacts the case results. 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 execution results.

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

Every component is independent of every other component. This feature 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 a user can test different systems by changing a single file. Using a different case, or by changing the existing case, a user can test different scenarios without writing a single line in Python.

Warrior framework comes with a set of core keywords that help a user to start with automation. Those keywords may not be sufficient for the user if the user has specific requirements, in that case, Warrior framework allows the user to write their own drivers and keywords, and those new, specific keywords can be used in the cases.

1.2 Recommendations

Warrior framework allows users to effectively create reusable generic keywords and leave the execution intelligence out of the keywords to be handled by Warrior framework. When using the Warrior framework, observe the following recommendations:

Important:

- Do not write keywords as one large Python script to be executed. Design the effort and break up the script into re-usable keywords. Warrior framework is not a Python scripting tool. If the 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. The directory structure simplifies the use of Warrior framework.
- Do not combine product drivers into one product driver. These drivers are designed to allow for smaller keyword files grouped by functionality or any logical demarcation. If a user combines into one driver, the keyword files are enlarged over time.

2
Getting Started

In this chapter:

2.1
Setting Up Environment to Execute Warrior

The Warrior environment can be set up manually as well. The user needs Python 3.6 or later.

Pip Installation

To install pip on Linux, run the following commands.

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

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

These commands install pip on the machine to set up the Warrior environment.

2.2
Create and Activate Virtual Environment

This procedure describes how to create a virtual environment for Warrior framework and activate the framework.

Step 1
Specify a directory to create the virtual environment for Warrior framework.

For example, */Documents/warrior*.

Syntax:
```
$cd path_to_create
```

Example:
```
$cd /Documents/warrior
```

Step 2
Create the virtual environment.

```
$python3.6 –m venv warriorenv
```

Step 3
Activate the virtual environment.

```
$source warriorenv/bin/activate
```

Step Result:
After activating the virtual environment, the terminal appears as shown in the following figure.

```
warrioruser ~> $cd Documents/warrior/
warrioruser:~/Documents/warrior> $python3.6 -m venv warriorenv
           :~/Documents/warrior> $source warriorenv/bin/activate
(warriorenv)warrioruser          :~/Documents/warrior> $
```

Figure 2
Virtual Environment Activation Result

✓ **This task is complete.**

Additional Instructions:
Deactivate the virtual environment.

```
$deactivate
```

2.3 Installing and Uninstalling Warrior Framework

In this section:

Warrior framework is available in the following path.

https://pypi.org/project/warriorframework/

2.3.1 Install Warrior Framework using Pip Mode

This procedure describes how to install Warrior framework.

Step 1
Does the user want to install the Warrior framework and Warrior modules separately or together?

> **To install seperately:**
> Continue with next step.
>
> **To install together:**
> Proceed with Step 4.

Note: By default, all the Warrior modules are installed while installing Warrior framework (refer to Step 2) if the Warrior framework version is earlier than Version 4.6.0. Skip Step 3 to Step 5 if Warrior framework version is earlier than Version 4.6.0.

Step 2
Install Warrior framework.

```
pip install warriorframework
```

Note: Install a specific version of Warrior framework.

```
pip install warriorframework==4.6.0
```

Step Result:
Warrior framework and its dependent packages are installed.

```
(venv_test) [redacted] ~ % pip install warriorframework
Collecting warriorframework
  Downloading https://files.pythonhosted.org/packages/83/cc/ef9bd2443e83ac46f6cb2df8503893cb40aadc14e9d4582a62967ab4eaf1/warriorframework-4.6.0
-py3-none-any.whl (5.2MB)
    100% |[redacted]| 5.2MB 144kB/s
Collecting configobj==5.0.6 (from warriorframework)
  Downloading https://files.pythonhosted.org/packages/64/61/079eb60459c44929e684fa7d9e2fdca403f67d64dd9dbac27296be2e0fab/configobj-5.0.6.tar.gz
Collecting requests==2.21.0 (from warriorframework)
  Downloading https://files.pythonhosted.org/packages/7d/e3/20f3d364d6c8e5d2353c72a67778eb189176f08e873c9900e10c0287b84b/requests-2.21.0-py2.py
3-none-any.whl (57kB)
    100% |[redacted]| 61kB 167kB/s
Collecting kafka-python==1.4.6 (from warriorframework)
  Downloading https://files.pythonhosted.org/packages/82/39/aebe3ad518513bbb2260dd84ac21e5c30af860cc4c95b32acbd64b9d9d0d/kafka_python-1.4.6-py2
.py3-none-any.whl (259kB)
    100% |[redacted]| 266kB 202kB/s
Collecting six (from configobj==5.0.6->warriorframework)
  Downloading https://files.pythonhosted.org/packages/d9/5a/e7c31adbe875f2abbb91bd84cf2dc52d792b5a01506781dbcf25c91daf11/six-1.16.0-py2.py3-non
e-any.whl
Collecting idna<2.9,>=2.5 (from requests==2.21.0->warriorframework)
  Downloading https://files.pythonhosted.org/packages/14/2c/cd551d81dbe15200be1cf41cd03869a46fe7226e7450af7a6545bfc474c9/idna-2.8-py2.py3-none-
any.whl (58kB)
    100% |[redacted]| 61kB 224kB/s
Collecting chardet<3.1.0,>=3.0.2 (from requests==2.21.0->warriorframework)
  Downloading https://files.pythonhosted.org/packages/bc/a9/01ffebfb562e4274b6487b4bb1ddec7ca55ec7510b22e4c51f14098443b8/chardet-3.0.4-py2.py3-
none-any.whl (133kB)
    100% |[redacted]| 143kB 236kB/s
Collecting certifi>=2017.4.17 (from requests==2.21.0->warriorframework)
  Downloading https://files.pythonhosted.org/packages/5e/a0/5f06e1e1d463903cf0c0eebeb751791119ed7a4b3737fdc9a77f1cdfb51f/certifi-2020.12.5-py2.
py3-none-any.whl (147kB)
    100% |[redacted]| 153kB 198kB/s
Collecting urllib3<1.25,>=1.21.1 (from requests==2.21.0->warriorframework)
  Downloading https://files.pythonhosted.org/packages/01/11/525b02e4acc0c747de8b6ccdab376331597c569c42ea66ab0a1dbd36eca2/urllib3-1.24.3-py2.py3
-none-any.whl (118kB)
    100% |[redacted]| 122kB 288kB/s
Installing collected packages: six, configobj, idna, chardet, certifi, urllib3, requests, kafka-python, warriorframework
  Running setup.py install for configobj ... done
Successfully installed certifi-2020.12.5 chardet-3.0.4 configobj-5.0.6 idna-2.8 kafka-python-1.4.6 requests-2.21.0 six-1.16.0 urllib3-1.24.3 wa
rriorframework-4.6.0
```

Figure 3
Dependent Warrior Package Installation

Note: This command provides *Warrior CLI* utility, which enables users to execute Warrior project, suite, or test case files.

Step 3
Install the required Warrior modules from PyPI Server.

```
pip install warriorcli
pip install warriornetconf
pip install warriorsnmp
pip install warriorgnmi
pip install warriorcloudshell
pip install warriorrest
pip install warriorselenium
pip install warriorkafka
pip install warriornetwork
pip install warriorserver
pip install warriormongo
pip install warriordemo
pip install warriorfile
pip install warriorciregression
pip install warriormicroapps
pip install warriorwapp
```

Note: The Warrior framework contains all the modules. A user can install only the required modules, for example, if the user has only CLI automation tasks, install only the Warrior CLI module.

Note: Install a specific version of Warrior module.

```
pip install warriorcli==1.0.0
```

Step Result:
The required Warrior modules are installed.

```
(venv_test) [redacted] ~ % pip install warriorcli
Collecting warriorcli
  Downloading https://files.pythonhosted.org/packages/67/e5/cbe515be8bd5702387937a18fcceadce59010a6022c0da161f9cfab6ee4e/warriorcli-1.0.0-py3-none-any.whl (49kB)
    100% |[redacted]| 51kB 298kB/s
Collecting pexpect==4.8.0 (from warriorcli)
  Downloading https://files.pythonhosted.org/packages/39/7b/88dbb785881c28a102619d46423cb853b46dbccc70d3ac362d99773a78ce/pexpect-4.8.0-py2.py3-none-any.whl (59kB)
    100% |[redacted]| 61kB 206kB/s
Collecting pycryptodome==3.6.1 (from warriorcli)
  Downloading https://files.pythonhosted.org/packages/9c/83/8f8a9e94d3cc495dd49082ac79e366b368cd10c8d25734fbcea59de93e5a/pycryptodome-3.6.1-cp36-cp36m-macosx_10_6_intel.whl (7.2MB)
    100% |[redacted]| 7.2MB 376kB/s
Collecting ptyprocess>=0.5 (from pexpect==4.8.0->warriorcli)
  Downloading https://files.pythonhosted.org/packages/22/a6/858897256d0deac81a172289110f31629fc4cee19b6f01283303e18c8db3/ptyprocess-0.7.0-py2.py3-none-any.whl
Installing collected packages: ptyprocess, pexpect, pycryptodome, warriorcli
Successfully installed pexpect-4.8.0 ptyprocess-0.7.0 pycryptodome-3.6.1 warriorcli-1.0.0
```

Figure 4
Warrior Module Installation

Step 4
Install both Warrior framework and Warrior modules.

```
pip install warriorframeworkallmodules
```

Note: Skip this step, if a user has already installed the Warrior framework and Warrior modules separately.

Step 5
Execute Warriormigrate.

- Execute the following command, if the user has installed all the Warrior modules.

  ```
  Warriormigrate -pkgs_list all
  ```

- Execute the following command, if the user has installed only the required modules, for example, warriorcli, warriornetconf.

  ```
  Warriormigrate -pkgs_list warriorcli,warriornetconf
  ```

Step 6
To know more about the Warrior CLI utility usage, enter `Warrior --help` command.

Figure 5
Warrior Help

✓ **This task is complete.**

2.3.2
Uninstall Warrior Framework using Pip Mode

This procedure describes how to uninstall Warrior framework.

Step 1
Uninstall Warrior framework.

```
pip uninstall warriorframework
```

Step 2
Uninstall a specific version of Warrior framework.

```
pip uninstall warriorframework==4.3.1
```

Step Result:
Warrior framework and its dependent packages are uninstalled.

✓ **This task is complete.**

2.4
Install Warrior Keyword Repository

This procedure describes how to install Warrior keyword repository in a particular directory.

Step 1
Go to a desired directory location.

```
$cd Documents/user_repos
```

Step 2
Enter the following Git URL.

```
$git clone ssh://git@test-swtl-git.test.net.local/warkey/kw_tseries.git
```

Step Result:
The Warrior keyword repository is installed in the directory.

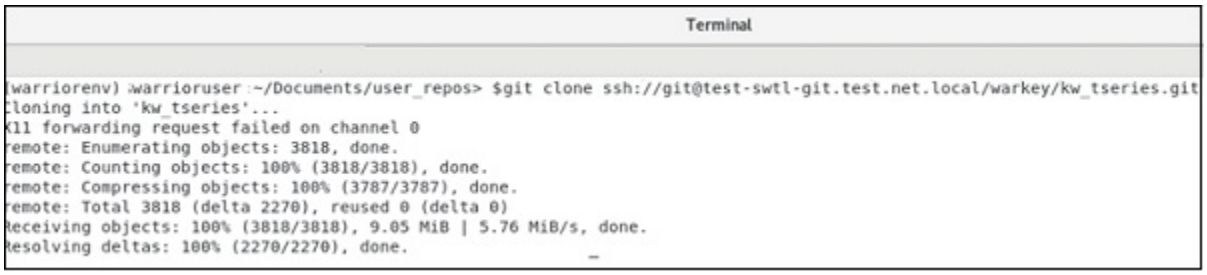


Figure 6
Warrior Keyword Repository Result

Step 3
Install any custom pip packages that are required for Warrior keyword repository (part of Warrior Keyword Repo requirements.txt).

Step 4
Add the keylib paths to *PYTHONPATH* environment variable.

✓ **This task is complete.**

2.5
Install Testcase Repository

This procedure describes how to install testcases in a particular directory.

Step 1
Go to a desired directory location.

```
$cd Documents/test
```

Step 2
Enter the following Git URL.

```
$git clone ssh://git@test-swtl-git.fnc.net.local/was/common.git
```

Step Result:
Testcases is installed in the directory.

Figure 7
Testcases Result

✓ **This task is complete.**

2.6
Run Warrior

After setting up the environment to run Warrior, perform the following procedure to execute Warrior.

Step 1
Go to a desired directory.

Step 2
Run one of the Demo Cases that Warrior contains.

Syntax:
```
$ Warrior path_to_script.xml
```

Example:
```
$ Warrior Warriorspace/Testcases/Demo_Cases/Demo_Test_Of_Inventory_
Management_System.xml
```

Step Result:
Warrior Demo Cases runs.

Figure 8
Warrior Demo Case

✓ **This task is complete.**

2.7
Changing Logs Location

Note: By default, all log directories are stored in the user home directory.

User can execute a CLI command to change the location of the logs directory.

Syntax:
```
$ Warrior -outputdir <location of log dir> <testscript>
```

Example:
```
$ Warrior -outputdir /data/usr/bin    ../Testcases/Demo_Cases/
Demo_Test_Of_Inventory_ Management_System.xml
```

Figure 9
Logs Directory

2.8
Creating External Keyword Directory

The user must observe the following recommendations to create the external keyword directory:

- Create a Python package with the following directory structure.

```
PACKAGE_NAME
    __init__.py
```

Notes:

- Framework—Files to use in *Actions* directory
- Actions—Contains files with user keywords
- ProductDrivers—Contains driver files (one driver for each file in *Actions* directory)
- Tools—Files to use in *Actions* or *Framework* directory

- All imports used in external keyword repository files should be fully qualified starting from the package name.

Example:
```
import package_name.Actions.CustomActions
```

- All imports referring to files in Warrior framework should use package name *warrior*.

Example:
```
from warrior.WarriorCore import kw_driver
```

3
Setting Up Warrior

In this chapter:

3.1
JIRA Settings

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

This procedure describes how to configure Warrior to report defects automatically.

Step 1
Open the JIRA configuration file located in the virtual environment folder from the following path.

`Virtualenv/warrior_settings/Tools/jira/jira_config.xml`

Step Result:
JIRA configuration file opens.

```
<?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>

```

Figure 10
JIRA Configuration File—Part 1

```
                <Assignee>Automatic</Assignee>
        <project_key>WAR</project_key>
        <append_log>true</append_log>
        <issue_type>
            <type>story</type>
            <pass>pass</pass>
            <fail>fail</fail>
            <error>error</error>
            <skip>skip</skip>
        </issue_type>
        <issue_type>
            <type>task</type>
            <pass>pass</pass>
            <fail>fail</fail>
            <error>error</error>
            <skip>skip</skip>
        </issue_type>
        <issue_type>
            <type>default</type>
            <pass>pass</pass>
            <fail>fail</fail>
            <error>error</error>
            <skip>skip</skip>
        </issue_type>
    </system>

</jira>
```

Figure 11
JIRA Configuration File—Part 2

Each system tag carries information about the project that the user wants to upload defects if failures are observed.

Step 2
Enter a name for that project in the name attribute for the system tag.

