Slayer Documentation

Release 0.1.0a9

FrancoLM

Feb 06, 2019

Contents:

CHAPTER 1

Slayer is the QA Automation Framework that came to SLAY!

Slayer is a QA Automation Framework developed in Python, based on BDD.

CHAPTER 2

Why use Slayer?

- Easy-to-read logging: Each scenario and each step is logged, and also you can add logging to each step
- Framework architecture versatility: Slayer supports the creation of multiple features and steps directory levels
- Slayer was created using open-source tools, so you can modify it's behavior in any way you need

BDD, or Behavior Driven Development, is a software development process emerged from Test Driven Development (TDD).

Besides following the practices of TDD, BDD provides an easy-to-understand vocabulary for developers and stakeholders. Having a common vocabulary helps both technical and non-technical members in an organization to better communicate and to create software with higher quality.

This framework uses a lot of the functionality provided by the Python Behave project.

2.1 Installing Slayer

2.1.1 Requirements

- Python 3.x (Python >= 3.6 recommended)
- For web automation, make sure you have a webdriver downloaded in your system, and it's added to your PATH (Windows).

Projects like Selenium automate web browsers, and the Python library provided by this project is used in Slayer. Refer to their documentation for more information.

2.1.2 Installing the Framework

Slayer can be installed either as a library or cloned from the source code.

If you want to use the latest stable version, the library would be your best option.

```
pip install slayer
```

If you want get the latest version, then install Slayer from the github repository.

```
pip install git+https://github.com/FrancoLM/slayer
```

2.2 Slayer Framework Tutorial

Slayer makes use of the Behave Python library to run test cases. So, very much like Behave, Slayer will look for a folder called "features" and a "steps" sub-folder inside it for feature files and the steps implementation, respectively. You can consult the Behave documentation for more information.

2.3 Tutorial

Let's go trough a simple example. Let's create a test that opens the Wikipedia webpage and searches for the term "Behavior Driven Development"

- Import Slayer in your project
- Install the Chrome webdriver
- Create a WikipediaPage class, that we'll use to automate our test. In the root of your project create a Python script "wikipedia_page.py":

```
import time
from selenium import webdriver
from selenium.webdriver.common.by import By
from slayer.lib.common.web.page_object import PageObject


class WikipediaPage(PageObject):
    url = "https://www.wikipedia.org"
    # Locators
    page_title = (By.CLASS_NAME, "svg-Wikipedia_wordmark")
    search_bar = (By.ID, "searchInput")
    search_button = (By.XPATH, '//*[@id="search-form"]/fieldset/button')
    search_result_title = (By.XPATH, '//*[@id="firstHeading"]')

    def __init__(self, webdriver):
        super().__init__(webdriver)

    def validate_page(self):
        self.find_element(*self.page_title)

    def search_for(self, query):
        self.find_element(*self.search_bar).send_keys(query)
        self.find_element(*self.search_button).click()
        time.sleep(1)

    def get_search_result_title(self):
        return self.find_element(*self.search_result_title).text
```

- Add a new directory called "features" in your root, and create a file "wikipedia.feature", and paste the following:

```
Feature: Open the Wikipedia webpage and perform a search

  @WIKIPEDIA
  Scenario Outline: My first Slayer test
    Given I open a browser
    And I navigate to the Wikipedia page
    When I search for the text '<search_query>'
    Then I see in the page '<search_result>'
  Examples:
    | search_query                | search_result                  |
    | Python language             | Python (programming language)  |
    | Behavior Driven Development | Behavior-driven development    |
```

- Add a new directory "steps" inside "features", and create a Python script "tutorial_steps.py":

```
import logging
import time
from behave import step
from selenium import webdriver
from tutorial.wikipedia_page import WikipediaPage


@step("I open a browser")
def step_impl(context, maximized=True):
    context.driver = webdriver.Chrome()
    if maximized:
        context.driver.maximize_window()


@step("I navigate to the Wikipedia page")
def step_impl(context):
    context.wikipedia_page = WikipediaPage(context.driver)
    logging.info("Navigating to the Wikipedia page")
    context.wikipedia_page.navigate()


@step("I see in the page '{search_text}'")
def step_impl(context, search_text):
    logging.info("Searching for the text '{}'".format(search_text))
    context.wikipedia_page.search_for(search_text)
    time.sleep(1)
```

- In your main script, import Slayer and run it:

And that's it! Slayer runs your test! You will find the output for the execution inside the "output" folder that Slayer creates automatically.

2.4 Modifying the Slayer execution

Slayer uses configuration files to setup the execution options. These options include output folders, logging format, and proxy compatibility. The default Slayer configuration files are shown below.

2.4.1 Slayer configuration: config.cfg

```
[slayer]
[[output]]
path = output

[[logs]]
# Logs path will be inside the output folder
path = logs

[[artifacts]]
# artifacts path will be inside the output folder
path = artifacts

[[proxy]]
http_proxy =
https_proxy =
no_proxy =
```

2.4.2 Slayer logging: logger.yaml

```
version: 1
disable_existing_loggers: True
formatters:
  # Add a new formatter if it's needed by your application
  basic:
    format: '%(asctime)s %(levelname)-8s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  in_console:
    format: '      %(asctime)s: %(levelname)-8s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
handlers:
  console:
    class: logging.StreamHandler
    level: INFO
    formatter: in_console
    stream: ext://sys.stdout
  error_console:
    class: logging.StreamHandler
    level: WARN
    formatter: in_console
    stream: ext://sys.stderr
  file:
    class: logging.FileHandler
    level: INFO
    formatter: basic
    filename: output.log
    mode: a
    encoding: utf-8
root:
  level: INFO
  propagate: False
  handlers: [console, error_console, file]
```

2.4.3 Provide your own configuration files to Slayer

The config.cfg and logger.yaml files describe the default behavior for Slayer. But you can change Slayer works by providing the path and filename for your own configuration and logger files, by providing any of these two arguments in your execution.

```
<execution command> --framework-config <custom_config_file> --logs-config <custom_
↪logger_file>
```

For example, if your main script is called main.py, and your configuration files are called "my_config.cfg" and "my_logger.yaml":

```
python main.py --framework-config my_config.cfg --logs-config my_logger.yaml
```

Slayer will run, but instead of using the default configuration, it will use your own configuration files.

Note: The logger file specifies the options for the Python default logger. The options defined are read as a dict and passed as option in a logging.config.dictConfig call. Please refer to the logging configuration for more details and available options

---

2.4.4 Behave execution

Slayer configures the behave execution before running the tests. But this configuration can be overridden by providing the framework with a behave.ini file. The default file define the following options:

In effect, all options defined in this file are default behave options.

If you want to define your own options, create a file called "behave.ini", and provide Slayer with the path to this file: Following the previous example:

```
python main.py --behave-config <path_to_behave_ini_file>
```

By default, Slayer will look for the "behave.ini" file in the same folder your main script is located, so you do not need to use this argument if your ini file is located there.

Note: Refer to the Behave Parameters documentation for information on available parameters.

2.5 Why Slayer was developed

In progress

2.6 Slayer Features

In progress. The benefits of using Slayer - Easy-to-read logging: Each scenario and each step is logged, and also you can add logging to each step - Framework architecture versatility: Slayer supports the creation of multiple features and steps directory levels - Slayer was created using open-source tools, so you can modify it's behavior in any way you need Mention these but in more details

CHAPTER 3

Indices and tables

- genindex
- modindex
- search