Hands-On Lab: Practicing Test Driven Development

Estimated time needed: 30 minutes

Welcome to the **Practicing Test Driven Development** lab. Test driven development (TDD) is an approach to software development in which you first write the test cases for the code you *wish* you had and then write the code to make the test cases pass.

In this lab, you will write test cases based on the requirements given to you, and then you will write the code to make the test cases pass. You will find out how easy this is.

Learning Objectives

After completing this lab, you will be able to:

- Follow the TDD workflow
- Write test cases based on an application requirements
- Write code to make test cases pass

About Theia

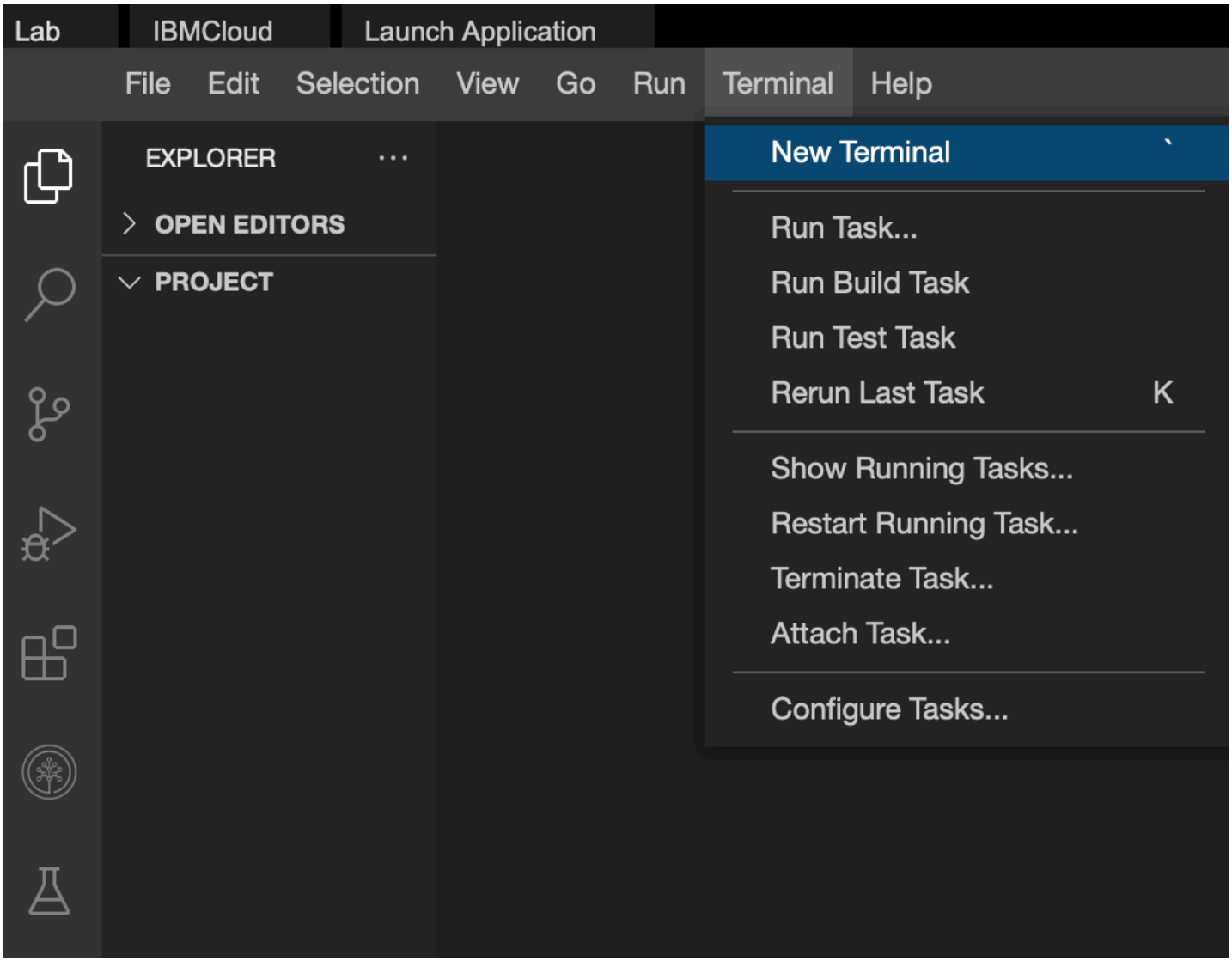
Theia is an open-source IDE (Integrated Development Environment) that can be run on desktop or on cloud. You will be using the Theia IDE to do this lab. When you log into the Theia environment, you are presented with a ‘dedicated computer on the cloud’ exclusively for you. This is available to you as long as you work on the labs. Once you log off, this ‘dedicated computer on the cloud’ is deleted along with any files you may have created. So, it is a good idea to finish your labs in a single session. If you finish part of the lab and return to the Theia lab later, you may have to start from the beginning. Plan to work out all your Theia labs when you have the time to finish the complete lab in a single session.