Step 3
Type in the URL for that project in the URL child tag.

Step 4
Type in the username and the password for the project.

Step 5
Enter the key for that project.

✓ **This task is complete.**

Additional Instructions:

Some rules are available by which Warrior uses this XML file:

- The XML file can contain multiple systems each being a separate JIRA project. Only one project can be defined as a default project.
- When executing Warrior through the CLI, if a JIRA project is not specified, Warrior uses the project defined as default.
- To define a project as default, it should be marked as default=”true”.
- If multiple projects are defined as default, Warrior uses the first project marked as default.
- If no project is marked as default, Warrior uses the first project in the XML file. In the following figure, the last project is marked as default.

User can execute Warrior either via command line JIRA Bug Reporting or Executing Warrior through Katana.

3.2
Encryption Settings

Warrior framework allows 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
```

A user can add *sudo* in front of the command, if the user wants to run it as an admin. This command provides 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 and Results Directory Settings

Prerequisites:

Warrior, by default, uses this directory to store the files created during its execution—Log and result files. To know more information, refer to Understanding Logs and Results.

A user can configure Warrior to create these log files and directories in a specific location rather than the default. This configuration can be achieved with the following steps:

Step 1
Open the *w_settings* file located in the virtual environment folder from the following path.

```
Virtualenv/warrior_settings/Tools/w_settings.xml
```

Step Result:
The W_Settings file opens.

```
<Default>
    <Setting name="def_dir">
        <Logsdir />
        <Resultsdir />
</Setting><Setting compress="No" mail_on="" name="mail_to">

        <smtp_host />
        <sender />
        <receiver />
        <subject />
    </Setting>

    <Setting location="http://localhost:5000/katana" name="katana" />
</Default>
```

Figure 12
W_Settings File

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

Step 2
To specify a directory for the Log files, add that file path as a value to the Logsdir tag.

Step 3
To specify a directory for the Results files, add that file path as a value to the Resultsdir tag.

Step Result:
Warrior uses the directory paths mentioned previously to store the generated log and result files.

✓ **This task is complete.**

3.4
Email Settings

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

Step 1
Open the w_settings configuration file that is located in the virtual environment folder from the following path.

```
Virtualenv/warrior_settings/Tools/w_settings.xml
```

Step Result:
The W_Settings file opens.

```
<Default>
    <Setting name="def_dir">
        <Logsdir />
        <Resultsdir />
</Setting><Setting compress="No" mail_on="" name="mail_to">

        <smtp_host />
        <sender />
        <receiver />
        <subject />
    </Setting>

    <Setting location="http://localhost:5000/katana" name="katana" />
</Default>
```

Figure 13
W_Settings File

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

Step 2
If the user wants Warrior to send this notification email only after the execution has finished, enter the value ***per_execution*** in the ***mail_on*** attribute for the Settings tag.

Step 3
If the user wants Warrior to send this notification email only when the first failure is observed during the execution, enter the value ***first_failure*** in the ***mail_on*** attribute for the Settings tag.

Step 4
If the user wants Warrior to send this notification email after each failure that may be observed during the execution, enter the value ***every_failure*** in the ***mail_on*** attribute for the Settings tag.

Step 5
Enter a combination of the three options mentioned in previous steps, as comma separated values in the ***mail_on*** attribute.

Important: If these details are wrong or left unfilled, Warrior is not able to send an email.

✓ **This task is complete.**

3.5
Database Settings

If the files are stored in a database, a user can connect to that database so that Warrior can access those files. Warrior supports the MongoDB as its database.

Open the database_config.xml configuration file in the virtual environment folder from the following path.

`Virtualenv/warrior_settings/Tools/database/database_config.xml`

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

```
<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>
```

Figure 14
Dataservers Block

Required fields for MongoDB server are:

dbtype	Type of the database
host	DB server IP, optional when URI is specified. ***Note:*** The default host is localhost.
port	DB server port, optional when URI is specified. ***Note:*** The default port is 27017.

uri	Connection string in URI format, optional when host and port are specified. ***Example:*** mongodb://localhost:27017
dbname	Database name

Each database contains 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, as shown in the following figure.

```
<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>
```

Figure 15
Database Configuration File

Warrior allows to store the results and logs generated after each execution in the database.

Systems specified under *resultservers* block are 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>
```

Figure 16
Resultservers Block

With the preceding information, Warrior is 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 is used. If no default system exists, the first system specified under *resultservers* block is used as a 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 sends relevant commands to it to obtain details about that systems, for example, the software it is running, the current version, and so on.

Open the connect_settings.xml configuration file in the virtual environment folder from the following path.

`Virtualenv/warrior_settings/Tools/connection/connect_settings.xml`

The Connect_Settings file opens.

```
<?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>
```

Figure 17
Connect_Settings File

Here, a user can add a system tag with a name attribute and provide a search string to Warrior. Warrior looks for this search string in the system response and identifies that system.

A user can provide a testdata file to Warrior as a relative path in the testdata tag under the system tag. Typically, this testdata file should be stored in the configs directory located in:

`Virtualenv/warrior_settings/Tools/connection/configs`

This testdata file appears as shown in the following figure.

```
<?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>
```

Figure 18
Testdata File

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 the response obtained would be displayed on the console.

4
Documentation

In this chapter:

4.1
User Guide

The user guide for Warrior is located at:

https://github.com/warriorframework/warriorframework_py3/blob/master/docs/Warrior%20Framework%20User%20Guide.pdf

TC_Generator

The user guide is located at:

https://github.com/warriorframework/warriorframework_py3/blob/master/docs/TC_Generator_user_guide.pdf

4.2
Keyword Documentation

Prerequisites:

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

Example: If the user is interested in knowing more about the connect keyword in the CLI module, perform the following steps to read the documentation for that keyword.

Step 1
Go to the *`warriorframework_py3/warrior/Actions`* directory.

Step Result:
A directory structure like this appears.

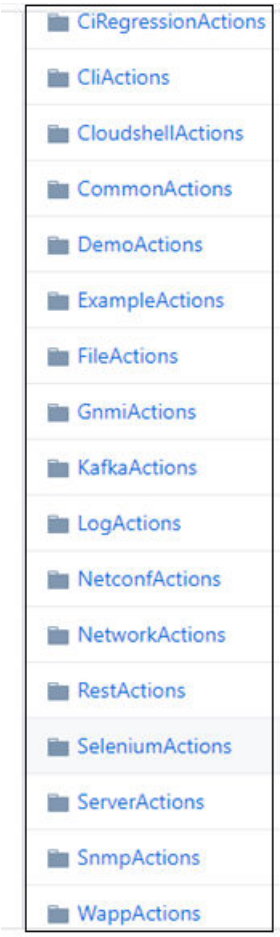


Figure 19
Actions Directory

Step 2
Open the directory for the module that a user is interested in—in this case, the CLI module. This CLI module has just one file inside of it.

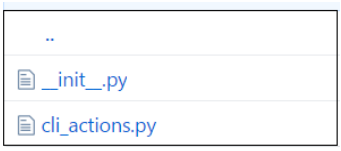


Figure 20
CLI Module

Step 3
Open the cli_actions.py file. The documentation for each available keyword is given right beneath the declaration of that keyword. See the following figure.

```
def connect(self, system_name, session_name=None, prompt=".*(%|#|\$)",
            ip_type="ip", via_host=None, tuple_pty_dimensions=None):
    """
    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.
```

Figure 21
cli_actions.py File

The preceding figure shows the (partial) documentation for the connect keyword.

Additional Instructions:
Opening a Python file and reading though it can be cumbersome for a case developer. So an easier way exists to do the same thing. The user can use Katana to view the documentations for all the available keywords. For that the user needs to:

1. Run Katana first and bring up UI, if a user has never used Katana before. If Katana is already set up, skip this step.

2. After it is completed, click the Cases app in Katana.

3. Refer to the *Cases section* in *Katana User Guide* for creating a case and selecting keyword, drivers, and adding steps. After selecting a developed keyword, a description is provided for the keyword under Signature field.

5
Warrior Tools

In this chapter:

5.1 Ironclaw

In this section:

Ironclaw is a Warrior framework XML verification tool. Ironclaw can be used to verify the structure of the XML files used by Warrior framework. Ironclaw automatically detects the XML type (case, suite, project) and do the following as applicable.

5.1.1 Verification of Case XML Files

After Ironclaw identified that the XML is a case XML. It performs the following tasks:

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 and check if the keyword is in the corresponded actions package.
4. Verify that the driver exists in the Product Drivers folder.

5.1.2 Verification of Suite XML Files

After Ironclaw identified that the XML is a suite XML. It performs the following tasks:

- Verify conformance to suite XSD.
- Verify all cases listed in suite exist.
- Verify each case in the suite.

5.1.3 Verification of Project XML Files

After Ironclaw identified that the XML is a project XML. It performs the following tasks:

1. Verify conformance to project XSD.
2. Verify all suites listed in project exist.
3. Verify each suite as described in Creating Project.

5.1.4 Running Ironclaw

To run Ironclaw, execute the following command.

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

5.2
TC_Generator Tool

TC_Generator is one of a Warrior tool. The tool has the capability to create a testcase with Warrior structure and without Warrior structure according to the user requirements. Finally, the tool provides the location of created files and executable command of the created testcase.

TC_Generator supports the following four types of testcases:

- CLI
- SNMP
- NETCONF
- REST

Run TC_Generator Tool

```
./Warrior -tc_gen - -type="cli" - -tc_name="cli_demo_with_dip.xml" - -ip="localhost" - -
username="uname" - -password="pwd" - –cli_port="1234" - –dip_port="1234"

./Warrior -tc_gen - -type="cli" - -tc_name="cli_demo_with_dip.xml" –ip="localhost" - -
username="uname" - -password="pwd" - -cli_port="1234" –dip_port="1234" -wrs

./Warrior -tc_gen --type="cli" --tc_name="cli_demo_without_dip.xml" --ip="localhost" --
username="uname" --password="pwd" -cp="23009"

./Warrior -tc_gen --type="cli" --tc_name="cli_demo_without_dip.xml" --ip="localhost" --
username="uname" --password="pwd" -cp="23009" –wrs
```

Figure 22
CLI Usage

```
./Warrior -tc_gen --type='snmp' --tc_name='tc_name.xml' --username='username' --password='pwd' --ip='localhost' -cp='1234' -sp='1234'

./Warrior -tc_gen --type='snmp' --tc_name='tc_name.xml' --username='username' --password='pwd' --ip='localhost' -cp='1234' -sp='1234' –wrs
```

Figure 23
SNMP Usage

```
./Warrior -tc_gen --type='netconf' --tc_name='tc_name.xml' --username=uname' --password='pwd' --ip='localhost' -np=1234'

./Warrior -tc_gen --type='netconf' --tc_name='tc_name.xml' --username='uname' --password='pwd' --ip='localhost' -np='23013' –wrs
```

Figure 24
Netconf Usage

```
./Warrior -tc_gen --type='rest' --tc_name='tc_name.xml' --username ='uname' --password ='pwd'

./Warrior -tc_gen --type='rest' --tc_name='tc_name.xml' --username ='uname' and --password ='pwd' -wrs

./Warrior -tc_gen --type='rest' --tc_name='tc_name.xml'

./Warrior -tc_gen --type='rest' --tc_name='tc_name.xml' –wrs
```

Figure 25
REST Usage

TC_Generator provides the Warrior structure when a user provided the - - *with_repo_structure*.

If the user did not provide - - *with_repo_structure* argument, it creates files in the current working directory.

5.3
MockRun

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, that is, 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, use the `–summary` option.

```
Python Warrior -summary –mockrun path/to/case
```

6
Warrior Directory Structure

In this chapter:

A total of seven directories and one Warrior framework are executable in the Warrior framework directory. What to expect in each directory and its utility has been explained in the following sections. Whether a user is a case developer or a keyword developer, acclimatizing to Warrior framework directory structure is very important for a good understanding of Warrior framework. The following figure shows the current Warrior framework directory structure.

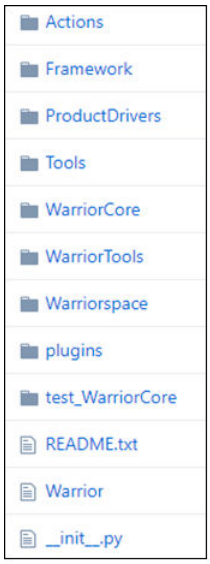


Figure 26
Warrior Framework Directory

6.1
Actions Directory

Warrior framework 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 needs to be added to either an existing library or to a new library within the Actions directory. All these libraries are explained in detail below.

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

```
Warriorframework_py3/warrior/Actions
```

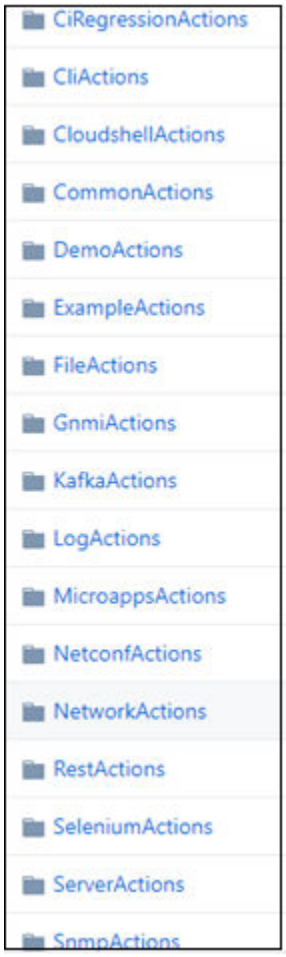


Figure 27
Action Directory

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

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

An Actions package in Warrior is shown in the following figure.

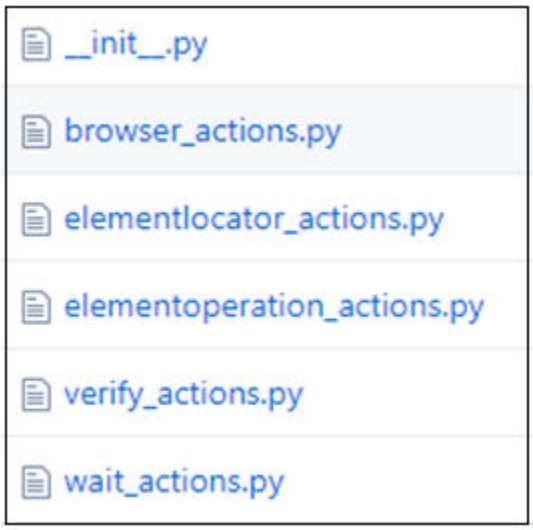


Figure 28
Actions Package

Selenium Actions Package

The Selenium Actions package contains keywords related to Selenium operations.

