**BDD (Breaking Down Behavioral Driven Development) , Cucumber and Selenium WebDriver based Test Automation Framework**

**Example:**

The different pages we wanted to test were:

* Dashboard: Home page of the application — provides information about the status, settings, battery, GPS etc.
* Calibration: Contains information about current calibration of the instrument and a couple of wizards to calibrate the device.
* Diagnostics: Contains diagnostics information about the instrument, allows to download current logs from the instrument.
* System: Network Settings, Wifi Settings, Instrument Control (reboot instrument, reset it to factory defaults etc).
* About: Has some legal terms, document links and a section for Firmware Update.

### Test Plan

I usually start with talking to different users of the product before starting to write a test plan — what are the different use-cases and/or scenarios that you would use the application for. This usually gives me a good foundation for a BDD based Test plan.

For example, for the Dashboard page shown in the screenshot above, we can have the following BDD scenarios which can be reliably automated.

**Feature:** Website Dashboard  
  
 **Scenario Outline:** Components  
 **Given** I load the website  
 **When** I go to "Dashboard" page  
 **Then** I see this component "<boxes>"  
 **Examples:  
 |** boxes **|  
 | Status |  
 | Settings |  
 | Battery |  
 | GPS |  
  
 Scenario Outline:** Status  
 **Given** I load the website  
 **When** I go to "Dashboard" page  
 **Then** Dashboard Status shows correct values for row "<rows>"  
 **Examples:  
 |** rows **|  
 | Status |  
 | Reference Count |  
 | Last Reference |  
 | Last Optimize |  
 | Last Wavelength Check |  
  
 Scenario:** Status Refresh  
 **Given** I load the website  
 **When** I go to "Dashboard" page  
 **Then** Clicking on Status Refresh should refresh status component  
  
 **Scenario:** Battery  
 **Given** I load the website  
 **When** I go to "Dashboard" page  
 **Then** Dashboard Battery shows Battery or AC Power with correct icon.  
  
 **Scenario:** Battery Refresh  
 **Given** I load the website  
 **When** I go to "Dashboard" page  
 **Then** Clicking on Battery Refresh should refresh battery component  
 **Scenario Outline:** GPS  
 **Given** I load the website  
 **When** I go to "Dashboard" page  
 **Then** Dashboard GPS shows correct values for row "<rows>"  
 **Examples:  
 |** rows **|  
 | Fix |  
 | Location |  
 | Altitude |  
 | Format |**

I came up with similar tests for different pages. Some of the configuration and workflow pages were slightly more complicated, but for the purposes of this write-up we can safely ignore them.

The next step is to write definitions for the above given , when , thenconditions. And this step is made easier with automatic code scaffolding provided by Python’s Behave package.

### Step Definitions

Make sure you have Python Behave package installed. Another package we will need soon is selenium webdriver bindings for Python. These can be installed using pip:

pip install behave  
pip install selenium

Store the above Feature file as dashboard.feature. I would ideally store all feature files in a folder called features but that is just my personal preference.

Run behave to generate the code scaffolding for this feature file. It would look something like below, copy over the code portion into a Python file for implementation. Again, the name and location of the Python implementation file does not matter — but to follow best practices, I would put it under steps/folder.

$ behave dashboard.feature

Feature: Dashboard # dashboard.feature:1  
...

You can implement step definitions for undefined steps with these snippets:

@given(u'I load the website')  
def step\_impl(context):  
 raise NotImplementedError(u'STEP: Given I load the website')

@when(u'I go to "Dashboard" page')  
def step\_impl(context):  
 raise NotImplementedError(u'STEP: When I go to "Dashboard" page')

@then(u'I see this component "Status"')  
def step\_impl(context):  
 raise NotImplementedError(u'STEP: Then I see this component "Status"')

...

### Passing Arguments to Step Definitions

I wanted to highlight one thing from the previous section that may go unnoticed. For couple of scenarios above, we take different examples that tell behave what argument the step definitions associated with that outline need to be called with. In Scenario Outline — we specify this through Examples, and in actual Step definition, we can retrieve these as arguments to the step definitions function. Below snippet clarifies how this is done:

# features/scenario.feature  
**Scenario Outline**:  
 **Given** An example scenario with "<rows>"  
 **When** An example "condition" occurs  
 **Then** An example "assertion" occurs  
**Examples**:  
 | rows |  
 | Item 1 |  
 | Item 2 |  
 | Item 3 |

# steps/scenario.py  
**@given**('An example scenario with "{row}"')  
**def step\_impl**(context, row):  
 # Will be called with Item 1, Item 2, Item 3  
 print('Scenario implementation for row: {}'.format(row))

...