Set Up the Lab Environment

You have a little preparation to do before you can start the lab.

Open a Terminal

Open a terminal window by using the menu in the editor: Terminal > New Terminal.



In the terminal, if you are not already in the `/home/projects` folder, change to your project folder now.

```
cd /home/project
```

Clone the Code Repo

Now get the code that you need to test. To do this, use the `git clone` command to clone the git repository:

```
git clone https://github.com/ibm-developer-skills-network/duwjx-tdd_bdd_PracticeCode.git
```

Change into the Lab Folder

Once you have cloned the repository, change to the lab directory:

```
cd duwjx-tdd_bdd_PracticeCode/labs/07_practicing_tdd
```

Install Python Dependencies

The final preparation step is to use `pip` to install the Python packages needed for the lab:

```
python3.8 -m pip install -r requirements.txt
```

You are now ready to start the lab.

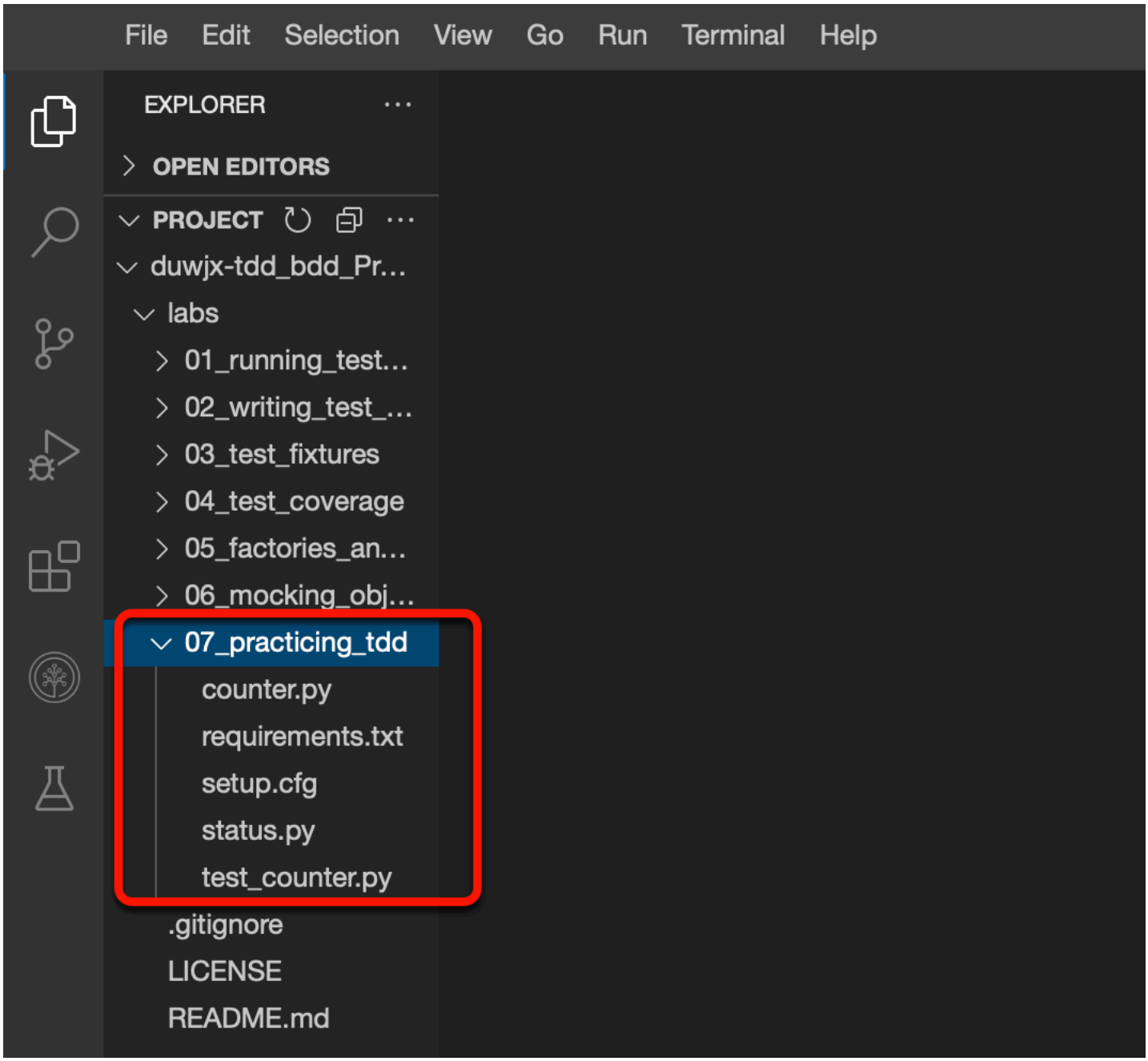
Optional

If working in the terminal becomes difficult because the command prompt is very long, you can shorten the prompt using the following command:

```
export PS1="[\[\033[01;32m\]\u\[\033[00m\]: \[\033[01;34m\]\W\[\033[00m\]]\$ "
```

Navigate to the Code

In the IDE on the left of your screen, navigate to the `duwjx-tdd_bdd_PracticeCode/labs/07_practicing_tdd` folder. This folder contains all of the source code that you will use for this lab.



The test cases you will add to are in `test_counter.py`, and the code you will add is in `counter.py`. These are the only two files you will work with.

Requirements

Assume you have been asked to create a web service that can keep track of multiple counters. The web service has the following requirements:

- The API must be RESTful.
- The endpoint must be called `/counters`.
- When creating a counter, you must specify the name in the path.
- Duplicate names must return a conflict error code.
- The service must be able to update a counter by name.
- The service must be able to get a counter's current value.
- The service must be able to delete a counter.

The last three requirements have not been implemented. You have been asked to implement them using TDD principles by writing the test cases first, and then writing the code to make the test cases pass.

REST API Guidelines Review

There are guidelines for creating REST APIs that enable you to write the test cases for this lab:

Action	Method	Return code	URL
Create	POST	201_CREATED	POST /counters/{name}
Read	GET	200_OK	GET /counters/{name}
Update	PUT	200_OK	PUT /counters/{name}

Action	Method	Return code	URL
Delete	DELETE	204_NO_CONTENT	DELETE /counters/{name}

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

Step 1: Write a test for update a counter