The Selenium Actions directory located in:

```
Warriorframework_py3/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.

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 web page like clicking, typing text, double-clicking, drag and drop, and so on.

CI Regression Actions Package

The CI Regression actions package is the first Actions package that a user notices after opening the Actions directory. This CI Regression class contains keywords that are used in Warrior regression test.

CliActions Package

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

Cloudshell Actions Package

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

CommonActions 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.

DemoActions Package

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

Example Actions Package

The Example actions package has all connect case-related keywords. These keywords connect to the SSH® port of the given system or subsystems.

File Actions Package

The file actions package has all file-related keywords. These keywords are related to actions used in file KW.

GNMI Actions Package

The GNMI actions package defines gRPC-based protocol for the modification and retrieval of configuration from a network element. As well as the control and generation of telemetry streams from a network element to a data collection system. Warrior keyword supports the gnmi client services.

Kafka Actions Package

The Kafka actions package has all Kafka keywords. These keywords are related to actions performed on Kafka.

Log Actions Package

The log actions package has all log printing-related keywords. These keywords are related to actions performed for logging within test and to print the given message.

Microapps Actions Package

The microapps actions package has all Warrior result product keywords. These keywords are related to actions performed to return warrior status and failed command details. By using these, a user must get current status of warrior script and failure reason.

NetconfActions Package

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

Network Actions Package

The Network actions package contains three sub packages—Connection package, Diagnostics package, and FileOps package.

Rest Actions Package

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

Server Actions Package

The Server actions package contains server-related keywords. This class has all the keywords related to bottle web socket server-related keywords.

SnmpActions Package

The SnmpActions package has the implementation of the standard SNMP protocol commands for SNMPv1 and SNMPv2c. It has the CommonSnmpActions class that contains all the keywords related to SNMP.

Wapp Actions Package

The Wapp actions package contains utilities that connect Warrior to Wapps.

6.2 Framework Directory

In this section:

Warrior framework 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, for example, data access, reading, writing, log file creation, parsing, and test results processing.

This Framework directory has two subdirectories: ClassUtils and Utils.

6.2.1 ClassUtils Directory

The ClassUtils directory has subdirectories—WNetwork and WSelenium and several subfiles. All these files are class files that provide specific utilities for objects. Every file has been described in detail below.

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, and getting a node in the tree.

Database_utils_class.py

This module provides database system details from database config file for handling the Mongodb operations to add HTML results, establishing the connection, and adding the case/suite/project results.

Gnmi_utils_class.py

This module performs scp/sftp operations, executes gnmi command using gnmi binary, verifies gnmi output JSON, closes the gnmi streaming or polling, and gets the command string for gnmi operation.

Json_utils_class.py

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

Kafka_utils_class.py

This module performs all Kafka consumer methods, subscribe and unsubscribe to all topics, getting messages from consumer.

Netconf.py

This file was added in as a replacement for the ncclient pre-requisite that Warrior framework had until Version 1.8. Fujitsu recommends that these files need not be modified.

netconf_utils_class.py

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 provides functionalities to get, edit, copy, and delete configuration from the datastore, lock, unlock, validate, and get configuration system, create and wait for subscriptions.

Rest_server_class.py

This module performs terminating and shutting down the server automatically.

Rest_utils_class.py

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

Snmp_utils_class.py

An 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, and checking for Octet string.

ssh_utils_class.py

An SSH utility module performs SSH communication using paramiko, connects to host using SSH object, closes the connection, executes the command on remote host, gets the file from remote path, and downloads a remote file from the remote server to the local path using SFTP.

testdata_class.py

This module has all test data-related class and methods that provide utilities for substituting the [VAR_SUB] patterns, resolving, getting, and validating the iteration patterns, expanding, validating, and verifying iteration patterns provided in each verification search provided for the command and more.

xl_utils_class.py

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

WNetwork Directory

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

Base_class.py

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

connection.py

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

diagnostics.py

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

File_ops.py

The Warrior framework Network File operations module contains functionalities to start FTP and SFTP on the remote host and check if a certain file exists on the remote host.

Loging.py

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

Network_class.py

This class inherits all other classes in the Network package. Instance of this class may be used to invoke the methods of all other classes in this package. When using the instance of this class, if the base classes have the same method name, the method is executed based on Python inheritance Method Resolution Order (MRO). Execute *Network.__mro__* to find out the current MRO. Order is from left to right of the MRO.

Warrior_cli_class.py

This class handles CLI operations. It initiates the SSH and Telnet connection via a specific port and creates a log file.

WSelenium Directory

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

- Browser_mgmt.py
- Element_location.py
- Element_operations.py
- Verify_operations.py
- Wait_operations.py

Browser_mgmt.py

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

Element_location.py

The Selenium element locator library. It provides the utilities for getting elements by name, xpath, link text, partial link, ID, tag name, class name, and CSS selector.

Element_operation.py

The Selenium element operations library. It provides functionalities for getting and clicking an element, performing an element action, sending keys, and clearing text.

Verify_operations.py

The verify operations library with functionalities for verifying HTML elements on a page.

Wait_operation.py

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.

6.2.2 Utils Directory

The Utils directory contains modules that provide various utilities. They have been described briefly in the following sections. The following directory provides the path for Utils Directory.

