TDD / BDD Final Project

Estimated time: 60 minutes

Welcome to the **TDD / BDD Final Project** development environment. Now it's time to prove to yourself that you can apply the learning from the Test Driven Development and Behavior Driven Development modules of this course. This lab environment will provide you with a Cloud IDE and Docker which will enable you to carry out the following objectives:

Objectives

- Develop unit test cases for a Product model
- Develop unit test cases and code for a RESTful API for products
- Load background data from your BDD scenarios into your service before each scenario executes
- Develop a feature file to test the Product administrative UI with Behave

All of your work on the final project should be completed from this environment.

Important Security Information

Welcome to the Cloud IDE with Docker. This is where all of your development will take place. It has all of the tools you will need to use Docker.

It is important to understand that the lab environment is **ephemeral**. It only lives for a short while and then it will be destroyed. This makes it imperative that you push all changes made to your own GitHub repository so that it can be recrated in a new lab environment any time it is needed.

Also note that this environment is shared and therefore not secure. You should not store any personal information, usernames, passwords, or access tokens in this environments for any purposes.

Your Task

1. If you haven't generated a **GitHub Personal Access Token** you should do so now. You will need it to push code back to your repository. It should have `repo` and `write` permissions, and set to expire in `60` days. When Git prompts you for a password in the Cloud IDE environment, use your Personal Access Token instead.

2. The environment may be recreated at any time so you may find that you have to perform the **Initialize Development Environment** each time the environment is created.

3. Create a repository from the GitHub template provided for this lab in the next step.

Create your own GitHub Repository

You will need your own repository to complete the final project. We have provided GitHub Template repository to create your own repository in your own GitHub account. No need to Fork it as it has been set up as a Template. This will avoid confusion when making Pull Requests in the future.

Your Task

1. In a browser, visit this GitHub repository:
   https://github.com/ibm-developer-skills-network/xgcyk-tdd-bdd-final-project-template

2. From the GitHub **Code** tab, press the green **Use this template** button to create your own repository from this template.

3. Select **Create a new repository** from the dropdown menu

On the next screen, fill out these prompts following the screenshot below:

Create a new repository from xgcyk-t template

The new repository will start with the same files and folders as il final-project-template.

Owner * 1 **Repository name** *

rofrano / tdd-bdd-final-project

Great repository names are short and memorable. Need inspirat

Description (optional) 3

Final project for the IBM Introduction to TDD/BDD course on C

Public 4

Anyone on the internet can see this repository. You choose wh

Private

You choose who can see and commit to this repository.

Include all branches

Copy all branches from ibm-developer-skills-network/xgcyk-tdd-bdd

You are creating a public repository in your personal account

1. Select your GitHub account from the dropdown list
2. Name the new repository: `tdd-bdd-final-project`
3. (Optional) Add a nice description to let people know what this repo is for
4. Make the repo **Public** so that others can see if (and grade it)
5. Press the **Create repository from template** button to create the repository in the GitHub account you have selected.

Note: These steps only need to be done once. Whenever you re-enter this lab, you should start from the next page: **Initialize Development Environment**

Initialize Development Environment

As previously covered, the Cloud IDE with Docker environment is ephemeral, and may be deleted at any time. The next time you come into the lab, a new environment may be created. Unfortunately, this means that you will need to initialize your development environment every time it is recreated. This shouldn't happen too often as the environment can last for several days at a time but when it is removed, this is the procedure to recreate it.

Overview

Each time you need to set up your lab development environment, you will need to run three commands.

Each command will be explained in further detail, one at a time, in the following section.

`{your_github_account}` represents your GitHub account username.

The commands include:

```
git clone https://github.com/{your_github_account}/tdd-bdd-final-project.git
cd tdd-bdd-final-project
bash ./bin/setup.sh
exit
```

Now, let's discuss each of these commands and explain what needs to be done.

Task Details

Initialize your environment using the following steps:

1. Open a terminal with `Terminal` -> `New Terminal` if one isn't open already.
2. Next, use the `export GITHUB_ACCOUNT=` command to export an environment variable that contains the name of your GitHub account.