You will start by implementing a test case to test updating a counter. Following REST API guidelines, an update uses a `PUT` request and returns code `200_OK` if successful. Create a counter and then update it.

Your Task

In `test_counter.py`, create a test called `test_update_a_counter(self)`. It should implement the following steps:

1. Make a call to Create a counter.
2. Ensure that it returned a successful return code.
3. Check the counter value as a baseline.
4. Make a call to Update the counter that you just created.
5. Ensure that it returned a successful return code.
6. Check that the counter value is one more than the baseline you measured in step 3.

> Hint: Use a different counter name for each test so that one test does not affect the others

Open **test_counter.py** in IDE

Solution

▼ Click here for the solution.

This is one solution for testing the functionality of update a counter.

```
    def test_update_a_counter(self):
        """It should increment the counter"""
        result = self.client.post("/counters/baz")
        self.assertEqual(result.status_code, status.HTTP_201_CREATED)
        data = result.get_json()
        baseline = data["baz"]
        # Update the counter
        result = self.client.put("/counters/baz")
        self.assertEqual(result.status_code, status.HTTP_200_OK)
        data = result.get_json()
        self.assertEqual(data["baz"], baseline + 1)
```

Run the Tests

Run `nosetests` to see the new test is failing:

```
nosetests
```

You may need to scroll up in the terminal to see all the results. The results should look like this:

```
Test Cases for Counter Web Service
- It should create a counter
- It should return an error for duplicates
- It should increment the counter (FAILED)

======================================================================
FAIL: It should increment the counter
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/rofrano/DevOps/duwjx-tdd_bdd_Practice
    self.assertEqual(result.status_code, status.HTTP
AssertionError: 405 != 200
---------------------- >> begin captured logging << --
counter: INFO: Request to create counter: baz
---------------------- >> end captured logging << ---

Name         Stmts   Miss  Cover   Missing
------------------------------------------
counter.py      11      0   100%
------------------------------------------
TOTAL           11      0   100%
----------------------------------------------------------------------
Ran 3 tests in 0.283s

FAILED (failures=1)
```

Notice that the failure is **AssertionError: 405 != 200**. A 405 is `405_METHOD_NOT_ALLOWED`, indicating that Flask found a route called `/counters/<name>` but it does not allow a `PUT` method. This is because you haven't written that code yet. That饿.