```
Warriorframework_py3/warrior/Framework/Utils
```

Cli_Utils.py

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.

Config_Utils.py

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

csv_Utils.py

The API related to operations on csv files. It has function that takes the csv file path as input and converts it to list of dictionaries.

data_Utils.py

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. Refer to the warrior_docs repository for full documentation.

datetime_Utils.py

The utility for date time functionalities, for example, getting the current time stamp.

demo_Utils.py

This API is for demo purposes only.

dict_Utils.py

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

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 deprecates in the next release of Warrior framework.

email_Utils.py

Provides utility to send email using SMPT.

encryption_utils.py

This file provides utilities for data encryption.

file_Utils.py

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

import_utils.py

This file provides utilities for package verifications and imports.

list_Utils.py

The library to hold all APIs related to list operations.

print_Utils.py

The Warrior framework print library provides utilities to print outputs on to the console and simultaneously marking them as *Information*, *Error*, *Exception*, and *Debug*.

rest_Utils.py

The REST Utils library 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.

selenium_utils.py

The library to hold all APIs related to selenium operations.

string_Utils.py

The library to hold all APIs related to string operations.

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.

testcase_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. Refer to the full documentation available in the warrior_docs repository.

testrandom_utils.py

This module contains functions for repetitive random testing with variables in xls file.

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, and get child of an XML parent node. Refer to the full documentation available in the warrior_docs repository.

6.3
Productdrivers Directory

The Warrior driver files reside in the Productdrivers directory. Drivers are Python files that create the execution layer of the Warrior framework. The product drivers 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.

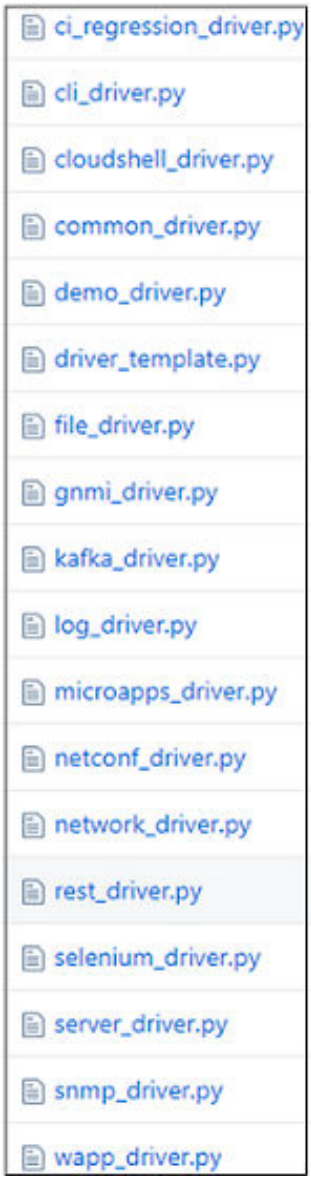


Figure 29
Productdrivers Directory

ci_regression_driver.py

The CI Regression driver is the first directory that a user notices after opening the product drivers directory. The directory imports all actions related to CI regression driver and calls the driver Utils to execute a keyword.

cli_driver.py

This product driver invokes execution of all keyword functions that can be used for automating a command line interface-based system.

cloudshell_driver.py

This product driver invokes execution of all keyword functions related to cloudshell.

common_driver.py

This common driver invokes execution of all the common keyword functions that can be reused among multiple products.

demo_driver.py

This product driver invokes execution of all demo test-related keywords.

driver_template.py

Sample driver is used in creating a new product driver. This driver provides the template to write a new product driver.

file_driver.py

This driver imports all actions related to file driver.

gnmi_driver.py

This gnmi driver is used in network elements.

kafka_driver.py

This driver performs all kafka-related operations.

log_driver.py

This common driver performs logging within test and printing the given message.

microapps_driver.py

This product driver performs actions related to microapp driver and gets the warrior result.

netconf_driver.py

This product driver invokes execution of all keyword functions that can be used for netconf interface.

network_driver.py

This product driver invokes execution of all keyword functions that can be used for all network keywords.

rest_driver.py

This product driver invokes execution of all keyword functions that can be used for REST.

selenium_driver.py

This product driver invokes execution of all keyword functions that can be used for all selenium keywords.

snmp_driver.py

This product driver invokes execution of all SNMP keywords.

wapp_driver.py

This product driver connects Warrior to wapps and executes.

6.4 Tools Directory

In this section:

The tools directory contains directories containing files specific to Warrior frameworks integration with other tools. The following directories and files are contained under these directories.

6.4.1 Admin Directory

Currently, the admin directory consists of the secret key file.

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

6.4.2 Connection 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.

6.4.3 Database Directory

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

6.4.4 JIRA Directory

The JIRA directory contains a jira.xml file that can be configured with information about JIRA projects that a user may upload defects to Warrior after the execution finishes.

6.4.5 Reporting Directory

This directory contains the files needed for the HTML file generation and reporting. These files need not be modified.

6.4.6 XSD Directory

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

6.4.7 w_settings File

The `w_settings` file is used to set up the email notifications and set the directories for log and result files.

6.5
Warrior Core Directory

This directory contains the Python files that compose Warrior framework execution engine. Do not make any changes to the files in this directory.

The following figure shows the core directory structure.

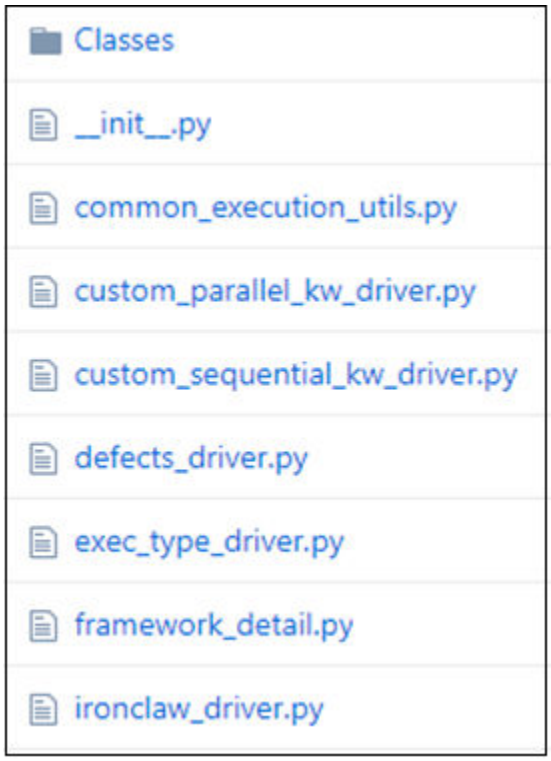


Figure 30
Core Directory

Classes

This directory contains the list of files where it works on executing the cases.

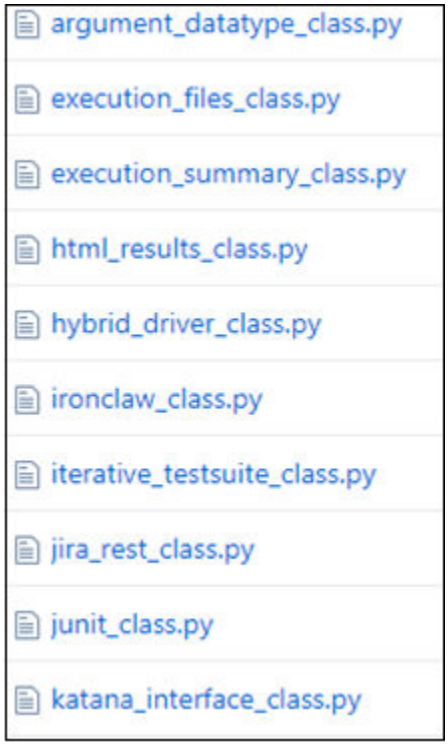


Figure 31
Classes Directory

Argument_datatype_class.py

The argument datatype class API converts the user input arguments in Warrior testcase XML into Python datatypes.

Execution_files_class.py

This class works on executing any file. This class creates the execution directory, retrieves the results directory and files, and retrieves the data files for test case and test suite.

Html_results_class.py

This class generates HTML result line items.

Hybrid_driver_class.py

This class handles hybrid mode test case execution and prints the logs.

Ironclaw_class.py

This class checks the correctness of the XML schema for testcase, testsuite, project XML files of Warrior framework.

Iterative_testsuite_class.py

This class executes the suites iterative parallel and sequential fashion.

Jira_rest_class.py

This class checks the JIRA server for any existing issue with the same summary and updates the JIRA issue with JIRA ID. This class can create JIRA ticket using JIRA rest API.

Junit_class.py

This class creates Junit file and converts to HTML file.

katana_interface_class.py

This Katana interface class is used for getting the location and sending the file.

Kw_driver_class.py

This class gathers the argument information about the keywords, executes the keywords, and reports the keyword status back to the product driver.

Manual_defect_class.py

This class is used for uploading JIRA issues in CLI interface and creating JIRA issues for each failure.

Project_utils_class.py

This class is related to project result file reporting.

testcase_junit_class.py

This class creates testcase Junit file.

testcase_utils_class.py

This class contains methods for test case-related operations and prints the dump of the XML object to the specified file.

War_cli_class.py

This class is used for handling and executing input arguments.

War_print_class.py

This class redirects the message from console to log file

War_mock_class.py

This class is used to mock attributes to input functions and map the commands.

common_execution_utils.py

This module contains common utilities required for execution.

Custom_parallel_kw_driver.py

This driver is custom parallel keyword driver which is used to execute the keywords of a testcase in parallel where data type is custom.

Custom_sequential_kw_driver.py

This driver is custom sequential keyword driver which is used to execute the keywords of a testcase in sequence where data type is custom.

Defects_driver.py

This driver is related to interaction of Warrior framework with JIRA REST API and other related actions.

Exec_type_driver.py

This driver handles the actions to be performed for conditional execution of a step in Testcase/Suite/Project.

Framework_detail.py

This driver gets framework details, for example, the executing framework path, release and version details.

Ironclaw_driver.py

This iron claw driver checks the correctness of the XML schema for testcase, testuite, project XML files of a framework.

Iterative parallel kw driver.py

This iterative parallel keyword driver is used to execute the keywords of a testcase in parallel where data type is iterative.

Iterative_parallel_testcase_driver.py

This iterative parallel testcase driver is used to execute the testcases of a suite in parallel across the systems in the suite data file.

Iterative_sequential_kw_driver.py

This iterative sequential keyword driver is used to execute the keywords of a testcase in sequential order where data type is iterative.

Kw_driver.py

This driver gathers the argument information about the keywords, executes the keywords, and reports the keyword status back to the product driver

Multiprocessing_utils.py

This driver creates Python multiprocesses for the provided target module with the provided arguments and starts them.

onerror_driver.py

This onerror driver handles all the failures in Warrior framework at levels like step/step conditions/testcase/testsuite/project. Returns the actions that should be taken corresponding to the failure.

Parallel_testcase_driver.py

The parallel testcase driver is used to execute the testcases of a suite in parallel.

Parallel_testsuite_driver.py

The parallel suite driver is used to execute the suites of a project in parallel.

Project_driver.py

The project driver executes collections of Warrior testsuites.

Sequential_testcase_driver.py

This sequential testcase driver which is used to execute the testcases of a suite in sequential order.

Sequential_testsuite_driver.py

This sequential suite driver is used to execute the suites of a project in sequential order.

Step_driver.py

This step driver module is used for sending the keyword to corresponding product driver for execution.

Testcase_driver.py

The test case driver is the driver responsible for execution of a testcase. It in turn calls the custom_sequential/ custom_parallel/iterative_sequential/iterative_parallel drivers according to the data type and run type of the testcase.

Testcase_steps_execution.py

This testcase steps execution driver is used for sequential execution of testcase steps.

Testsuite_driver.py

This test suite driver module is used to execute a collection of testcases.

Testsuite_utils.py

This driver can be used for debugging purpose.

Warrior_cli_driver.py

This driver handles all the CLI commands and uploads the log and result file to JIRA, if JIRA ID is provided.

6.6 Warriorspace Directory

In this section:

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

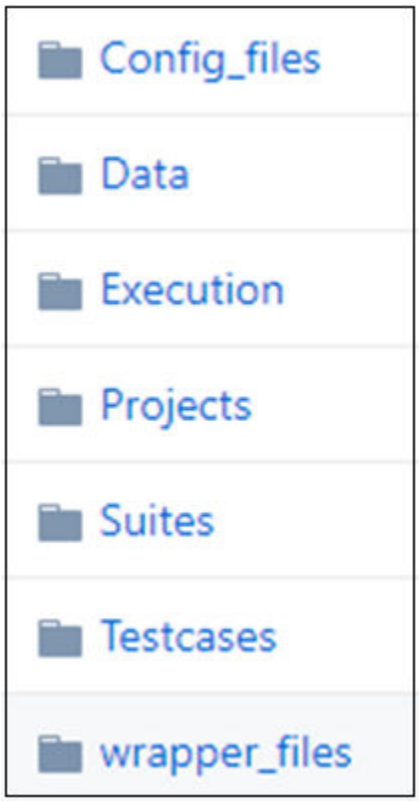


Figure 32
Warriorspace Directory

6.6.1 Config_files Directory

The `Config_files` directory contains the configuration files as indicated in the input data files.

6.6.2
Data Directory

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

6.6.3
Execution Directory

- Warrior stores all the execution-related information like the log and result files in the Execution directory by default.
- The Execution directory is automatically created by Warrior under Warriorspace directory. If the Execution directory already exists, Warrior uses the existing directory.
- If a case, suite or project is being executed for the first time, the Warrior creates a subdirectory inside the Execution folder with the same name as the case, suite, or project, and all the related log and result files are stored inside that directory.
- If a case, suite, or project is being executed for the nth time, Warrior creates as subdirectory named after that case, suite, or project and appends the time/date of the execution. Under this time/date stamped directory, the result and log files are stored.
- To change this default behavior and make Warrior store the execution information in a different directory, a user needs to change the setting for it in the w_settings.xml file.

6.6.4
Projects Directory

This directory includes the project.xml files that define the projects to be executed. This is the default space provided by Warrior framework to save the projects. A user may use a different location.

6.6.5
Suites Directory

This directory includes the suite.xml files that define the suites to be executed. This is the default space provided by Warrior framework to save the suites. A user may use a different location.

6.6.6
Testcases Directory

This directory includes the case.xml files that define the keywords to be executed. This is the default space provided by Warrior framework to save the cases. A user can use a different directory to save the cases.

6.6.7
Wrapper_Files Directory

This directory includes the wrapper files that define the setup, cleanup, and debug section keywords to be executed. This is the default space provided by Warrior framework to save the wrapper files. A user can use a different directory to save the test wrapper files.

7
Creating Case

In this chapter:

A Warrior Case is a file that has a predefined structure established by a Warrior framework. This section gives a general overview of what a Warrior Case is made up of. To correctly design a Warrior Case, refer to Designing Case.

A Case consists of three sections: Details, Steps, Requirements.

7.1 Details

In this section:

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

7.1.1 Name

Note: This field is mandatory.

The Name field contains the case name. It must be unique. The name of the file and the value of this field should match.

7.1.2 Case Title

Note: This field is mandatory.

Add a title or a description for this case so that a user can refer to it later on to understand the purpose of this case.

7.1.3 Engineer

Note: This field is mandatory.

The Case developer name.

7.1.4 Category

Note: This field is an optional field, but it is highly recommended to a user to categorize the cases.

This field is a feature or category. This field is used to group cases quickly. A user can classify the case under any category (like Sanity or Alarm).

7.1.5 Date

Note: This field is an optional field, only defined so that a user can date stamp a case.

This field stores the date of creation of this Case.

7.1.6 Time

Note: This field is an optional field, only defined so that a user can date stamp a case.

This field stores the time of creation of this Case.

7.1.7 State

Note: This is an optional field.

This feature contains the state of the Case, that is, whether the Case is a new, released, test-assigned, or anything else that a user might consider appropriate for the Case. This feature creates a Case when the keywords for this functionality are being simultaneously developed.

7.1.8 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, a No_Data is required. If this is left blank, Warrior expects to find a data file named casename_data.xml in the Data directory of the Warriorspace.

7.1.9 Data Type

This feature 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 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 is used to access the data from the data file. If Hybrid is selected, each step gets to run according to what has been specified against it. For more details on the Hybrid mode, see the *Iteration_type Tag*. The default is *Custom*.

7.1.10
Default On Error

The default error handling for the case. This is where a user sets 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, it is used instead of the default. The Default_OnError options are: Next, Goto, and Abort.

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

The Actions Attribute

An attribute of the *Default_OnError* tag. This attribute is an optional attribute. If not set, Warrior defaults to ***next***, that is, it would proceed to the next Case. The options for this attribute are next, abort, and Goto.

Value Attribute

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 a user wants to jump to in case the current step throws an error. Warrior directly goes to execute the step corresponding to the number given if the current step does not pass.

7.1.11
Logsdir

Note: This is an optional field.

Location of directory where log files are stored. If field is left blank, Warrior uses the default location under the execution folder of the Warrior space.

7.1.12
Resultsdir

Note: This is an optional field.

Location of directory where result files are stored. If field is left blank, Warrior uses what is in the case, if that has been left blank, Warrior uses the location defined in the w_setting.py file, if that is left blank, Warrior uses the default under the Execution directory.

7.1.13
Expected Results

Note: This is an optional field.

The ExpectedResults tag is for user benefit, so that the user knows if a particular case is supposed to Pass or Fail or Error out.

7.1.14
Testwrapper File

The Testwrapper file tag is used to choose which wrapper files need to execute.

7.2 Requirements

In this section:

A user can add as many requirements as needed. These requirements are for informational purposes only. The requirements entered in the case are shown in the XML result file.

7.2.1 Requirement

Note: This is an optional field.

Every requirement needed for this case must be added inside the requirements section.

7.3 Steps

In this section:

7.3.1 Step

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

7.3.2 Arguments

Section within the step where the arguments are detailed.

Argument

The argument tag is where user 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, when 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.

7.3.3 On Error

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

1. Selecting Next results 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 a user has already logged on to a website and wants to make rest calls to its API and verify the data received from the rest calls. To achieve this, user has

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 preceding case, 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***.

2. Selecting Goto and the step number to Goto results in execution of the step indicated. The execution of steps continues from that step going forward.

a. When the Onerror is set to GoTo, a user 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 the user is creating a Case that updates the software of a system whenever run. When such sensitive Cases are developed, it is necessary to have a restore facility handy in case something goes wrong when updating the software. 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.

3. Selecting Abort terminates 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 the user is connecting to CLI-based system using Warrior. The first keyword would be connecting to the system and the rest of the commands would follow. 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***.

4. Selecting Abort_as_error terminates the execution of the case if the step does not pass. The step status would be set to Error.

7.3.4
Description

Note: This step is an optional field, but it is highly recommended that a user fill it out.

Text description for each step can be specified within this tag.

7.3.5 Execute Type

This field is an optional field that decides when a step should be executed. If this field is not set, the default executes the keyword. The Execute Type attribute has four options: If, If Not, Yes, No.

Creating a step which gets executed conditionally is explained in this section.

- When the execute type for a step is set to ***Yes***, the step gets executed no matter what.
- 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, user may not want to run some steps every time user runs that Case.
- When the execute type for a step is set to ***No***, Warrior executes this step only if a certain condition is met; otherwise, it skips this step and either proceed to the next step or proceed to a specified step or abort execution of the Case. After the execute type has been set to ***If***, user 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 is available in Warrior data repository.
    - If the user wants 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 user wants to execute this step.
    - If user wants to execute this step based on a value of a key available in Warrior data repository, user needs to add the key as it would be found in the data repository as the condition and its value as user expects it to be as the condition value.
    - An ***else*** for this ***if*** is available in case the condition is not met. The value set in this else lets Warrior know what to do in case the condition is not met. User has the option of continuing the execution by executing the next step, or executing a particular step, or aborting the execution of the Case.
- When the execute type for a step is set to ***No***, Warrior executes this step only if a certain condition is NOT met; otherwise, it skips this step and either proceed to the next step or proceed to a specified step or abort execution of the Case. After the execute type has been set to “If Not”, user 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 is available in Warrior data repository.

- If user wants 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, depending upon when the user wants to execute this step, can be:
  - PASS
  - FAIL
  - ERROR
  - SKIPPED
- If user wants to execute this step based on a value of a key available in Warrior data repository, user needs to add the key as it would be found in the data repository as the condition and its value as user expects it to be as the condition value.
- An ***else*** for this ***if*** is available in case the condition is met. The value set in this else lets Warrior know what to do in case the condition is met. User has the option of continuing the execution by executing the next step, or executing a particular step, or aborting the execution of the Case.

7.3.6 Iteration Type

This tag is required only when the Hybrid mode is selected. This tag 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 after per case when the *once_per_iter* mode lets that particular step run for every iteration that the set of steps go through.

7.3.7 Context

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

- Every step created is, by default, in a positive context, that is, the end result of that step is reported as is to Warrior. If the step fails, the end result is reported as a failure. If it passes, the end result is reported as a pass.
- Creating a step in the positive context in Katana is explained here.
- A step can be evaluated in the negative context, that is, the end result of that step is flipped and reported to Warrior. If the step fails, the end result is reported as a pass. If it passes, the end result is reported as a fail.

Example: If user knows that a failure is expected as the result of executing a step 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) is evaluated as a PASS. Note that the end result of a *PASS* is evaluated as a *FAIL*.

7.3.8 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.

- Every step created is, by default, an impacting step, that is, 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.
- Creating an impacting step in Katana is explained here.
- A step can be marked as a non-impacting step, that is, 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.

7.3.9 Run Mode

Note: This is an optional tag.

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, user can specify either 'RMT', 'RUF', or 'RUP', and in 'value', user can specify the maximum number of times the step needs to run.

- When the Runmode for a step is set to RMT, user has to provide the number of times that step should run. Warrior executes this step those many number of times. If user does not provide Warrior with the number of times the step should be run, this step would run once.
- When the Runmode for a step is set to RUF, user has to provide the maximum number of times that step should run. Warrior keeps on executing this step as long it passes and stops 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.
- When the Runmode for a step is set to RUP, user has to provide the maximum number of times that step should run. Warrior would then keep on executing this step as long it fails and stops 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 is marked as FAIL even though the step eventually passes—this is to alert the Case developer that somewhere a failure has occurred.