   Note: Substitute your real GitHub account that you used to create the repository, for the `{your_github_account}` place holder below:

   ```
   export GITHUB_ACCOUNT={your_github_account}
   ```

3. Then use the following commands to clone your repository, change into the `devops-capstone-project` directory, and execute the `./bin/setup.sh` command.

   ```
   git clone https://github.com/$GITHUB_ACCOUNT/tdd-bdd-final-project.git
   cd tdd-bdd-final-project
   bash ./bin/setup.sh
   ```

You should see the follow at the end of the setup execution:

4. Finally, use the `exit` command to close the current terminal. The environment won't be fully active until you open a new terminal in the next step.

```
exit
```

Validate

In order to validate that your environment is working correctly, you must open a new terminal because the Python virtual environment will only activate when a new terminal is created. You should have ended the previous task by using the `exit` command to exit the terminal.

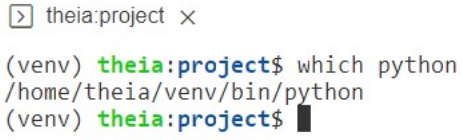
1. Open a terminal with `Terminal` -> `New Terminal` and check that everything worked correctly by using the `which python` command:

   Your prompt should look like this:

   Check which Python you are using:

```
which python
```

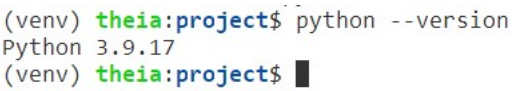
You should get back:



Check the Python version:

```
python --version
```

You should get back some patch level of Python 3.8:



This completes the setup of the development environment. Anytime your environment is recreated, you will need to follow this procedure.

You are now ready to start working.

Final Project Scenario

You have been asked by the product catalog manager at your company to develop an Product microservice to build a catalog for your e-commerce website. The user interface (UI) has been developed by another team and will be used by product administrators to maintain the product catalog. Since it is a microservice, it is expected to have a well-formed REST API that the UI and other microservices can call. This service initially needs the ability to create, read, update, delete, and list products by various attributes.

You have also been informed that someone else has started on this project and has already developed the database model and a Python Flask-based REST API with an endpoint to **create** products. You will already have this code in your copy of the repository.

In the first part of this project you will use good **Test Driven Development** practices to create a **REST API** that allows users to Create, Read, Update, Delete, and List products by various attributes. As mentioned, the Create implementation and tests will be provided for you to use as an example for your code.

In the second part of this project you will write **Behavior Driven Development** scenarios to test that the administrative user interface, which has been provided for you, behaves as expected. The scenario for **Create** is already implemented. You will also need to write code to load the background data, and more scenarios to fill out the allowed actions.

Good Luck!

Exercise 1: Create Fake Products

Update the ProductFactory class

Open the `service/models.py` file to familiarize yourself with the attributes of the `Product` class. These are the same attributes that you will need to add to the `ProductFactory` class in the `tests/factories.py` file.

Open **models.py** in IDE

Open the `tests/factories.py` file in the IDE editor. This is the file in which you will add the attributes of the `Product` class to the `ProductFactory` class.

Open **factories.py** in IDE

Your Task

In the `tests/factories.py` file, use the **Faker** providers and **Fuzzy** attributes to create fake data for the `name`, `description`, `price`, `available`, and `category` fields by adding them to the `ProductFactory` class.

You can also refer to the video Factories and Fakes and the lab Using Factories and Fakes in case you want to familiarise yourself with the concepts before proceeding further.

▶ Click here for a hint.

Solution

▶ Click here for the solution.

Now you can move on to writing test cases for the model.

Exercise 2: Create Test Cases for the Model

Before we proceed with creating test cases for the Product Model, let us see the different attributes used in this model:

Attribute	Description
id	The id of the product
name	The name of the product
description	The description of the product
price	The price of the product
available	True if the product is available, otherwise False
category	The category under which the product belongs

Please refer to the Reading: About the Product Model (include link to the course reading) to understand the classes and methods used in the models.py file.

The first thing you need to do while creating test cases for the Model, is ensure that the database model is working correctly. Someone wrote all of the code but only wrote a test case for create. You have no idea if the other functions work properly or not.

Your Task

Create test cases in `tests/test_models.py` to exercise the code in `service/models.py` to test that the `Product` model works.

