**BDD Lab: Testing a Bookstore Web Application**

BDD, meaning “Behavior-driven development” is a software development style that encourages collaboration between the dev team, QA, and non-technical partners such as investors or business participants. BDD is presented in natural language styles such as the Gherkin language which presents behavior in a given-when-then format. This lab will guide you through using the Python based Behave Framework to utilize BDD methods in testing a book store web application.

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

1. **Getting Python, Behave and Required Packages:**

**Install Python**:

* + Ensure Python (version 3.7 or higher) is installed on your system. You can download it from [python.org](https://www.python.org/downloads/).
  + You can ensure Python is installed and check its version by opening a command line and running:

python --version

**Install Required Packages**:

* + Open a command line interface inside of the ‘WebApp’ folder
  + Run the requirements.txt file to install all required packages

pip install -r requirements.txt

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

1. **Set Up Your Testing Environment**:

**Setting Up the Python Virtual Environment**

* + Navigate to the ‘WebApp’ folder and open a command line. Initiate your Python virtual environment by running:

python -m venv env

**Run the Flask Application**:

* Navigate to your ‘WebApp’ folder in the command line and run:

flask --debug --app main run

* If you receive the error ‘flask is not a recognized command’, alternatively run:

**python -m** flask --debug --app main run

* The application should now be running at <http://127.0.0.1:5000/> and will respond to changes you make in the code when you refresh the page. The flask application will need to be running for the lab to interact with it.

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

1. Learning Resources:

**Official Behave Documentation**:

* + [Behave Documentation](https://behave.readthedocs.io/en/latest/): The official docs for Behave and how to use it.
  + [Official Behave GitHub Repo](https://github.com/behave/behave): The official Behave GitHub repository

**Community Forums and Support**:

* + [Stack Overflow](https://stackoverflow.com/questions/tagged/python-behave): Community Q&A on Behave-related topics.

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

1. **Writing Behave Tests**

**Understanding the file structure and Behave**:

* + In your *WebApp* folder, you will have a sub-folder called *Features*, this is where all Gherkin-based tests are stored for Behave in files noted as *.feature* files.
  + Inside the *Features* folder is another sub-folder called *Steps*, this is where the actual tests’ functionality goes in the form of Python scripts.
  + While multiple features can be put into a single file, they can also be split into multiple.
  + Similarly, steps can be combined into single files or split into multiple such as steps focusing on a specific action that will be reused often. (Ex: A step that opens the website to be tested).
  + For this lab, each scenario in the feature file will be broken down into its own step file.
  + Behave is a BDD **Python** Testing Framework. This means for web-interaction, a framework such as [Selenium Web-Driver](https://www.selenium.dev/documentation/webdriver/) or [Splinter](https://splinter.readthedocs.io/en/latest/) will need to be utilized. For this lab, Selenium Web-Driver will be used for all step-functionality.

**Basic Behave Test Structure**:

* + A base Behave feature written in Gherkin *given-when-then* looks like this:

Feature: showing off behave

    Scenario: Run a simple test

        Given we have behave installed

        When we implement a test

        Then behave will test it for us!

* + This feature’s corresponding steps would be written as:

from behave import \*

@given('we have behave installed')

def step\_impl(context):

    pass

@when('we implement a test')

def step\_impl(context):

    assert True is not False

@then('behave will test it for us!')

def step\_impl(context):

    assert context.failed is False

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

1. **Test Examples:**

**Test 1 – Validate that only partial input for a new book shows an error**:

This test enters only a title in the new book box and upon trying to submit, will be met with an error instead.

from behave import \*

from selenium import webdriver

from selenium.webdriver.common.by import By

driver = webdriver.Chrome()

@given('We have the bookstore and the new book box open')

def step\_impl(context):

    # Get the bookstore page

    driver.get("http://127.0.0.1:5000/")

    # Show the new book box

    new\_book\_button = driver.find\_element(By.ID, "newBookToggle")