```
<?xml version="1.0" ?>
<Testcase>
        <Details>
                <Name>Name</Name>
                <Title>Title</Title>
                <ExpectedResults>ExpectedResults</ExpectedResults>
                <Category>Feature</Category>
                <Engineer>Warrior_user</Engineer>
                <Date>2020-03-11</Date>
                <Time>22:06</Time>
                <default_onError action="goto" value="9"/>
                <InputDataFile>Path_to_InputDataFile</InputDataFile>
                <Datatype>Custom</Datatype>
                <Logsdir/>
                <Resultsdir/>
                <State>New</State>
        </Details>
        <Requirements>
                <Requirement>requirement-001</Requirement>
        </Requirements>
        <Steps>
                <step Driver="DriverName" Keyword="KeywordName" TS="2">
                        <Execute ExecType="Yes">
                                <Rule Condition="" Condvalue="" Else="" Elsevalue=""/>
                        </Execute>
                        <onError action="next" value=""/>
                        <Description/>
                        <Iteration_type type="Standard"/>
                        <context>positive</context>
                        <impact>impact</impact>
                        <runmode type="Standard" value=""/>
                </step>
        </Steps>
</Testcase>
```

Figure 33
Testcase

8
Creating Suite

In this chapter:

Through CLI

A user 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 set up 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 is created, default=cwd

-cat (optional): 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. A suite file is shown in the following figure.

Figure 34
Suite File

Through Katana

To create a Suite through Katana, refer Katana documentation.

A Warrior Suite is a file that has a predefined 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. A Suite can be created using Katana.

Three sections exist in a Suite: Details, Cases, and Requirements. Each section has been described in detail in the following sections.

8.1 Details

In this section:

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

8.1.1 Name

Note: This field is mandatory.

The Name field contains the Suite name. It must be unique. The name of the file and the value of this field should match.

8.1.2 Title

Note: This field is mandatory.

Add a title or a description for this case so that a user can refer to it later on to understand the purpose of this case.

8.1.3 Engineer

Note: This field is mandatory.

This field stores the Suite developer name.

8.1.4 Date

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

8.1.5 Time

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

8.1.6 Execute 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.

Note: 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) are 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) are 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) are executed until the suite Pass or the max attempts have been reached.

8.1.7 State

Note: This is an optional field.

This field contains the state of the Suite, that is, whether the Suite is a new, released, test-assigned, or anything else that user might consider appropriate for the Case.

8.1.8 On Error (Default)

Default error handling for the suite. This is where user sets 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 is used instead of the default.

The Actions Attribute

Note: This is an optional attribute.

An attribute of the *Default_OnError*. If not set, Warrior default to the *next*, that is, it would proceed to the next case. User is able to set different onError setting per case if needed. That is done at the case level. The options for this attribute are next, abort, and goto.

The Value Attribute

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 user wants to jump to in case the current case throws an Error. Warrior directly goes to executing the case corresponding to the number given if the current case throws an Error.

8.1.9 Input Data File

Note: This is an optional field.

Path to Data file associated with the suite. If a data file is listed, that data file overwrites the data file used in the cases in the suite. If a data file is not listed, the data files within the cases are used.

8.1.10 Testwrapper File

Add the test wrapper file where a user has created to check the setup, cleanup, and debug steps.

8.1.11 Results Directory

Note: This is an optional field.

Location of directory where result files are stored here. If field is left blank, Warrior uses what is in the case; if field is left blank, Warrior uses the location defined in the w_setting.py file; if field is left blank, Warrior uses the default under the Execution directory.

8.2
Requirements

In this section:

8.2.1
Requirement

Note: This is an optional field.

Every requirement needed for this case must be added inside a `<Requirement></Requirement>`.

8.3 Cases

In this section:

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

8.3.1 Case Path

Note: This field is mandatory.

This field is for the location of the case. A user can enter a relative path.

8.3.2 Data File

Enter the data file path which was created as input data file.

8.3.3 Run Type

This run type field determines if the sequential_keywords or parallel_keywords.

Note: Default is sequential_keywords.

If the sequential_keywords is selected, the keywords (or steps) within the case run in sequence.

If the parallel_keywords is selected, the keywords (or steps) within the case run in parallel.

8.3.4 Execute

Note: This is an optional field that decides when a Case should be executed.

If this field is not set, the default value executes the case. The Execute Type attribute has four options: If, If Not, Yes, No.

If: The case is executed if the condition below is met, if the condition is not met, the Else action is executed.

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

Yes: The case is executed without any conditions.

No: The case is not executed.

8.3.5 Run Mode

Note: This is an optional field.

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.

8.3.6 Context

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

Note: The default is *positive*.

8.3.7 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.

8.3.8 On Error

Note: This is an optional field.

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

- Selecting Next results in the next case being executed if the case fails.
- Selecting Goto and the case number to Goto result in execution of the case indicated. The execution of cases continues from that case going forward.

- Selecting Abort terminates the execution of the suite if the case does not pass. The case status would be set to FAIL
- Selecting Abort_as_error terminates the execution of the suite if the case does not pass. The case status would be set to ERROR.

9
Creating Project

In this chapter:

A Warrior Project is a file that has a pre-defined structure established by a 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. Refer to *Complete Guide to Creating Project using Katana in Katana User Guide*.

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

9.1 Details

In this section:

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

9.1.1 Name

Note: This field is mandatory.

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

9.1.2 Project Title

Note: This field is mandatory.

The project title or description. User can describe what this project does and what all kind of suites it contains in this field.

9.1.3 Engineer

Note: This field is mandatory.

The name of the Project designer.

9.1.4 State

Note: This is an optional field.

The state of the Project, that is, whether the Project is a new, released, test-assigned, or anything else that user might consider appropriate for the Case.

9.1.5 Date

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

9.1.6 Time

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

9.1.7 Default On Error

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, it is used instead of the default.

The Actions Attribute

An attribute of the default_OnError. This is an optional attribute. If not set, Warrior default to the *next*, that is, it would proceed to the next suite. User can set different onError setting per suite if needed. That is done at the suite level. The options for this attribute are next, abort, and goto.

The Value Attribute

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 user wants to jump to in case the current suite throws an Error. Warrior directly executes the suite corresponding to the number given if the current suite throws an error.

9.1.8 Results Directory

By providing the path of the results directory can save all results in one particular directory path.

9.2 Suite

In this section:

A user can add as many suites in a project, but at least one suite is required.

9.2.1 Path

Note: This field is mandatory.

This section contains information about the location of the suite. Enter a relative path.

9.2.2 On Error

Note: This is an optional field.

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

- Selecting Next results in the next suite being executed if the suite fails.
- Selecting Goto and the suite number to Goto result in execution of the suite indicated. The execution of suites continues from that suite going forward.
- Selecting Abort terminates the execution of the suite if the suite does not pass. The suite status would be set to FAIL.
- Selecting Abort_as_error terminates the execution of the suite if the step does not pass. The suite status would be set to ERROR.

9.2.3 Impact

This determines if the failure of this suite impacts the overall pass/fail status of the project. Setting it to impact causees the project to fail if the suite fails. Setting it to no impact does not cause the project to fail if the suite fails.

Note: The default is *impact*.

9.2.4
Execute Type

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

If: The Suite is executed if the following condition is met, if the condition is not met, the Else action is executed.

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

Yes: The Suite is executed without any conditions.

No: The Suite need not be executed.

The following figure shows the project file.

```
<?xml version="1.0" ?>
<Project>
        <Details>
                <Name>Demo_Project</Name>
                <Title>Demo_Project</Title>
                <Engineer>Warrior_user</Engineer>
                <State>In Review</State>
                <Date>05/11/2020</Date>
                <Time>21:40:17</Time>
                <default_onError action="abort" value=""/>
        </Details>
        <Testsuites>
                <Testsuite>
                        <path>../Suites/demo_suite.xml</path>
                        <onError action="abort" value=""/>
                        <impact>impact</impact>
                        <Execute ExecType="Yes">
                                <Rule Condition="" Condvalue="" Else="next" Elsevalue=""/>
                        </Execute>
                </Testsuite>
        </Testsuites>
</Project>
```

Figure 35
Project File

10
Creating Input Data File

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>`. 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 name can be added as attribute name and value in the `<subsystem></subsystem>` tags.

This data file is provided as the input data file in Warrior Case.

This data files supports the following items:

- 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 data file.

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.

For example, `${ENV.IPADDR}` is replaced with the value of `IPADDR` variable in the environment settings.

To create a datafile in Katana, refer to *Katana User Guide*.

The following figure shows the typical data file.

```
<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>
```

Figure 36
Input Data File

11
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 is stored in the case, suite, or project data repository as a key-value pair.
- Keywords have to access the case data repository to get the data returned from a previous keyword, cases in a suite have access to case statuses returned from the previous cases, and suites in a project 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. To use these values in the next keyword, a user 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')
  ```

12
Designing Wrapper File

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

Wrapper file has details, setup, cleanup and debug sections.

Details Section	■ User can specify Name, Title, Date, Time, Engineer ■ Datatype—custom/iterative/hybrid ■ 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.
Debug	Contains all the debug steps which are executed upon test case and test suite failure.

```
<?xml version="1.0" ?>
<TestWrapper>
  <Details>
      <Name>tw_sample</Name>
      <Title>connect and disconnect all systems and subsystems in the data file</Title>
      <Datatype>Custom</Datatype>
   <Runtype>Sequential_keywords</Runtype>
   <Date>2020-05-11</Date>
      <Time>11:20</Time>
      <Engineer>Warrior_User</Engineer>
   </Details>
  <Setup>
      <step Driver="cli_driver" Keyword="connect_all" TS="1">
          <Arguments>

          </Arguments>
          <onError action="next"/>
          <Description>connect</Description>
          <Execute ExecType="Yes"/>
          <context>positive</context>
          <impact>impact</impact>
   </step>
  </Setup>
  <Cleanup>
      <step Driver="cli_driver" Keyword="disconnect_all" TS="8">
          <Arguments>

          </Arguments>
          <onError action="next"/>
          <Description>connect</Description>
          <Execute ExecType="Yes"/>
          <context>positive</context>
          <impact>impact</impact>
      </step>
```

Figure 37
Testwrapper File—Part 1

```
  </Cleanup>
   <Debug>
      <step TS="1" Driver="cli_driver" Keyword="send_commands_by_testdata_title">
          <Description>None</Description>
          <Execute ExecType="Yes">
          </Execute>
          <runmode type="standard"></runmode>
          <Iteration_type type="standard"></Iteration_type>
          <context>positive</context>
          <impact>impact</impact>
          <onError action="next"></onError>
          <Arguments>
              <argument name="title" value="valid-single-no_verifications"></argument>
              <argument name="system_name" value="server1[interface1]"></argument>
          </Arguments>
      </step>
   </Debug>
</TestWrapper>
```

Figure 38
Testwrapper File—Part 2

13
Designing Case

In this chapter:

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, 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 different ways to design the case to suit the requirements.

13.1
Design 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.

Step 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.

Step 2
Set the ***Data Type*** field in the Details section of a Case to *Iterative*.

Note: 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.

```
<?xml version="1.0" encoding="utf-8"?>
<Testcase>
    <Details>
        <Name>CLI_Sample</Name>
        <Title>cli sample testcase</Title>
        <Category>Iterative</Category>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>2020-07-14</Date>
        <Time>21:48</Time>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../../Data/Samples/CLI_Data_Sample.xml</InputDataFile>
        <TestWrapperFile>../../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Datatype>Iterative</Datatype>
        <Logsdir>/home/logs</Logsdir>
        <Resultsdir>/home/results</Resultsdir>
        <ExpectedResults></ExpectedResults>
    </Details>
    <Requirements></Requirements>
    <Steps>
        <step TS="1" Repo="warrior" Driver="cli_driver" Keyword="connect">
            <Description>connect_ssh</Description>
            <Execute ExecType="Yes"></Execute>
            <runmode type="standard"></runmode>
            <Iteration_type type="standard"></Iteration_type>
            <context>positive</context>
            <impact>impact</impact>
            <onError action="next"></onError>
            <Arguments></Arguments>
        </step>
    </Steps>
</Testcase>
```

Figure 39
Iterative Case

✓ **This task is complete.**

13.2
Design 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.

Step 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.

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

Note: It is required to enter the system name in the arguments to the keywords in all the steps as all the steps would be executed on that particular system.

Figure 40
Custom Case

✓ **This task is complete.**

13.3
Design Hybrid Case

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

Step 1
Change the ***Data Type*** field in the Details section of a Case to *Hybrid*.

Note: Every step in that Case would have a new field *Iteration Type*.

Step 2
If user wants to run a particular step as a standard iterative step, set the *Iteration Type* to *standard*.

Step 3
If user wants to run a particular step only after throughout the case, set the *Iteration Type* to *once_per_tc*. If user wants to run a particular step after all the other steps in a case have been executed, set the *Iteration Type* to *end_of_tc*. If multiple *end_of_tc* steps exist in a case, all of them are executed at the end and the order in which they would be executed would be the order in which they appear in a case.

Figure 41
Hybrid Case

✓ **This task is complete.**

13.4
Designing Draft Case

This feature lets the user to create a case when the keywords for this functionality are being simultaneously developed. A user can create a regular case and make it as a Draft Case if the user does not want anyone to run this case.

13.5
Design Case without using Input Data File

A custom case does not require an Input Data File. So, if a user does not want to add a data file into the case, the user has 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.

Step 1
Make the ***Input Data File*** field as blank.

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

Note: All the necessary values to the arguments of a keyword would have to be passed through the Case.

```
<Testcase>
    <Details>
        <Name>CLI_Sample</Name>
        <Title>cli sample testcase</Title>
        <Category>custom</Category>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>2020-07-14</Date>
        <Time>21:48</Time>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile></InputDataFile>
        <TestWrapperFile>../../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Datatype>Custom</Datatype>
        <Logsdir>/home/logs</Logsdir>
        <Resultsdir>/home/results</Resultsdir>
        <ExpectedResults></ExpectedResults>
    </Details>
    <Requirements></Requirements>
    <Steps>
        <step TS="1" Repo="warrior" Driver="cli_driver" Keyword="connect">
            <Description>connect_ssh</Description>
            <Execute ExecType="Yes"></Execute>
            <runmode type="standard"></runmode>
            <Iteration_type type="standard"></Iteration_type>
            <context>positive</context>
            <impact>impact</impact>
            <onError action="next"></onError>
            <Arguments>
                <argument name="system_name" value="NE1"></argument>
            </Arguments>
        </step>
    </Steps>
</Testcase>
```

Figure 42
Case without Input Data File

✓ **This task is complete.**

13.6 Designing Conditional Case

In this section:

13.6.1 Execute Type: Yes

When the execute type for a Case is set to *yes*, the Case is executed.

This procedure describes how to set the Execute Type of a Case to *Yes*.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the Execute Type to *Yes*.

Figure 43
Execute Type: Yes

✓ **This task is complete.**

13.6.2 Execute Type: No

When the execute type for a Case is set to *no*, the step would not be executed under any circumstances. This option exists because when running a Suite, a user need not want to run some cases every time in that Suite.

This procedure describes how to set the Execute Type of a Case to *No*.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the Execute Type to *No*.

Figure 44
Execute Type: No

✓ **This task is complete.**

13.6.3
Execute Type: If

This condition is the programming equivalent of an *If* statement in Warrior.

If a certain condition is met:

- Execute this Case
- Otherwise, skip this Case

and either:

- Proceed to the next Case
- Or, proceed to a specified Case
- Or, abort execution of the Suite

After the execute type has been set to *If*, user 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 case based on the result of a previously executed Case, the condition would be `testcase_<case-number>_status` and the value for this condition, depending upon when the user wants to execute this Case, can be:

- PASS
- FAIL
- ERROR
- or SKIPPED

An *else* for this *if* is available in case the condition is not met. The value set in this else lets Warrior know what to do in case the condition is not met. A user has the option of continuing the execution by executing the next Case, or executing a particular Case, or aborting the execution of the Suite.

This procedure describes how to set the Execute Type of a Case to *If*.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the Execute Type to *If*.

Step 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 set Else

```
<Testcase>
    <Details>
        <Name>CLI_Sample</Name>
        <Title>cli sample testcase</Title>
        <Category>custom</Category>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>2020-07-14</Date>
        <Time>21:48</Time>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../../Data/Samples/CLI_Data_Sample.xml</InputDataFile>
        <TestWrapperFile>../../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Datatype>Custom</Datatype>
        <Logsdir>/home/logs</Logsdir>
        <Resultsdir>/home/results</Resultsdir>
        <ExpectedResults></ExpectedResults>
    </Details>
    <Requirements></Requirements>
    <Steps>
        <step TS="1" Repo="warrior" Driver="cli_driver" Keyword="connect">
            <Description>connect_ssh</Description>
            <Execute ExecType="If">
                <Rule Condition="step_1_result" Condvalue="Pass" Else="next" Elsevalue=""></Rule>
            </Execute>
            <runmode type="standard"></runmode>
            <Iteration_type type="standard"></Iteration_type>
            <context>positive</context>
            <impact>impact</impact>
            <onError action="next"></onError>
            <Arguments><argument name="system_name" value="NE1"></argument></Arguments>
        </step>
    </Steps>
</Testcase>
```

Figure 45
Execute Type: If

✓ **This task is complete.**

13.6.4 Execute Type: If Not

This condition 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 either:

- Proceed to the next Case
- Or, proceed to a specified Case
- Or, abort execution of the Suite

After the execute type has been set to *If Not*, user 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, depending upon when the user wants to execute this Case, can be:

- PASS
- FAIL
- ERROR
- or SKIPPED

An *else* for this *if* is available in case the condition is met. The value set in this else lets Warrior know what to do in case the condition is met. A user has the option of continuing the execution by executing the next Case, or executing a particular Case, or aborting the execution of the Suite.

This procedure describes how to set the Execute Type of a Case to *If Not*.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the Execute Type to *If Not*.

Step 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 set Else

Figure 46
Execute Type: If Not

✓ **This task is complete.**

13.7

Design Steps in Case to Run in Sequential

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

Step 1
Create a case that contains the keywords that need to be run in sequential.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the runtype tag to *sequential_keywords*.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="No"></Execute>
            <runmode type="rmt" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="next"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 47
Run Case Steps Sequentially

✓ **This task is complete.**

13.8
Design Steps in Case to Run in Parallel

Warrior keywords defined within a case can be executed in parallel. This procedure describes how to run the keywords in parallel.

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

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the runtype tag to *parallel_keywords*.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>parallel_keywords</runtype>
            <Execute ExecType="No"></Execute>
            <runmode type="rmt" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="next"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 48
Run Case Steps in Parallel

✓ **This task is complete.**

13.9
Design Case to Run Multiple Times

When the Runmode for a Case is set to *RMT*, user has to provide the number of times that step should run. Warrior would execute this Case those many number of times. If the user does not provide Warrior with the number of times the Case should be run, 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 is executed or not depend on the kind of execute type set. All attempts report the status back depending upon the context and all attempts can either be impacting or non-impacting.