1. Write test case to **read** a product and ensure that it passes
2. Write test case to **update** a product and ensure that it passes
3. Write test case to **delete** a product and ensure that it passes
4. Write test case to **list all** products
5. Write test case to search for a product by **name** and ensure that it passes
6. Write test case to search for a product by **category** and ensure that it passes
7. Write test case to search for a product by **availability** and ensure that it passes

Open **test_models.py** in IDE

Note: To open in file explorer go to the location:

`/home/project/tdd-bdd-final-project/tests/test_models.py`

Test Case to Read a Product

1. Create a `Product` object using the `ProductFactory`
2. Add a log message displaying the `product` for debugging errors
3. Set the ID of the `product` object to `None` and then create the product.
4. Assert that the `product ID` is not None
5. Fetch the product back from the database
6. Assert the properties of the found product are correct

Overall, this test case should verify if the reading functionality of the Product class is working correctly. It should create a product, assign it an ID, save it to the system, retrieve it back using the ID, and verify that the retrieved product has the same properties as the original product.

▶ Click here for a hint.
▶ Click here to check your solution.

Test Case to Update a Product

1. Create a `Product` object using the `ProductFactory`
2. Add a log message displaying the `product` for debugging errors
3. Set the ID of the `product` object to `None` and create the product.
4. Log the `product` object again after it has been created to verify that the product was created with the desired properties.
5. Update the description property of the `product` object.
6. Assert that that the `id` and `description` properties of the `product` object have been updated correctly.
7. Fetch all products from the database to verify that after updating the product, there is only one product in the system.
8. Assert that the fetched product has the original id but updated description.

Overall, this test case should verify if the update functionality of the Product class is working correctly. It should create a product, save it to the system, update its properties, verify the updated properties, fetch the product back from the system, and confirm that the fetched product has the updated properties.

▶ Click here for a hint.
▶ Click here to check your solution.

Test Case to Delete a Product

1. Create a `Product` object using the `ProductFactory` and save it to the database.
2. Assert that after creating a product and saving it to the database, there is only one product in the system.
3. Remove the product from the database.
4. Assert if the product has been successfully deleted from the database.

Overall, this test case should verify if the deletion functionality of the Product class is working correctly. It should create a product, save it to the database, delete it, and verify that the product is no longer present in the database.

▶ Click here for a hint.
▶ Click here to check your solution.

Test Case to List all Products

1. Retrieve all products from the database and assign them to the products variable.
2. Assert there are no products in the database at the beginning of the test case.
3. Create five products and save them to the database.
4. Fetching all products from the database again and assert the count is 5

Overall, this test case should verify if the listing functionality of the Product class is working correctly. It should check that initially there are no products, create five products, and confirm that the count of the retrieved products matches the expected count of 5.

▶ Click here for a hint.
▶ Click here to check your solution.

Test Case to Find a Product by Name

1. Create a batch of 5 Product objects using the ProductFactory and save them to the database.
2. Retrieve the name of the first product in the products list
3. Count the number of occurrences of the product name in the list
4. Retrieve products from the database that have the specified name.
5. Assert if the count of the found products matches the expected count.
6. Assert that each product's name matches the expected name.

Overall, this test case should verify if the `Product.find_by_name()` method correctly retrieves products from the database based on their name, by creating a batch of products, saving them to the database, finding products by name, and verifying that the count and names of the found products match the expected values.

▶ Click here for a hint.
▶ Click here to check your solution.

Test Case to Find a Product by Availability

1. Create a batch of 10 Product objects using the ProductFactory and save them to the database.
2. Retrieve the availability of the first product in the products list
3. Count the number of occurrences of the product availability in the list
4. Retrieve products from the database that have the specified availability.
5. Assert if the count of the found products matches the expected count.
6. Assert that each product's availability matches the expected availability.

Overall, this test case should verify if the `Product.find_by_availability()` method correctly retrieves products from the database based on their availability, by creating a batch of products, saving them to the database, finding products by availability, and verifying that the count and availability of the found products match the expected values.

▶ Click here for a hint.
▶ Click here to check your solution.