Step 2: Implement update a counter

Now you粑te the code to make the test pass. If you宦amiliar with **Flask**, note that all of the routes for the counter service are the same; only the **method** changes.

To start, you will implement a function to update the counter. Per REST API guidelines, an update uses a `PUT` request and returns a `200_OK` code if successful. Create a function that updates the counter that matches the specified name.

Your Task

In `counter.py`, create a function called `update_counter(name)`. It should implement the following steps:

1. Create a route for method PUT on endpoint `/counters/<name>`.
2. Create a function to implement that route.
3. Increment the counter by 1.
4. Return the new counter and a `200_OK` return code.

Open **counter.py** in IDE

Solution

▼ Click here for the solution.

This is one solution for implementing the functionality of update a counter.

```
@app.route("/counters/<name>", methods=["PUT"])
def update_counter(name):
    """Update a counter"""
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    COUNTERS[name] += 1
    app.logger.info(f"Counter: {name} is now {COUNTERS[name]}")
    return { name: COUNTERS[name] }, status.HTTP_200_OK
```

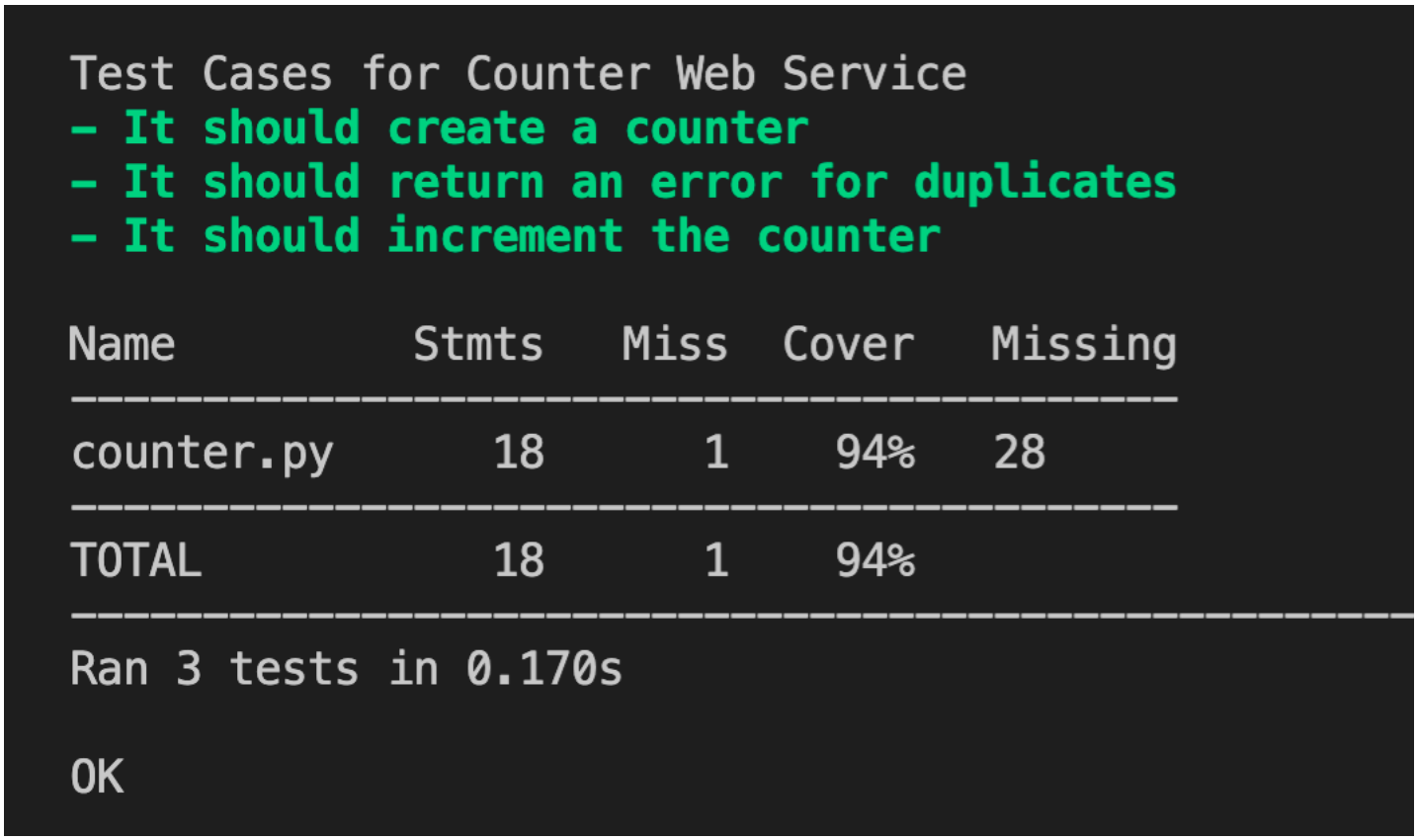
> Notice that this code does not provide any error handling. What if the counter doesn't exist? You would need a new test case and then more code to check that the counter exists before incrementing it.