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

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

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the Case created in Step 1.

Step 4
Set the ***Run Mode*** field to *RMT*.

Step 5
Set the Max Attempts for RMT.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="Run Until Pass" Max_Attempts="2"></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="Yes"></Execute>
            <runmode type="rmt" value="2" runmode_timer="4"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="goto" value="1"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 49
Run Case Multiple Times

✓ **This task is complete.**

13.10
Design Case to Run Until it Fails

When the Runmode for a Case is set to *RUF*, user has to provide the maximum number of times that Case should run. Warrior would keep on executing this Case as long as it passes and stops 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 the execution of this Case is stopped to proceed to the execution of the next Case.

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

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

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

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the Case created in Step 1.

Step 4
Set the ***Run Mode*** field to *RUF*.

Step 5
Set the Max Attempts for RUF.

Figure 50
Run Case Until Fail

✓ **This task is complete.**

13.11
Design Case to Run Until it Passes

When the Runmode for a Case is set to *RUP*, user has to provide the maximum number of times that Case should run. Warrior would keep on executing this Case as long as it fails and stops 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 the execution of this Case is stopped to proceed to the execution of the next Case.

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

Such 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 is 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.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the Case created in Step 1.

Step 4
Set the ***Run Mode*** field to *RUP*.

Step 5
Set the Max Attempts for RUP.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="Run Until Pass" Max_Attempts="2"></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="Yes"></Execute>
            <runmode type="rup" value="2" runmode_timer="4"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="goto" value="1"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 51
Run Case Until Pass

✓ **This task is complete.**

13.12
Design Impacting Case

Every Case creates, by default, an impacting Case, that is, 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.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the Case created in Step 1.

Step 4
Set the ***Impact*** field to *impact*.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>parallel_keywords</runtype>
            <Execute ExecType="No"></Execute>
            <runmode type="rup" value="2"></runmode>
            <context>positive</context>
            <impact>impact</impact>
            <onError action="next"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 52
Impacting Case

✓ **This task is complete.**

13.13
Design Non-Impacting Case

Prerequisites:

A Case can be marked as a non-impacting Case, that is, 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.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the Case created in Step 1.

Step 4
Set the ***Impact*** field to *noimpact*.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>parallel_keywords</runtype>
            <Execute ExecType="No"></Execute>
            <runmode type="rup" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="next"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 53
Non-Impacting Case

✓ **This task is complete.**

13.14
Design Case in Positive Context

Prerequisites:

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

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

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the Case created in Step 1.

Step 4
Set the ***Context*** field to *positive*.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>parallel_keywords</runtype>
            <Execute ExecType="No"></Execute>
            <runmode type="rup" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="next"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 54
Positive Context Case

✓ **This task is complete.**

13.15
Design Case in Negative Context

Prerequisite:

Negative cases are cases that are expected to fail. If a negative case fails, the cases 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.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the ***Context*** field to *Negative*.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>parallel_keywords</runtype>
            <Execute ExecType="No"></Execute>
            <runmode type="rup" value="2"></runmode>
            <context>negative</context>
            <impact>noimpact</impact>
            <onError action="next"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 55
Negative Context Case

✓ **This task is complete.**

13.16 On Error Actions for Cases

In this section:

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 action prevents the remaining Cases from running and thus unnecessary failures can be avoided.

To create an aborting Case, implement the following steps.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the ***On Error*** field to *abort*.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>parallel_keywords</runtype>
            <Execute ExecType="No"></Execute>
            <runmode type="rup" value="2"></runmode>
            <context>negative</context>
            <impact>noimpact</impact>
            <onError action="abort"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 56
On Error Actions for Case: Abort

✓ **This task is complete.**

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.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the ***On Error*** field to *next*.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>parallel_keywords</runtype>
            <Execute ExecType="No"></Execute>
            <runmode type="rup" value="2"></runmode>
            <context>negative</context>
            <impact>noimpact</impact>
            <onError action="next"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 57
On Error Actions for Case: Next

✓ **This task is complete.**

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.

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the ***On Error*** field to *abort_as_error*.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>parallel_keywords</runtype>
            <Execute ExecType="No"></Execute>
            <runmode type="rup" value="2"></runmode>
            <context>negative</context>
            <impact>noimpact</impact>
            <onError action="abort_as_error"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 58
On Error Actions for Case: Abort_as_error

✓ **This task is complete.**

13.16.4 On Error: Goto

When the On Error is set to Goto, the user would need to provide a Case number to Warrior. The On Error of Goto means that, if a failure is observed, Warrior would jump to execute the mentioned Case.

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

Step 1
Create a case containing the keywords.

Step 2
Create a suite or modify an existing one.

Step 3
In the case block of the suite, add the case created in Step 1.

Step 4
Set the ***On Error*** field to *goto*.

Step 5
Specify the case number.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements>
    </Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="Yes">
            </Execute>
            <runmode type="rmt" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="goto" value="1"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 59
On Error Actions for Case: Goto

✓ **This task is complete.**

14
Designing Case for Common Setup and Cleanup

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

Test wrapper file can be included in testcase as mentioned in <Details> section with <TestWrapperFile> tag.

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

When Test wrapper file is included in testcase Details section, test case execution follows the given order.

1. Set up steps (from test wrapper file <Setup> block).
2. Testcase steps (steps mentioned in <Steps> block).
3. Cleanup steps (from test wrapper file <Cleanup> block).
4. Debug steps (from test wrapper file <Debug> block).

```
<Testcase>
    <Details>
        <Name>CLI_Sample</Name>
        <Title>cli sample testcase</Title>
        <Category>custom</Category>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>2020-07-14</Date>
        <Time>21:48</Time>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../../Data/Samples/CLI_Data_Sample.xml</InputDataFile>
        <TestWrapperFile>../../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Datatype>Custom</Datatype>
        <Logsdir>/home/logs</Logsdir>
        <Resultsdir>/home/results</Resultsdir>
        <ExpectedResults></ExpectedResults>
    </Details>
    <Requirements></Requirements>
    <Steps>
        <step TS="1" Repo="warrior" Driver="cli_driver" Keyword="connect">
            <Description>connect_ssh</Description>
            <Execute ExecType="If Not">
                <Rule Condition="step_1_result" Condvalue="Pass" Else="next" Elsevalue=""></Rule>
            </Execute>
            <runmode type="standard"></runmode>
            <Iteration_type type="standard"></Iteration_type>
            <context>positive</context>
            <impact>impact</impact>
            <onError action="next"></onError>
            <Arguments><argument name="system_name" value="NE1"></argument></Arguments>
        </step>
    </Steps>
</Testcase>
```

Figure 60
Case for Common Setup and Cleanup Steps

15
Designing Suite

In this chapter:

15.1 Designing Cases in Suite

In this section:

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

15.1.1 In Sequence

Warrior Cases defined within a suite can be executed in sequence. This procedure describes how to run the case in sequence.

Step 1
Create the case that needs to run in sequence.

Step 2
Create a suite or modify an existing one.

Step 3
Set the type tag in the suite to *Sequential*.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="sequential_testcases" Number Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements>
    </Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="Yes">
            </Execute>
            <runmode type="rmt" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="goto" value="1"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 61
Run Suite Case in Sequence

Step 4
Execute the suite.

Step Result:
The cases run in sequence.

✓ **This task is complete.**

15.1.2 In Parallel

Warrior Cases defined within a suite can be executed in parallel. This procedure describes how to run the case in parallel.

Step 1
Create the case that needs to run in parallel.

Step 2
Create a suite or modify an existing one.

Step 3
Set the type tag in the suite to *Parallel*.

Step 4
In the case block of the suite, add the case created in Step 1.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="parallel_testcases" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements>
    </Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="Yes">
            </Execute>
            <runmode type="rmt" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="goto" value="1"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 62
Run Suite Case in Parallel

Step 5
Execute the suite.

Step Result:
The cases run in parallel.

✓ **This task is complete.**

15.1.3 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 two cases and the Data File contains three systems with the *iterative_sequential* functionality, the cases would be executed as:

- Case 1 would be executed on system 1
- Case 2 would be executed on system 1
- Case 1 would be executed on system 2
- Case 2 would be executed on system 2
- Case 1 would be executed on system 3
- Case 2 would be executed on system 3

This procedure describes how to run the case in iterative_sequentially.

Step 1
Create the cases that needs to run in iterative_sequentially.

Step 2
Create a suite or modify an existing one.

Step 3
Set the type tag in the suite to *iterative_sequential*.

Step 4
Add the Input Data File in the ***Input Data File*** field in *Details* section of the suite. This Input Data File is used to run the Cases iterative_sequentially.

Step 5
In the case block of the suite, add the cases created in Step 1.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="iterative_sequential" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements>
    </Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="Yes">
            </Execute>
            <runmode type="rmt" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="goto" value="1"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 63
Run Suite Case Iteratively and in Sequence

Step 6
Execute the suite.

Step Result:
The cases run in iterative_sequentially.

✓ **This task is complete.**

15.1.4 Iteratively and in Parallel

Warrior Cases defined within a suite can be executed on each system given in a data file iteratively and in parallel. That is, if a Suite contains two cases and the Data File contains three systems with the *iterative_parallel* functionality, the Cases would be executed as:

- Case 1 would be executed on system 1, system 2, and system 3 simultaneously
- Case 2 would be executed on system 1, system 2, and system 3 simultaneously

This procedure describes how to run the case in iterative_parallel.

Step 1
Create the cases that needs to run in iterative_parallel.

Step 2
Create a suite or modify an existing one.

Step 3
Set the type tag in the suite to iterative_parallel.

Step 4
Add the Input Data File in the ***Input Data File*** field in *Details* section of the suite. This Input Data File is used to run the cases iterative_sequentially.

Step 5
In the case block of the suite, add the cases created in Step 1.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="iterative_parallel" Number_Attempts=""></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements>
    </Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="Yes">
            </Execute>
            <runmode type="rmt" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="goto" value="1"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 64
Run Suite Case Iteratively and in Parallel

Step 6
Execute the suite.

Step Result:
The cases run in iterative_sequentially.

✓ **This task is complete.**

15.1.5
Run Multiple Times

Warrior Cases defined within a suite can be executed multiple times. This procedure describes how to run the case multiple times.

Step 1
Create a suite or modify an existing one.

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

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="run_multiple" Number_Attempts="2"></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements>
    </Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="Yes">
            </Execute>
            <runmode type="rmt" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="goto" value="1"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 65
Run Suite Case Multiple Times

Step 3
Execute the suite.

Step Result:
The cases in the suite run multiple times as indicated in the Max_Attempts attribute.

✓ **This task is complete.**

15.1.6
Run Until Failure

Warrior Cases defined within a suite can be executed until failure. This procedure describes how to run the case multiple times until failure.

Step 1
Create a suite or modify an existing one.

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

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="Run Until Fail" Max_Attempts="2"></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="Yes"></Execute>
            <runmode type="ruf" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="goto" value="1"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 66
Run Suite Case Until Failure

Step 3
Execute the suite.

Step Result:
The cases in the suite run until the suite fails or the suite is executed as indicated in the Max_Attempts attribute.

✓ **This task is complete.**

15.1.7 Run Until Pass

Warrior Cases defined within a suite can be executed until pass. This procedure describes how to run the case multiple times until pass.

Step 1
Create a suite or modify an existing one.

Step 2
Set the type tag in the suite to *Run_Until_Pass* and type in the value for maximum number of attempts.

```
<?xml version="1.0" encoding="utf-8"?>
<TestSuite>
    <Details>
        <Name>common_action_suite</Name>
        <Title>common_action_suite</Title>
        <State>New</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/15/2020</Date>
        <Time>16:37:44</Time>
        <type exectype="Run Until Pass" Max_Attempts="2"></type>
        <default_onError action="next" value=""></default_onError>
        <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
        <TestWrapperFile>../wrapper_files/tw_sample.xml</TestWrapperFile>
        <Resultsdir>/home/results</Resultsdir>
    </Details>
    <Requirements></Requirements>
    <Testcases>
        <Testcase TS="1">
            <path>../../testcases/common_actions_tests/comparison.xml</path>
            <InputDataFile>../Data/Input_data_File_Template.xml</InputDataFile>
            <runtype>sequential_keywords</runtype>
            <Execute ExecType="Yes"></Execute>
            <runmode type="rup" value="2"></runmode>
            <context>positive</context>
            <impact>noimpact</impact>
            <onError action="goto" value="1"></onError>
        </Testcase>
    </Testcases>
</TestSuite>
```

Figure 67
Run Suite Case Until Pass

Step 3
Execute the suite.

Step Result:
The cases in the suite run until the suite passes or the suite is executed as indicated in the Max_Attempts attribute.

✓ **This task is complete.**

15.2 Designing Conditional Suite

In this section:

15.2.1 Execute Type: Yes

When the execute type for a suite is set to *yes*, the suite is executed.

This procedure describes how to set the Execute Type of a suite to *Yes*.

Step 1
Create a project that contains the suites to be executed.

Step 2
After selecting the suite path, set its Execute Type to *Yes*.

```
<?xml version="1.0" encoding="utf-8"?>
<Project>
    <Details>
        <Name>Demo_Project</Name>
        <Title>Demo_Project</Title>
        <State>In Review</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/16/2020</Date>
        <default_onError action="abort" value=""></default_onError>
        <Resultsdir>/home/results</Resultsdir>
        <Time>21:40:17</Time>
    </Details>
    <Testsuites>
        <Testsuite TS="1">
            <path>../Suites/demo_suite.xml</path>
            <Execute ExecType="Yes">
            </Execute>
            <impact>impact</impact>
            <onError action="abort"></onError>
        </Testsuite>
    </Testsuites>
</Project>
```

Figure 68
Execute Type: Yes

✓ **This task is complete.**

15.2.2 Execute Type: No

When the execute type for a suite is set to *no*, the suite would not be executed under any circumstances. This option exists because when running a project, a user may not want to run some suites every time the user runs that project.

This procedure describes how to set the Execute Type of a suite to *No*.

Step 1
Create a project that contains the suites to be executed.

Step 2
After selecting the suite path, set its Execute Type to *No*.

```
<?xml version="1.0" encoding="utf-8"?>
<Project>
    <Details>
        <Name>Demo_Project</Name>
        <Title>Demo_Project</Title>
        <State>In Review</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/16/2020</Date>
        <default_onError action="abort" value=""></default_onError>
        <Resultsdir>/home/results</Resultsdir>
        <Time>21:40:17</Time>
    </Details>
    <Testsuites>
        <Testsuite TS="1">
            <path>../Suites/demo_suite.xml</path>
            <Execute ExecType="No">
            </Execute>
            <impact>impact</impact>
            <onError action="abort"></onError>
        </Testsuite>
    </Testsuites>
</Project>
```

Figure 69
Execute Type: No

✓ **This task is complete.**

15.2.3 Execute Type: If

This condition is the programming equivalent of an *If* statement in Warrior.