Test Case to Find a Product by Category

1. Create a batch of 10 Product objects using the ProductFactory and save them to the database.
2. Retrieve the category of the first product in the products list
3. Count the number of occurrences of the product that have the same category in the list.
4. Retrieve products from the database that have the specified category.
5. Assert if the count of the found products matches the expected count.
6. Assert that each product's category matches the expected category.

Overall, this test case should verify if the `Product.find_by_category()` method correctly retrieves products from the database based on their category, by creating a batch of products, saving them to the database, finding products by category, and verifying that the count and categories of the found products match the expected values.

▶ Click here for a hint.
▶ Click here to check your solution.

Acceptance Criteria

- Ensure that all tests pass when you run `nosetests`, and maintain at least 95% code coverage.
- There should be no linting errors when you run `make lint`

REST API Guidelines Review

For your review, these are the guidelines for creating REST APIs that enable you to write the test cases for this lab:

RESTful API Endpoints

Action	Method	Return code	Body	URL Endpoint
List	GET	200_OK	Array of products `[{...}]`	`GET /products`
Create	POST	201_CREATED	A product as json `{...}`	`POST /products`
Read	GET	200_OK	A product as json `{...}`	`GET /products/{id}`
Update	PUT	200_OK	A product as json `{...}`	`PUT /products/{id}`
Delete	DELETE	204_NO_CONTENT	`""`	`DELETE /products/{id}`

Following these guidelines, you can make assumptions about how to call the web service and assert what it should return.

HTTP Status Codes

Here are some other HTTP status codes that you will need for this lab:

Code	Status	Description
200	HTTP_200_OK	Success
201	HTTP_201_CREATED	The requested resource has been created
204	HTTP_204_NO_CONTENT	There is no further content
404	HTTP_404_NOT_FOUND	Could not find the resource requested
405	HTTP_405_METHOD_NOT_ALLOWED	Invalid HTTP method used on an endpoint
409	HTTP_409_CONFLICT	There is a conflict with your request

All of these codes are defined in `service/common/status.py` and are already imported for your use.

Exercise 3: Create Test Cases and Code for REST API

Now that you know that the model is working, you can move on to developing the REST API which will accept `json` requests, call the model, and return an answer as `json`.

Your REST API must implement the following endpoints:

- Create a Product
- Read a Product
- Update a Product
- Delete a Product
- List all Products
- List Products by Category, Availability, and Name

Note: The Create endpoint is already implemented for you and should serve as an example. Also, there is only one List endpoint that takes optional parameters to filter the data.

Your Task

Edit the test cases in `tests/test_routes.py` and code in `service/routes.py` for a RESTful endpoint using Flask that contains the following endpoints:

1. Write a test case to **Read** a Product and watch it fail
2. Write the code to make the Read test case pass
3. Write a test case to **Update** a Product and watch it fail
4. Write the code to make the Update test case pass
5. Write a test case to **Delete** a Product and watch it fail
6. Write the code to make the Delete test case pass
7. Write a test case to **List all** Products and watch it fail
8. Write the code to make the List all test case pass
9. Write a test case to **List by name** a Product and watch it fail
10. Write the code to make the List by name test case pass
11. Write a test case to **List by category** a Product and watch it fail
12. Write the code to make the List by category test case pass

13. Write a test case to **List by availability** a Product and watch it fail
14. Write the code to make the List by availability test case pass

Following test driven development, you write a test case to assert that the code you are about to write will have the correct behavior as expected.

Task 1: Write a Test Case to Read a Product

Write the test case in tests/test_routes.py for **Reading a Product**

Open **test_routes.py** in IDE

Note: To open in file explorer go to the location:

`/home/project/tdd-bdd-final-project/tests/test_routes.py`

1. Create a test case called `test_get_product(self)`.
2. Use the `_create_products()` method to create one product, and then assign the first product from the returned list to the `test_product` variable.
3. Make a `GET` request to the API endpoint to retrieve the product and construct the URL by appending the `test_product.id` to the `BASE_URL`.
4. Assert that the return code was `HTTP_200_OK`, to verify that the request was successful and the product was retrieved.
5. Check the `json` that was returned and `assert` that it is equal to the data that you sent.
6. Run `nosetests` and watch it fail because there is no code yet.

▶ Click here for a hint.
▶ Click here to check your solution.