    # Click the Add New Book button to display the book section if not displayed already

    if("none" in driver.find\_element(By.ID, "addNewBook").get\_attribute("style")):

        new\_book\_button.click()

    # Wait half a second for the box to appear

    driver.implicitly\_wait(0.5)

    # Ensure the box is open

    assert "block" in driver.find\_element(By.ID, "addNewBook").get\_attribute("style")

@when('We type a title in for a new book and nothing else')

def step\_impl(context):

    # Enter value into the title block

    driver.find\_element(By.ID, "newTitle").send\_keys("This is a test")

    # Nothing to assert, pass automatically

    pass

@then('It will not make a book and instead make an error')

def step\_impl(context):

    # Click the submit button

    driver.find\_element(By.ID, "submitNewBook").click()

    # Wait half a second for the box to appear

    driver.implicitly\_wait(0.5)

    # Assert the error box came up

    assert "block" in driver.find\_element(By.ID, "errorBox").get\_attribute("style")

**Test 2 – Sorting by the page column shows the smallest page count book**:

This test sorts the page column, leaving the results from smallest to largest then ensures the smallest page count is the one at the top of the list.

from behave import \*

from selenium import webdriver

from selenium.webdriver.common.by import By

driver = webdriver.Chrome()

@given('We have the bookstore open')

def step\_impl(context):

    # Get the bookstore page

    driver.get("http://127.0.0.1:5000/")

    # Wait for the page to load

    driver.implicitly\_wait(0.5)

    # Assert the page title to ensure its the correct page

    assert driver.title == "County Bookstore"

@when('We click the page count header')

def step\_impl(context):

    # Click the page header one time to sort from smallest -> largest

    driver.find\_element(By.ID, "pageHeader").click()

    # Nothing to assert, pass

    pass

@then('The book with the smallest page count will be at the top of the list')

def step\_impl(context):

    # Establish smallest page count

    smallest\_pages = None

    # Get the body of the table

    table\_body = driver.find\_element(By.ID, "tbody")

    rows = table\_body.find\_elements(By.TAG\_NAME, "tr")

    # Loop through all rows

    for row in rows:

        # Find the third column in each row and cast its contents as an int

        col = int(row.find\_elements(By.TAG\_NAME, "td")[3].text)

        # If the smallest page variable is set to None or greater than the col value, replace it

        if smallest\_pages == None or smallest\_pages > col:

            smallest\_pages = col

    # Get the page count of the first row

    first\_row\_page = int(rows[0].find\_elements(By.TAG\_NAME, "td")[3].text)

    # Make final assert

    assert smallest\_pages == first\_row\_page

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

1. **Incomplete Test Scenarios (Exercises YOU need to Complete):**

**Test 3 (Incomplete) – Ensure the search bar only filters by titles**:

* + - Write a test that uses the search bar to search for something other than a title of a book and get no results.
    - Instruction: Fill in the search input with a value other than something present in a title of a book and ensure there are no results.

**Test 4 (Incomplete) – Add a new book**:

* + - Write a test that adds a new book and find it in the list after submission.
    - Instruction: Fill in the new book contents, submit the form, and check to see if the new book is present.

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

1. **Expected Results from Testing**

* Test 1: After partial input of a book, it will display an error instead of submitting the new book.
* Test 2: The smallest page book will be the top row of the list.
* Test 3: After searching for a value not in a title, there will be no results in the books list.
* Test 4: The newly added book will be present in the list.

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

1. **Running the Tests**

Execute your tests using the command:

behave

It will automatically run all feature files in the *Features* sub-folder. Observe the results to ensure all tests are functioning as expected. Your command line should indicate:

1 feature passed, 0 failed, 0 skipped

4 scenarios passed, 0 failed, 0 skipped

XX steps passed, 0 failed, 0 skipped, 0 undefined

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

1. **Conclusion**

This lab should provide a good insight into the workings of the Behave BDD testing framework and how to implement it into a workflow. Behave allows business partners, QA testers, and development teams all get involved in the testing process to allow everyone to better understand how the program should function.