Run the Tests

Run `nosetests` and make sure that the test cases pass:

```
nosetests
```

The results should look like this:



Step 3: Write a test for read a counter

Next, you will write a test case to read a counter. Following REST API guidelines, a read uses a `GET` request and returns a `200_OK` code if successful. Create a counter and then read it.

Your Task

In `test_counter.py`, create a test called `test_read_a_counter(self)`. It should implement the following steps:

1. Make a call to create a counter.
2. Ensure that it returned a successful return code.
3. Make a call to read the counter you just created.
4. Ensure that it returned a successful return code.
5. Check that the counter value returned is 0.

   Hint: Use a different counter name for each test so that one test does not affect the others.

Solution

▼ Click here for the solution.

This is one solution for testing the functionality of read a counter.

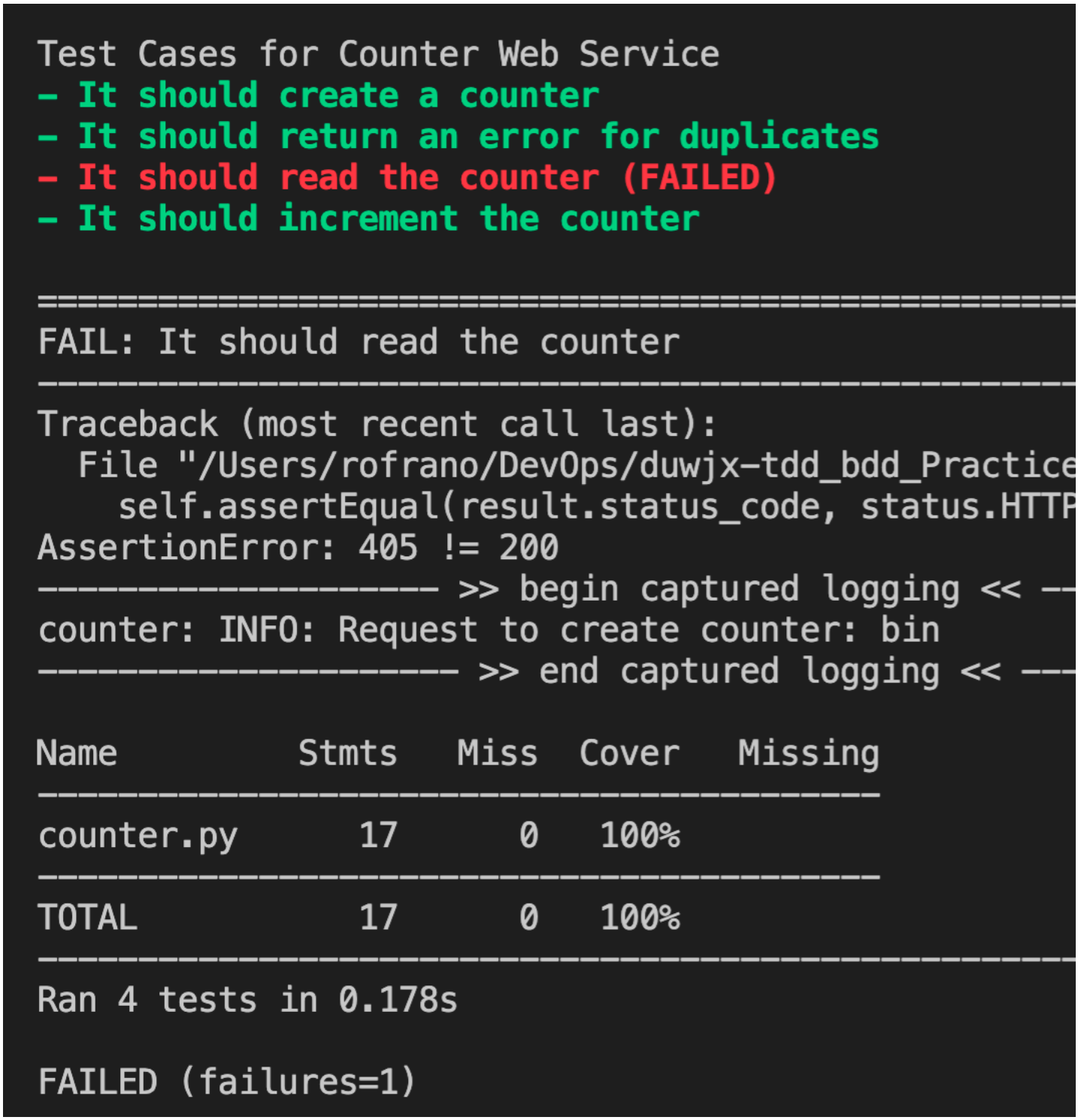
```
        def test_read_a_counter(self):
            """It should read the counter"""
            result = self.client.post("/counters/bin")
            self.assertEqual(result.status_code, status.HTTP_201_CREATED)
            # Read the counter
            result = self.client.get("/counters/bin")
            self.assertEqual(result.status_code, status.HTTP_200_OK)
            data = result.get_json()
            self.assertEqual(data["bin"], 0)
```