Task 2: Write the Code to make the Read Test case pass

Once you have a test case, you can begin to write the code in service/routes.py to make it pass.

Open **routes.py** in IDE

Note: To open in file explorer go to the location:

`/home/project/tdd-bdd-final-project/service/routes.py`

1. Create a Flask route that responds to the `GET` method for the endpoint `/products/<product_id>`.
2. Create a function called `get_products(product_id)` to hold the implementation.
3. Call the `Product.find()` method which will return a product with the given `product_id`.
4. Abort with a return code `HTTP_404_NOT_FOUND` if the product was not found.
5. Call the `serialize()` method on a product to serialize it to a Python dictionary.
6. Send the serialized data and a return code of `HTTP_200_OK` back to the caller.
7. Run `nosetests` until all of the tests are `green`, which means they passed.

▶ Click here for a hint.
▶ Click here to check your solution.

Maintain Code Coverage

You must maintain code coverage of **95%** or greater. You will not achieve this by only testing the happy paths. The test case you wrote probably did not test for a product that was not found, so you will need to write another test case that reads a product with an product id that does not exist. This should get your test coverage back up to where it needs to be.

1. Create a test case called `test_get_product_not_found(self):`.
2. Make a `self.client.get()` call to `/products/{id}` passing in an invalid product id `0`
3. Assert that the return code was `HTTP_404_NOT_FOUND`.
4. Run `nosetests` and fix the code in `test_routes.py` until it passes.

▶ Click here for a hint.
▶ Click here to check your solution.

Acceptance Criteria

- Ensure that all tests pass when you run `nosetests`, and maintain at least 95% code coverage.
- There should be no linting errors when you run `make lint`

TDD Hints and Solutions

This page contains the remaining hints and solutions for the **Update**, **Delete** , **List all** , **List By Name**, **List By Category**, **List By Availability** REST APIs, now that you have implemented **Read**.

Update

First write a test for the **Update** function:

▶ Click here for a hint.
▶ Click here to check your solution.

Now write the code to make the **Update** test case pass:

▶ Click here for a hint.
▶ Click here to check your solution.

Delete

First write a test for the **Delete** function:

▶ Click here for a hint.
▶ Click here to check your solution.

Now write the code to make the **Delete** test case pass:

▶ Click here for a hint.
▶ Click here to check your solution.

List All

First write a test for the **List All** function:

▶ Click here for a hint.
▶ Click here to check your solution.

Now write the code to make the **List All** test case pass:

▶ Click here for a hint.
▶ Click here to check your solution.

List By Name

First write a test for the **List By Name** function:

▶ Click here for a hint.
▶ Click here to check your solution.

> Note: Please import `quote_plus` by including the below line in the `test_routes.py` to ensure the `query by name` test case passes. Please add it above `import app`

```
from urllib.parse import quote_plus
```

Now write the code to make the **List By Name** test case pass:

> Note: List by name is an extension of List All. You are going to add a filter to the code that lists all products to check if the `name` parameter has been passed in and filter by name if it is, and return all if it doesn't:

▶ Click here for a hint.
▶ Click here to check your solution.

List By Category

First write a test for the **List By Category** function:

▶ Click here for a hint.
▶ Click here to check your solution.

Now write the code to make the **List By Category** test case pass:

> Note: List by Category is an extension of List All. You are going to add a filter to the code that lists all products to check if the `category` parameter has been passed in and filter by category if it is, and return all if it doesn't:

▶ Click here for a hint.
▶ Click here to check your solution.

> Note: Please import `Category` from `service.models` by including the below line in the `routes.py` to ensure the `query by category` test case passes.

```
from service.models import Product, Category
```

List By Availability

First write a test for the **List By Availability** function:

▶ Click here for a hint.
▶ Click here to check your solution.

Now write the code to make the **List By Availability** test case pass:

Note: List by Availability is an extension of List All. You are going to add a filter to the code that lists all products to check if the `available` parameter has been passed in and filter by available if it is, and return all if it doesn't:

- ▶ Click here for a hint.
- ▶ Click here to check your solution.

Web site User Interface

The administrative user interface for this web site has already been provided for you by another team. Unfortunately, that team did not use good Behavior Driven Development techniques and so there are no test cases for the UI.