If a certain condition is met:

- Execute this Suite
- Otherwise, skip this Suite

Either:

- Proceed to the next Suite
- Proceed to a specified Suite
- or, abort execution of the Project

After the execute type has been set to *If*, the user 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 test *suite_<suite-number>_status* and the value for this condition, depending upon when the user wants to execute this Suite, can be either:

- PASS
- FAIL
- ERROR
- or SKIPPED

An *else* for this *if* is available in case the condition is not met. The value set in this else lets Warrior know what to do in case the condition is not met. A user has the option of continuing the execution by executing the next Suite, or executing a particular Suite, or aborting the execution of the project.

This procedure describes how to set the Execute Type of a suite to *If*.

Step 1
Create a project that contains the suites to be executed.

Step 2
After selecting the suite path, set its Execute Type to *If*.

Step 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 set Else

```
<?xml version="1.0" encoding="utf-8"?>
<Project>
    <Details>
        <Name>Demo_Project</Name>
        <Title>Demo_Project</Title>
        <State>In Review</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/16/2020</Date>
        <default_onError action="abort" value=""></default_onError>
        <Resultsdir>/home/results</Resultsdir>
        <Time>21:40:17</Time>
    </Details>
    <Testsuites>
        <Testsuite TS="1">
            <path>../Suites/demo_suite.xml</path>
            <Execute ExecType="If">
                <Rule Condition="testsuite_1_result" Condvalue="Pass" Else="next" Elsevalue=""></Rule>
            </Execute>
            <impact>impact</impact>
            <onError action="abort"></onError>
        </Testsuite>
    </Testsuites>
</Project>
```

Figure 70
Execute Type: If

✓ **This task is complete.**

15.2.4
Execute Type: If Not

This condition is the programming equivalent of an *! if* statement in Warrior.

If a certain condition is not met:

- Execute this Suite
- Otherwise, skip this Suite

Either:

- Proceed to the next Suite
- Or, proceed to a specified Suite
- Or, abort execution of the Project

After the execute type has been set to *If Not*, the user 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, depending upon when the user wants to execute this Suite, can be:

- PASS
- FAIL
- ERROR
- or SKIPPED

An *else* for this *if* is available in case the condition is met. The value set in this else lets Warrior know what to do in case the condition is met. A user has the option of continuing the execution by executing the next Suite, or executing a particular Suite, or aborting the execution of the Project.

This procedure describes how to set the Execute Type of a Suite to *If Not*.

Step 1
Create a project that contains the suites to be executed.

Step 2
After selecting the suite path, set its Execute Type to *If Not*.

Step 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 set Else

```
<?xml version="1.0" encoding="utf-8"?>
<Project>
    <Details>
        <Name>Demo_Project</Name>
        <Title>Demo_Project</Title>
        <State>In Review</State>
        <Engineer>Warrior_user</Engineer>
        <Date>07/16/2020</Date>
        <default_onError action="abort" value=""></default_onError>
        <Resultsdir>/home/results</Resultsdir>
        <Time>21:40:17</Time>
    </Details>
    <Testsuites>
        <Testsuite TS="1">
            <path>../Suites/demo_suite.xml</path>
            <Execute ExecType="If Not">
                <Rule Condition="testsuite_1_result" Condvalue="Pass" Else="next" Elsevalue=""></Rule>
            </Execute>
            <impact>impact</impact>
            <onError action="abort"></onError>
        </Testsuite>
    </Testsuites>
</Project>
```

Figure 71
Execute Type: If Not

✓ **This task is complete.**

16

Designing Suite for Common Setup, Cleanup, and Debug

In this chapter:

16.1 <TestSuite> tag

Test wrapper file can be included in Test Suite as mentioned 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)

   Note: When executing these testcases only test steps are executed which are in <Steps> block (test case level test wrapper file setup and cleanup steps need not be executed). All these testcases are executed as specified in exectype of test suite (parallel/sequential/RMT/RUP/RUF)
3. Cleanup steps (from wrapper file <Cleanup> block)
4. Debug steps (from wrapper file <Debug> block)

16.2
Execution Order

Refer to the following 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

✓	Setup Steps from Suite TestWrapper File—Executed
X	Setup Steps from Test Case1 TestWrapper File—Not Executed
✓	Test Steps from Test Case1—Executed
X	Cleanup Steps from Test Case1 TestWrapper File—Not Executed
X	Setup Steps from Test Case2 TestWrapper File—Not Executed
✓	Test Steps from Test Case2—Executed
X	Cleanup Steps from Test Case2 TestWrapperFile—Not Executed
✓	Cleanup Steps from Suite TestWrapper File—Executed

- 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 Test Case1—Executed
✓	Cleanup Steps from Test Case1 TestWrapper File—Not Present
✓	Setup Steps from Test Case2 TestWrapper File—Executed
✓	Test Steps from Test Case2—Executed
✓	Cleanup Steps from Test Case2 TestWrapper File—Not Present
X	Cleanup Steps from Suite TestWrapper File- Not Present

16.3
Exec Type RMT, RUP, RUF Behavior

1. Setup Steps and Cleanup Steps from Suite TestWrapper file in the preceding given blocks need 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, for example, Setup Steps and Cleanup Steps from Test TestWrapper file, Test steps are considered for RMT/RUP/RUF.

3. TestWrapperFile is optional, if not provided testcase or testsuite is executed as per existing design.

4. If one of the setup steps fails, 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.

16.4
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	

17
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 proceeds 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 proceeds to next step.

2. **Step Error Handling:** When an error is encountered during execution of a keyword, the keyword fails. 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' />

   - The step error handling supersedes the case default error handling.
   - If step error handling is not defined, the default error handling is used. If the default error handling is not set as well, Warrior proceeds 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 proceeds to next case.

   - 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' />
   - The case error handling supersedes the suite default error handling.
   - If case error handling is not defined, the default error handling is used. If the default error handling for the suite is not set as well, Warrior proceeds 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 proceeds to next suite.

   - 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'/>
   - The suite error handling supersedes the project default error handling.
   - If suite error handling is not defined, the default error handling for the project is used. If the default error handling is not set as well, Warrior proceeds to next suite in case of a suite failure.

18
Creating Keywords

Keywords in Warrior are created in Action files that are imported by the product drivers. If a user is creating a new keyword file, perform the following actions:

- Create a directory under the Actions directory. Use the following naming convention: NameofyourchoiceActions
- Inside the newly created directory, create the 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:

- The keywords can be methods of a class or they can be an independent function in a library
- When using classes for keywords:
  - Users should always use the default __init__ for a class as shown beneath, 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 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 substeps that are implemented in the code.

Sample Keyword Class

```
'''
Copyright 2017, Fujitsu Network Communications, Inc.
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
'''
#============================================================
# Import any python library you want to use below this line"""
#============================================================
import time
import pexpect

#=========================================================================
# """ Import Warrior Framework Utilities below this line"""
#=========================================================================
# import warrior.Framework
from warrior.Framework import Utils

#++++++++++++++++++++++++++++++++++++++
#WARRIOR KEYWORD TEMPLATE AND RULES
#++++++++++++++++++++++++++++++++++++++

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

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

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

class CliActions(object):

""" Default __init__ field must be used when using classes for keywords """
def __init__(self):
""" Constructor
"""
self.resultfile = Utils.config Utils.resultfile
```

19

Executing Warrior Framework Cases, Suites, and Projects

In this chapter:

19.1 Executing Warrior through CLI

In this section:

To run Warrior through a Command Line Interface, user needs to be inside this directory:

```
Warriorframework_py3/warrior
```

If a user wants to execute Warrior from any directory, substitute the Warrior in the following commands with a path to the Warrior executable.

Example: If the user is currently inside the home directory, that is, inside */home/user*, and the warriorframework_py3 directory is inside this directory, user can simply run Warrior from */home/user* and the following commands would become:

```
/home/user/warriorframework_py3/warrior/Warrior/path/case.xml
```

Warrior Cases can be executed in multiple ways. The following sections are a list of CLI execution options.

19.1.1 Run Case

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

19.1.2 Run Multiple Cases

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

19.1.3 Run Case Multiple Times

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

19.1.4
Run Case Until Failure

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

19.1.5
Run Case Until it Passes

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

19.1.6
Run Suite

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

19.1.7
Run Multiple Suites

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

19.1.8
Run Project

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

19.1.9
Run Multiple Projects

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

19.1.10
Run Combination of Case, Suite, and Project

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

19.1.11
Schedule Execution

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

19.1.12
Run Keywords in Case in Parallel and Cases in Sequence

By using the following command, a user can run all the cases in sequence (that is, one after the other) and all the keywords in each case would run in parallel (that is, all at once).

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

19.1.13
Run Keywords in Case in Sequence and Cases in Parallel

By using the following command, a user can run all the cases in parallel (that is, all at once) and all the keywords in each case would run in sequence (that is, one after the other).

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

19.1.14
Execution Based on Category

Warrior CLI supports execution of cases by category. The category field within the cases reads to determine if a case is a match. By providing a category and a test directory, Warrior searches the cases in the directory for cases that match the given 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 is 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 match at least one of the provided categories and execute them as individual cases.

Note: The cases are not 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 match at least one of the provided categories cat1 cat2 cat3, and execute them as individual cases.

Note: No suite is 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 location 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 match 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 match 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
```

19.1.15
JIRA Bug Reporting

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

Default JIRA Project Definition

1. Warrior framework uses 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*, 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
```

19.2
Executing Warrior through Katana

Warrior can be executed through Katana. Refer to *Katana User Guide*.

20
Understanding Logs and Results

In this chapter:

When a case, suite, or 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, in the path mentioned in the case, suite, or project—if that is left empty, in the default directory—the Execution folder in Warriorspace.

20.1
Project Execution Stucture

The results directory for a project would be the same name as that of the project. If the results directory is the first one in the destination for that particular project, the time stamp would not be appended to the directory name, like the first directory in the following figure.

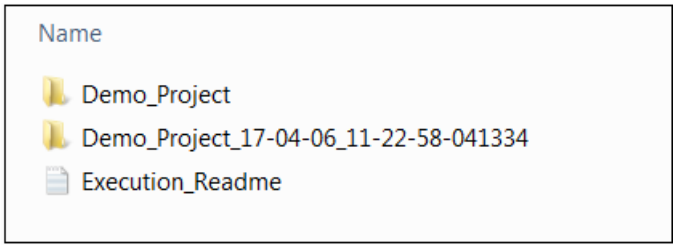


Figure 72
Results Directory

If the same project is running for the second time, a date time stamp would be appended to the directory name, like the second directory in the preceding figure.

This project-execution directory has directories for each of the suite contained in the project. These directories have the same name as the Suites that were run. Because the project has only one Suite with the name *Demo_Suite* and there would be only one subfolder with the name *Demo_Suite* in this directory.

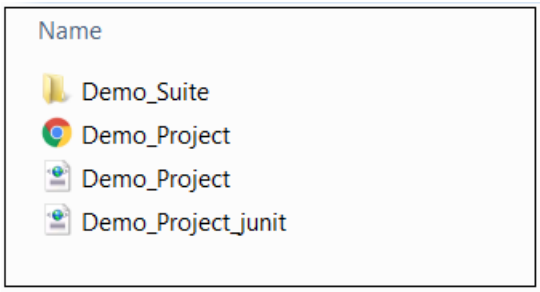


Figure 73
Demo_Suite Folder

- 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 (the Project that was executed) exist inside this directory.

The Suite folder has individual folders for all the cases contained in it. The Suite contains three cases, we can see three subfolders with the name of the cases created here.

Figure 74
Suite Folder

A Suite file (the Suite that was executed) exists inside this directory.

When we go into one of the directories for the Case, we can see three subfolders exist underneath it—Defects, Logs, and Results.

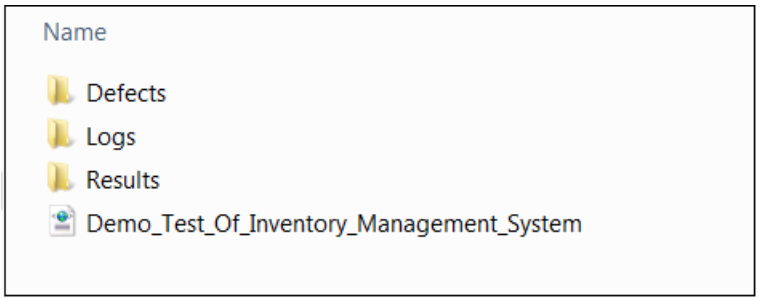


Figure 75
Case Directory

A Case file (the Case that was executed) exists inside this directory.

- If the case passed with no defects, error, or exceptions (which happened in this case), the defects folder is empty.
- The Logs folder would contain logs from all the system that were used by the case and the consoleLogs. The consoleLogs contain all the activity that went on the console when the case was running. These consoleLogs contain all the information a user needs to know what happened during the execution.

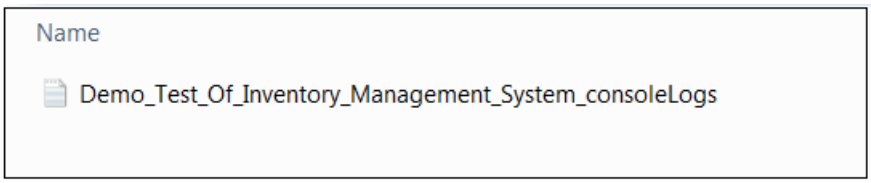


Figure 76
Console Logs

- The Results directory 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.

Figure 77
Keyword_Results

- 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 used by the HTML result file, although can open these files up and go through them if a user wants to.

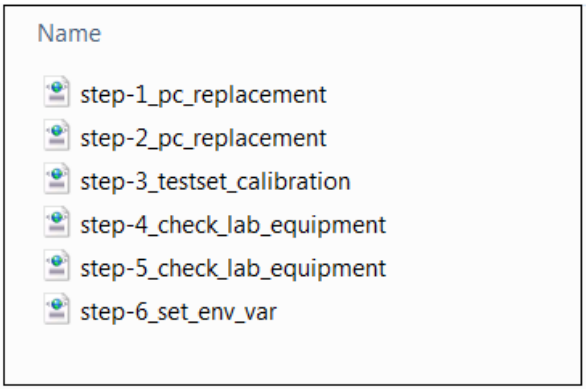


Figure 78
Results Files

20.2
Suite Execution Stucture

The results directory for a suite would be the same name as that of the suite. If the results directory is the first one in the destination for that particular suite, the time stamp would not be appended to the directory name, like the first directory in the following figure.

Figure 79
Results Directory

If the same suite is running for the second time, a date time stamp would be appended to the directory name, like the second directory in the preceding figure.

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

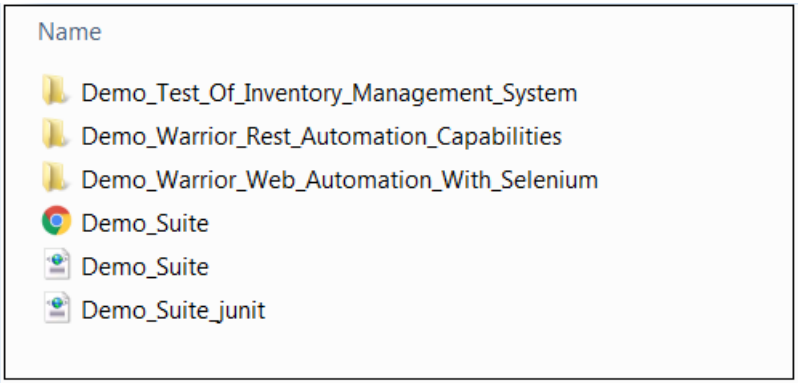


Figure 80
Suite-Execution Directory

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 (the Suite that was executed) exists inside this directory.