Run the Tests

Run `nosetests` to see that the new test is failing:

```
nosetests
```

The results should look like this:



Notice again, that the failure is **AssertionError: 405 != 200**. A 405 is `405_METHOD_NOT_ALLOWED`, indicating that Flask found a route called `/counters/<name>` but it does not allow a `GET` method. This is because you haven't written that code yet. That幾.

Step 4: Implement read a counter

Once again, it's time to write code to make a test pass. You will implement the code for read a counter. Per REST API guidelines, a read uses a `GET` request and returns a `200_OK` code if successful. Create a function that returns the counter that matches the specified name.

Your Task

In `counter.py`, create a function called `read_counter(name)`. It should implement the following steps:

1. Create a route for method `GET` on endpoint `/counters/<name>`.
2. Create a function to implement that route.
3. Get the current value of the counter.
4. Return the counter and a `200_OK` return code.

Solution

▼ Click here for the solution.

This is one solution for implementing the functionality of read a counter.

```
@app.route("/counters/<name>", methods=["GET"])
def read_counter(name):
    """Read a counter"""
    app.logger.info(f"Request to read counter: {name}")
    counter = COUNTERS[name]
    app.logger.info(f"Counter: {name} is {counter}")
    return { name: counter }, status.HTTP_200_OK
```