Your next two exercises involve creating test cases in the form of BDD scenarios to test that the UI behaves as expected. In order to write these test cases, you need to know what the UI looks like. It is a minimalist administrative interface. Below is an image of the web page that you are testing:

Product Catalog Administra

Status:

Create, Retrieve, Update, and Delete a

Product ID: Enter ID of Product

Name: Enter name for Product

Description: Enter description for Product

Available: True

Category: Unknown

Price: Enter price for Product

Search Clear Create U

Exercise 4: Load BDD background data

The first thing you will need to do for BDD testing, is write the Python code to load the data from the `background:` statement in the `products.feature` file. Remember that the data is stored in the `context.table` attribute and each `row` is a Python dictionary (`dict`) that you can dereference using the names at the top of the columns in the `background:` statement.

Your Task

Update the `features/steps/load_steps.py` file to load background data from your BDD scenarios into your service before each scenario executes.

The code to delete all of the products is already given to you. You just need to write the code to load the products from `context.table`.

Open **load_steps.py** in IDE

Note: To open in file explorer go to the location:

```
/home/project/tdd-bdd-final-project/features/steps/load_steps.py
```

▶ Click here for a hint.
▶ Click here to check your solution.

Exercise 5: Create BDD Scenarios

Now that the background data is loaded, it's time to write the scenarios to test the UI. The **Create a Product** scenario is already written as an example of what you need to do.

The service already contains a UI that looks like this:

Product Catalog Administra

Status:

Create, Retrieve, Update, and Delete a

Product ID: Enter ID of Product

Name: Enter name for Product

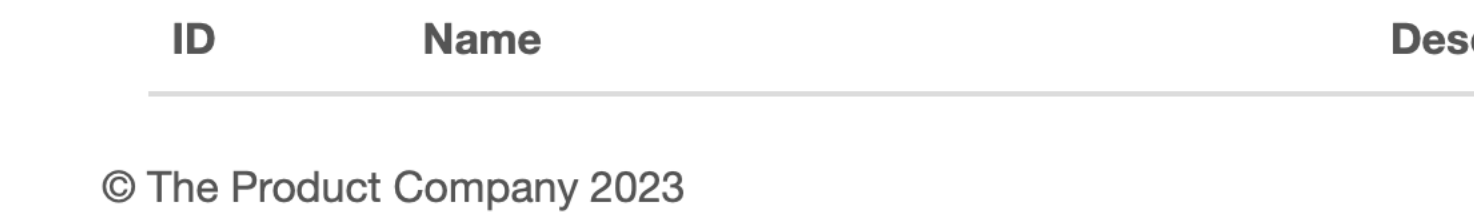
Description: Enter description for Product

Available: True

Category: Unknown

Price: Enter price for Product

Search Clear Create U



Your Task

1. Update the `features/products.feature` file with BDD Scenarios that prove that the following behaviors of the UI work as expected:
    - Read a Product
    - Update a Product
    - Delete a Product
    - List all Products
    - Search for Products by Category
    - Search for Products by Availability
    - Search for Products by Name

Start by opening the file `petshop.feature`.

Open **products.feature** in IDE

Note: To open in file explorer go to the location:

`/home/project/tdd-bdd-final-project/features/products.feature`

BDD Scenario for Reading a Product

Document the steps that the customer needs to take:

1. Use the `Scenario:` keyword with the title **"Read a Product"**
2. Use the `When` keyword to specify to start on the **"Home Page"**
3. Use the `And` keyword to describe the action of setting the **"Name"** to **"Hat"**
4. Use the `And` keyword to describe the action of clicking the **"Search"** button
5. Use the `Then` keyword to describe that they should see the message **"Success"**
6. Use the `When` keyword to specify copy the **"Id"** field
7. Use the `And` keyword to describe the action of pressing the **"Clear"** button
8. Use the `And` keyword to describe the action of pasting the **"Id"** field
9. Use the `And` keyword to describe the action of pressing the **"Retrieve"** button
10. Use the `Then` keyword to describe that they should see the message **"Success"**
11. Use the `And` keyword to state that they should see **"Hat"** in the **"Name"** field
12. Use the `And` keyword to state that they should see **"A red fedora"** in the **"Description"** field
13. Use the `And` keyword to state that they should see **"True"** in the **"Available"** dropdown
14. Use the `And` keyword to state that they should see **"Cloths"** in the **"Category"** dropdown
15. Use the `And` keyword to state that they should see **"59.95"** in the **"Price"** field

▶ Click here for a hint.
▶ Click here for the answer.