In summary, most of what we need is available outside of behave’s context variable — and keeps our code simple and straightforward to read.

### Steps Consolidation

If you have used Scenario Outlines or Examples in your BDD steps, you are more than likely to have the code scaffolder generate repetitive steps with different example names tied in. I have no idea why the Behave Code Scaffolder generates it this way, so I created an issue for them to look into. But for now, let’s consolidate different step definitions. For the dashboard file, I came up with following list.

Another thing I would do at this point is to separate out common step implementations into a separate file — in above example, loading website, going to a page etc., would be common steps that can be included in a steps/common.py file.

My steps/common.py file looks like below.

from behave import given, when, then

@given(u'I load the website')  
def step\_impl\_load\_website(context):  
 print('I load the website')

@when(u'I go to "{page}" page')  
def step\_impl\_goto\_page(context, page):  
 print('I go to {} page'.format(page))

@then(u'I see this component "{component}"')  
 def step\_impl\_verify\_component(context, component):  
 print('I see this component "{}"'.format(component))

And my steps/dashboard.py file looks like this.

from behave import given, when, then

@then(u'Dashboard Status shows correct values for row "{row}"')  
def step\_impl\_dashboard\_status(context, row):  
 print('Dashboard Status for row {}'.format(row))

@then(u'Clicking on Status Refresh should refresh status component')  
def step\_impl\_status\_refresh(context):  
 print('Status Refresh button')

@then(u'Dashboard Battery shows Battery or AC Power with correct icon.')  
def step\_impl\_battery\_status(context):  
 print('Dashboard battery status')

@then(u'Clicking on Battery Refresh should refresh battery component')  
def step\_impl\_battery\_refresh(context):  
 print('Battery refresh')

@then(u'Dashboard Settings shows correct values for row "{row}"')  
def step\_impl\_settings(context, row):  
 print('Dashboard settings for {}'.format(row))

@then(u'Dashboard GPS shows correct values for row "{row}"')  
def step\_impl\_gps\_settings(context, row):  
 print('Dashboard GPS row: {}'.format(row))

This is actually a runnable test implementation with BDD. If you run from terminal: $ behave dashboard.feature now, you will see that all the test cases are passing!

$ behave dashboard.feature

...

1 feature passed, 0 failed, 0 skipped  
22 scenarios passed, 0 failed, 0 skipped  
66 steps passed, 0 failed, 0 skipped, 0 undefined

Took 0m0.009s

### Page Object Planning

The next step is to implement the step definitions in a meaningful way. For this we will separate out the framework’s step definitions from the actual implementations. We will create Python classes that can implement our common functionality and functionality that we desire from each page. I like to approach this problem in a top-down fashion. Some people go about implementing this in a bottom-up fashion (develop the page objects first, and the use them in an application), but I feel that way we would probably be developing unnecessary code. Top-down approach (defining what we need based on step requirements, seems to be the shortest path to complete implementation).

So, let’s go ahead and define what we need from these page objects. I will just share the code here from step definitions to see what we need.

The steps/common.py looks like below. I am assuming we will implement a base class called WebApp which will actually implement this functionality.

from behave import given, when, then  
from framework.webapp import webapp

@given(u'I load the website')  
def step\_impl\_load\_website(context):  
 webapp.load\_website()

@when(u'I go to "{page}" page')  
def step\_impl\_goto\_page(context, page):  
 webapp.goto\_page(page)

@then(u'I see this component "{component}"')  
def step\_impl\_verify\_component(context, component):  
 webapp.verify\_component\_exists(component)

And the steps/dashboard.py now looks like:

from behave import given, when, then  
from pages.dashboard import dashboard

@then(u'Dashboard Status shows correct values for row "{row}"')  
def step\_impl\_dashboard\_status(context, row):  
 dashboard.verify\_status(row)

@then(u'Clicking on Status Refresh should refresh status component')  
def step\_impl\_status\_refresh(context):  
 dashboard.verify\_refresh()

@then(u'Dashboard Battery shows Battery or AC Power with correct icon.')  
def step\_impl\_battery\_status(context):  
 dashboard.verify\_battery\_status()

@then(u'Clicking on Battery Refresh should refresh battery component')  
def step\_impl\_battery\_refresh(context):  
 dashboard.verify\_battery\_refresh()

@then(u'Dashboard Settings shows correct values for row "{row}"')  
def step\_impl\_settings(context, row):  
 dashboard.veify\_setting(row)

@then(u'Dashboard GPS shows correct values for row "{row}"')  
def step\_impl\_gps\_settings(context, row):  
 dashboard.verify\_gps\_setting(row)

As you can see from above snippets, what we need from our page-objects, is not so much as a representation of the page, but rather some abstraction to perform and/or verify selected actions on the page. This is why I recommend following top-down approach to Page Object design — we will know exactly what we want from the pages that will allow us to develop it quicker.