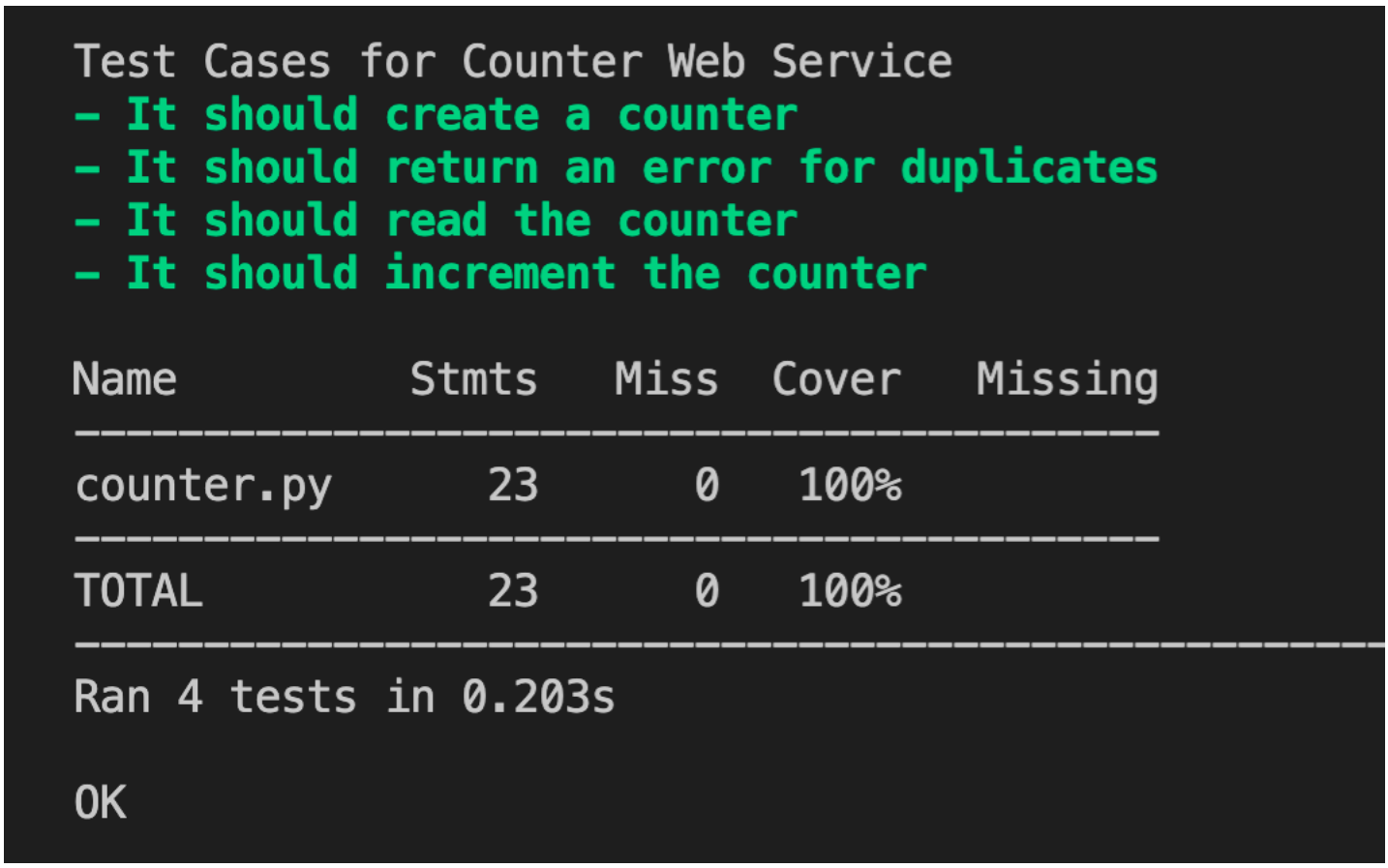
Notice that the code does not provide any error handling. Again, if the counter didn't exist, you would need a new test case and then more code to check that the counter exists before returning it.

Run the Tests

Run `nosetests` and make sure that the test cases pass:

```
nosetests
```

The results should look like this:



Step 5: Write a test for delete a counter

Now you will write a test case to delete a counter. Per REST API guidelines, a read uses a `DELETE` request and returns a `204_NO_CONTENT` code if successful. Create a function that deletes the counter that matches the specified name.

Your Task

In `test_counter.py`, create a function called `test_delete_a_counter(self)`. It should implement the following steps:

1. Make a call to Create a counter.
2. Ensure that it returned a successful return code.
3. Make a call to delete the counter you just created.
4. Ensure that it returned a successful return code.

> Hint: Use a different counter name for each test so that one test does not affect the others

Solution

▼ Click here for the solution.

This is one solution for testing the functionality of delete a counter.