Now that you have written the scenario for **Read a Product** follow the same lines for the remaining scenarios.

BDD Hints and Solutions

This page contains the remaining hints and solutions for the **Update**, **Delete** , **Search for Products by Category** , **Search for Products by Availability**, **Search for Products by Name** scenarios.

BDD Scenario for Updating a Product

▶ Click here for a hint.
▶ Click here for the answer.

BDD Scenario for Deleting a Product

▶ Click here for a hint.
▶ Click here for the answer.

BDD Scenario for Listing all Products

▶ Click here for a hint.

▶ Click here for the answer.

BDD Scenario for Searching a Product based on Category

▶ Click here for a hint.
▶ Click here for the answer.

BDD Scenario for Searching a Product based on Availability

▶ Click here for a hint.
▶ Click here for the answer.

BDD Scenario for Searching a Product based on Name

▶ Click here for a hint.
▶ Click here for the answer.

Your Task

1. Start the service in a Terminal with the command `honcho start`
2. In another Terminal, run the `behave` tool to seewhat steps need to be implemented. You will implement them next.

Acceptance Criteria

- Ensure that you see suggested step definitions from `behave`.

Exercise 6: Implementing Steps

In BDD, test scenarios are often written in a human-readable format, such as Gherkin, and these scenarios are translated into automated tests using step definitions.

The `web_steps.py` file contains the step definitions for the web-related actions on the UI.

Please check [Hands-on Lab: Implementing Your First Steps](link to the lab page to be included) for any reference you may need to understand about how step definitions are implemented, before proceeding further.

The step definitions for the first few steps are already given to you.

Your Task

Update the `features/steps/web_steps.py` file with the remaining step definitions.

Start by opening the file `web_steps.py`.

Open **web_steps.py** in IDE

Button-Click

▶ Click here for a hint.
▶ Click here for the answer.

Verify for a specific name or text to be present

▶ Click here for a hint.
▶ Click here for the answer.

Verify for a specific name or text to NOT be present

▶ Click here for a hint.
▶ Click here for the answer.

Verify that a specific message is present

▶ Click here for a hint.
▶ Click here for the answer.

Your Task

1. Start the service in a Terminal with the command `honcho start`
2. In another Terminal, run the `behave` tool to make sure that all seven (7) scenarios pass

Acceptance Criteria

- Ensure that all seven (7) scenarios exist and pass

Submission

Commit the code to your Github repository

1. Use `git status` to make sure that you have committed your changes locally in the development environment.
2. Use the `git add` command to add the updated code to the staging area.
3. Commit your changes using `git commit -m <commit message>`
4. Push your local changes to a remote branch using the `git push` command

    Note: Use your GitHub **Personal Access Token** as your password in the Cloud IDE environment. You may also need to configure Git the first time you use it with:

    ```
    git config --local user.email "you@example.com"
    git config --local user.name "Your Name"
    ```

Submit a the link to your GitHub repository when completed.

Evaluation

Use the lab environment to clone the project from the GitHub link provided

- Run `nosetests` and ensure that all tests pass and test coverage is 95% or better
- Run the application with `honcho start` in on terminal shell
- In another terminal shell run the `behave`command and ensure that there are seven (7) scenarios (one each for Read a Product, Update a Product, Delete a Product, List all Products, List by Category, List by Available, and List by Name) and that all scenarios pass.

Conclusion

Congratulations on completing the TDD / BDD Final Project. Hopefully now you have proven to yourself that you understand how to implement good test Driven and behavior Driven Development practices.

Next Steps

Incorpoate these new practces into your projects an home and at work. Write the test cases for the code you “wish you had” and then write the code to make those tests pass. Describe the behavior of you system from the outside in and then prove that it behaves that way by automating those tests with Behave.

Author(s)

John J. Rofrano

Other Contributor(s)

Lavanya Rajalingam