Another pattern to notice from above is that, we are looking to contain the state of our test in terms of simple singletons. We don’t want to be creating new Python objects (and thereby creating multiple browser windows trying to do the same thing). So let’s begin our page design with singleton design pattern.

### Singleton Pattern

The next part would be to implement the Page Objects and Framework code that we specified in the previous sections. But first, let’s look at a simple singleton instance and how we can implement that.

First of all, why do we need a singleton pattern? In Python, all files are like scripts. Anything outside of functions and class definitions is executed as it is encountered. All code is interpreted top-down; that means, if Python interpreter sees any global statements, it will execute those. In our examples, if we did not have singleton patterns, we would be executing and creating multiple webapp and dashboard objects. Which could internally mean we are opening more than one browser window to test same things — which is not the behavior we want from our test framework.

A simple Singleton pattern might look like this:

# framework/webapp.py

class WebApp():  
 instance = None

@classmethod  
 def get\_instance(cls):  
 if cls.instance is None:  
 cls.instance = WebApp()  
 return cls.instance

def \_\_init\_\_(self):  
 pass  
   
 def load\_website(self):  
 # Code for loading website.

webapp = WebApp.get\_instance()

With above, what we are exporting is a class method that can be used to create an instance of the class. Clients of this class would call WebApp.get\_instance() to use this class, thereby making sure that only one instance of the class (the singleton object) is in use throughout the codebase. Python does not have private constructors, so we cannot force anyone else trying to instantiate WebApp directly, but that is not a problem we want to solve today.

Another thing to notice here: we are directly exporting webapp object as part of this file. This basically allows any client of this file to directly import webapp and start using it as follows.

# steps/common.py

from behave import given, when, then  
**from framework.webapp import webapp**

@given(u'I load the website')  
def step\_impl\_load\_website(context):  
 **webapp.load\_website()**

### Page Object Implementation

With the singleton design pattern and interfaces defined through step-definition files, our Page Object implementation looks as below. Note, I am not showing full implementation of these files as it requires access to a private website (and it would be too lengthy and unnecessary to the points in discussion in this article).

# webapp.py  
**from** selenium **import** webdriver  
**from** data.config **import** settings  
**from** urllib.parse **import** urljoin  
  
**class** WebApp():  
 instance = **None** @classmethod  
 **def** get\_instance(cls):  
 **if** cls.instance **is None**:  
 cls.instance = WebApp()  
 **return** cls.instance  
  
 **def** \_\_init\_\_(self):  
 self.driver = webdriver.Chrome()

**def** get\_driver(self):  
 **return** self.driver

**def** load\_website(self):  
 self.driver.get(settings[**'url'**])  
  
 **def** goto\_page(self, page):  
 self.driver.get(urljoin(settings[**'url'**], page))  
  
 **def** verify\_component\_exists(self, component):  
 # Simple implementation **assert** component **in** self.driver.find\_element\_by\_tag\_name(**'body'**).text, \  
 **"Component {} not found on page"**.format(component)  
  
  
webapp = WebApp.get\_instance()

And our implementation of dashboard page object might look like:

**from** framework.webapp **import** webapp  
  
**class** Dashboard():  
 instance = **None** @classmethod  
 **def** get\_instance(cls):  
 **if** cls.instance **is None**:  
 cls.instance = Dashboard()  
 **return** cls.instance  
  
 **def** \_\_init\_\_(self):  
 self.driver = webapp.get\_driver()  
  
 **def** verify\_status(self, row):  
 # Ex: # status = self.driver.find\_element\_by\_id('dashboard-status-component').text  # asssert row in status, "{} not present in status component".format(row) **pass  
  
 def** verify\_refresh(self):  
 # Ex: # refresh = self.driver.find\_element\_by\_id('dashboard-status-refresh-btn')  # refresh.click() **pass  
  
 def** verify\_battery\_status(self):  
 # Ex: # battery = self.driver.find\_element\_by\_id('dashboard-battery-status-component').text  # assert battery is not None **pass  
  
 def** verify\_battery\_refresh(self):  
 # Ex: # refresh = self.driver.find\_element\_by\_id('dashboard-battery-refresh-btn')  # refresh.click() **def** verify\_gps\_setting(self, row):  
 # Ex: # status = self.driver.find\_element\_by\_id('gps-setting-component').text  # asssert row in status, "{} not present in GPS component".format(row) **pass**dashboard = Dashboard.get\_instance()