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

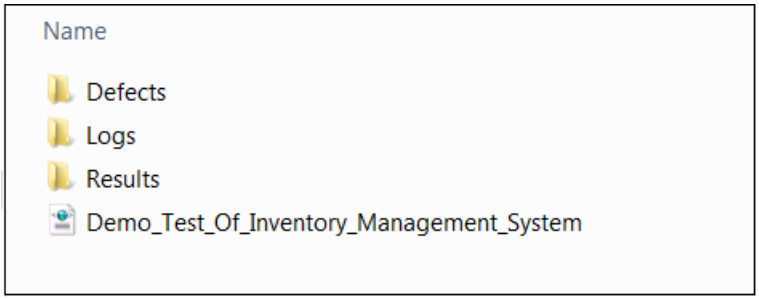


Figure 81
Case Directory

A Case file (the Case that was executed) exists inside this directory.

- If the case passed with no defects, error, or exceptions (which happened in this case), the defects folder is empty.
- The Logs folder would contain logs from all the system that were used by the case and the consoleLogs. The consoleLogs contain all the activity that went on the console when the case was running. These consoleLogs contain all the information a user needs to know what happened during the execution.

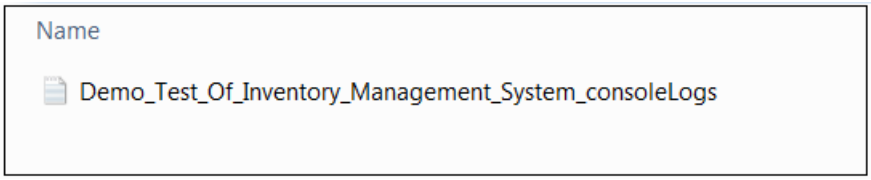


Figure 82
Console Logs

- The Results directory 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.

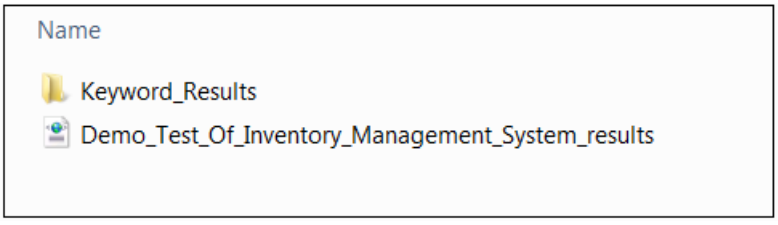


Figure 83
Keyword_Results

- 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 used by the HTML result file, although can open these files up and go through them if the user wants to.

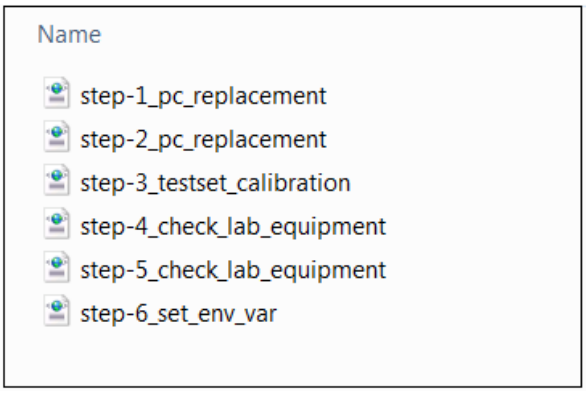


Figure 84
HTML Results Files

20.3
Case Execution Stucture

The Results directory for a Case would be the same name as that of the Case. If the results directory is the first one in the destination for that particular suite, the time stamp would not be appended to the directory name—like the first, third, and fifth directories in the following figure.

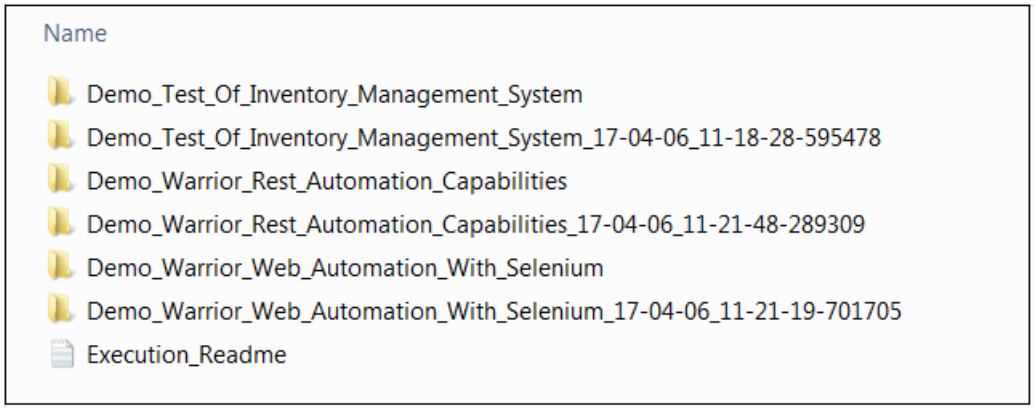


Figure 85
Results Directory

If the same case is running for the second time, a date time stamp would be appended to the directories name, like the second, fourth, and sixth directories in the preceding figure.

When we go into one of the directories for the Case, we can see three subfolders exist underneath in—Defects, Logs, and Results.

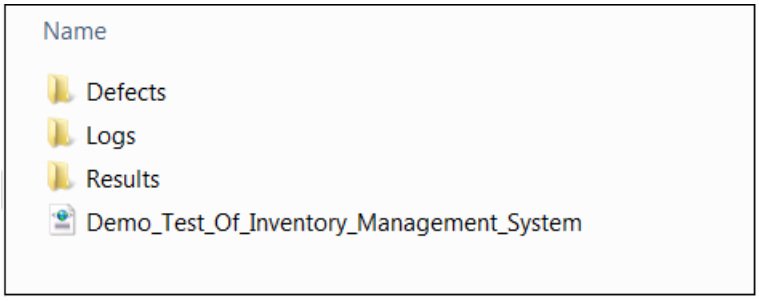


Figure 86
Case Directory

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 (the Case that was executed) exist inside this directory.

- If the case passed with no defects, error, or exceptions (which happened in this case), the defects folder is empty.
- The Logs folder would contain logs from all the system that were used by the case and 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 the user needs to know what happened during the execution.

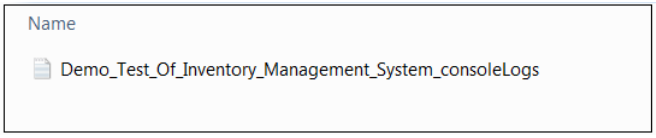


Figure 87
Console Logs

- The Results directory 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.

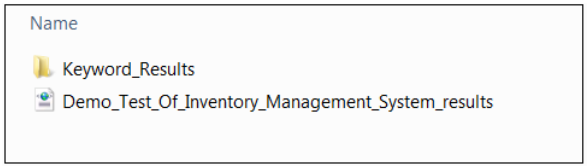


Figure 88
Keywords_Results

- 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 used by the HTML result file, although can open these files up and go through them, if a user wants to.

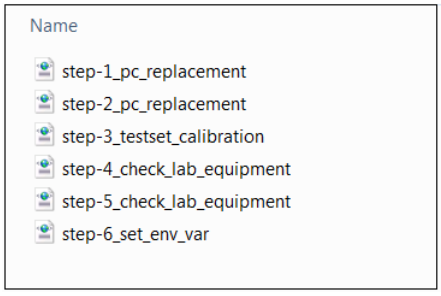


Figure 89
HTML Results Files

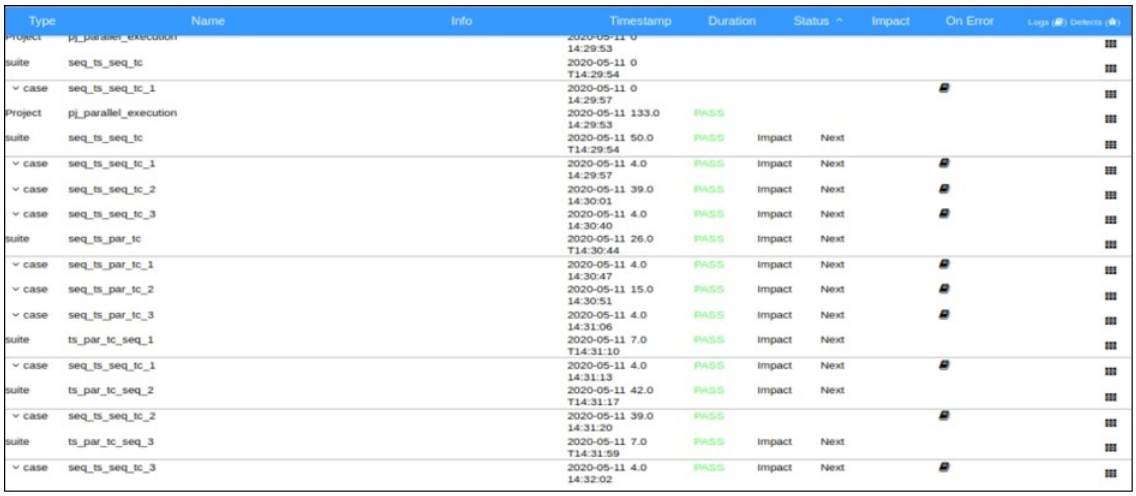
20.4
HTML File

The HTML files available in each of the execution folders described previously 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. The following subsections provide a brief explanation for all three levels.

Project

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

By seeing the result file, a user can determine which keywords, cases, and suites passed and which did not.



Type	Name	Info	Timestamp	Duration	Status	Impact	On Error	Logs Defects
Project	pj_parallel_execution		2020-05-11 14:29:53	0				
suite	seq_ts_seq_tc		2020-05-11 T14:29:54	0				
case	seq_ts_seq_tc_1		2020-05-11 14:29:57	0				
Project	pj_parallel_execution		2020-05-11 14:29:53	133.0	PASS			
suite	seq_ts_seq_tc		2020-05-11 T14:29:54	50.0	PASS	Impact	Next	
case	seq_ts_seq_tc_1		2020-05-11 14:29:57	4.0	PASS	Impact	Next	
case	seq_ts_seq_tc_2		2020-05-11 14:30:01	39.0	PASS	Impact	Next	
case	seq_ts_seq_tc_3		2020-05-11 14:30:40	4.0	PASS	Impact	Next	
suite	seq_ts_par_tc		2020-05-11 T14:30:44	26.0	PASS	Impact	Next	
case	seq_ts_par_tc_1		2020-05-11 14:30:47	4.0	PASS	Impact	Next	
case	seq_ts_par_tc_2		2020-05-11 14:30:51	15.0	PASS	Impact	Next	
case	seq_ts_par_tc_3		2020-05-11 14:31:06	4.0	PASS	Impact	Next	
suite	ts_par_tc_seq_1		2020-05-11 T14:31:10	7.0	PASS	Impact	Next	
case	seq_ts_seq_tc_1		2020-05-11 14:31:13	4.0	PASS	Impact	Next	
suite	ts_par_tc_seq_2		2020-05-11 T14:31:17	42.0	PASS	Impact	Next	
case	seq_ts_seq_tc_2		2020-05-11 14:31:20	39.0	PASS			
suite	ts_par_tc_seq_3		2020-05-11 T14:31:59	7.0	PASS	Impact	Next	
case	seq_ts_seq_tc_3		2020-05-11 14:32:02	4.0	PASS	Impact	Next	

Figure 90
Project HTML Files

Suite

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

By seeing the result file, a user 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.

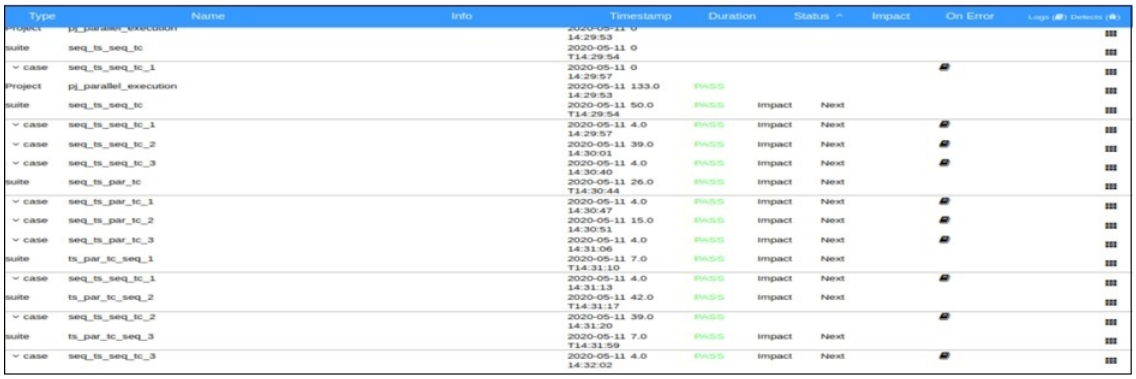


Figure 91
Suite HTML Files

Case

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

By seeing the result file, a user 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.

Figure 92
Case HTML Files

20.5
JUnit File

This file is generated for the benefit of external tools, for example, Jenkins which can read the JUnit files generated by case executions to display the results graphically.

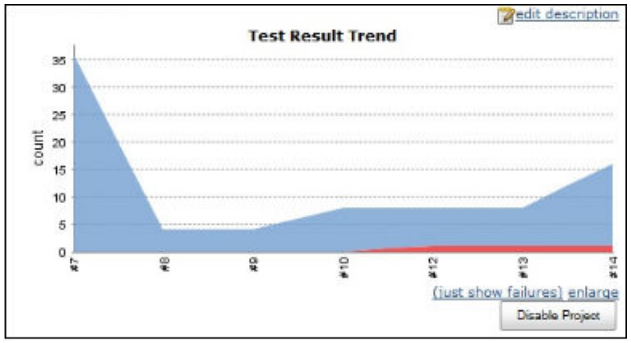


Figure 93
Test Result Trend

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.

21
Guidelines to Create 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 guidelines help a user to create a brand new Warrior keyword repository:

- The repository name must be the name of the product for which a user is developing the Warrior keywords. This would be the root of the repository.
- Inside the root of the repository, a directory structure as shown in the following figure must be maintained.

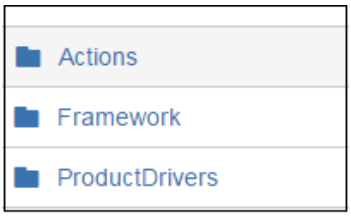


Figure 94
Root Repository

- 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.

22
Guidelines to Create Warriorspace Repository

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

- The repository name should be the name of the product for which a user is developing the Warriorspace. This would be the root of the repository.
- Inside the root of the repository, a directory structure as shown in the following figure should be maintained.

Figure 95
Warriorspace Directory

- Inside this Warriorspace directory, this file structure is recommended.

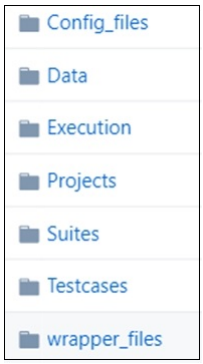


Figure 96
Warriorspace Directory Structure