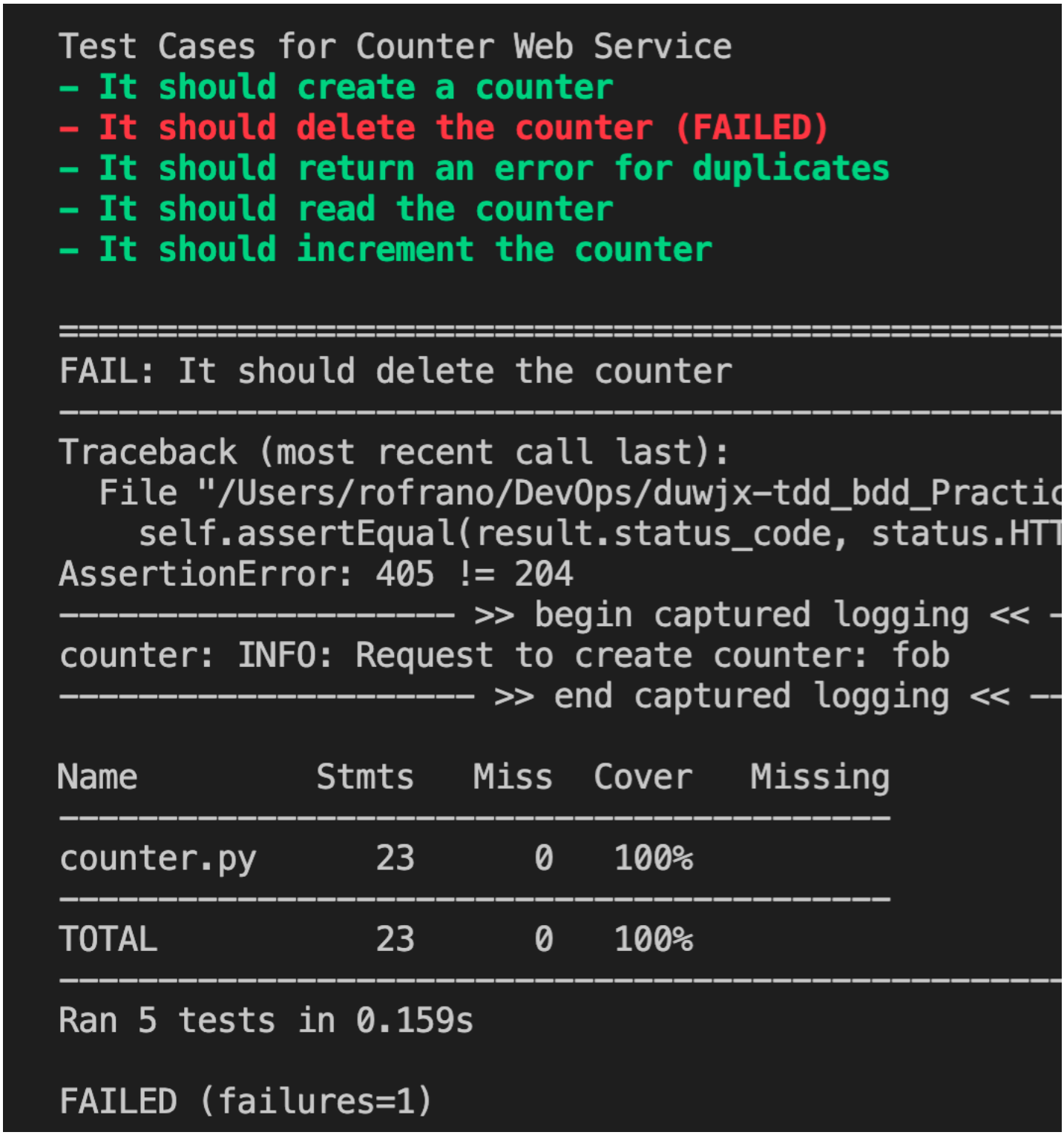
```
        def test_delete_a_counter(self):
            """It should delete the counter"""
            result = self.client.post("/counters/fob")
            self.assertEqual(result.status_code, status.HTTP_201_CREATED)
            # Delete the counter
            result = self.client.delete("/counters/fob")
            self.assertEqual(result.status_code, status.HTTP_204_NO_CONTENT)
```

Run the Tests

Run `nosetests` to see the new test is failing:

```
nosetests
```

The results should look like this:



Notice again, that the failure is **AssertionError: 405 != 204**. A 405 is `405_METHOD_NOT_ALLOWED`, indicating that Flask found a route called `/counters/<name>` but it does not allow a `DELETE` method. This is because you haven't written that code yet. That幾.

Step 6: Implement delete a counter

In this last step, you will again write code to make a test pass. This time, you will implement the code to delete a counter. Per REST API guidelines, a delete uses a `DELETE` request and returns a `204_NO_CONTENT` code if successful.

Your Task

In `counter.py`, create a function called `delete_counter(name)`. It should implement the following steps:

1. Create a route for method `DELETE` on endpoint `/counters/<name>`.
2. Create a function to implement that route.
3. Delete the counter that matches the name.
4. Return the counter and a `204_NO_CONTENT` return code.

Solution

▼ Click here for the solution.

This is one solution for implementing the functionality of delete a counter.

```
@app.route("/counters/<name>", methods=["DELETE"])
def delete_counter(name):
    """Delete a counter"""
    app.logger.info(f"Request to delete counter: {name}")
    del(COUNTERS[name])
    app.logger.info(f"Counter: {name} has been deleted")
    return '', status.HTTP_204_NO_CONTENT
```

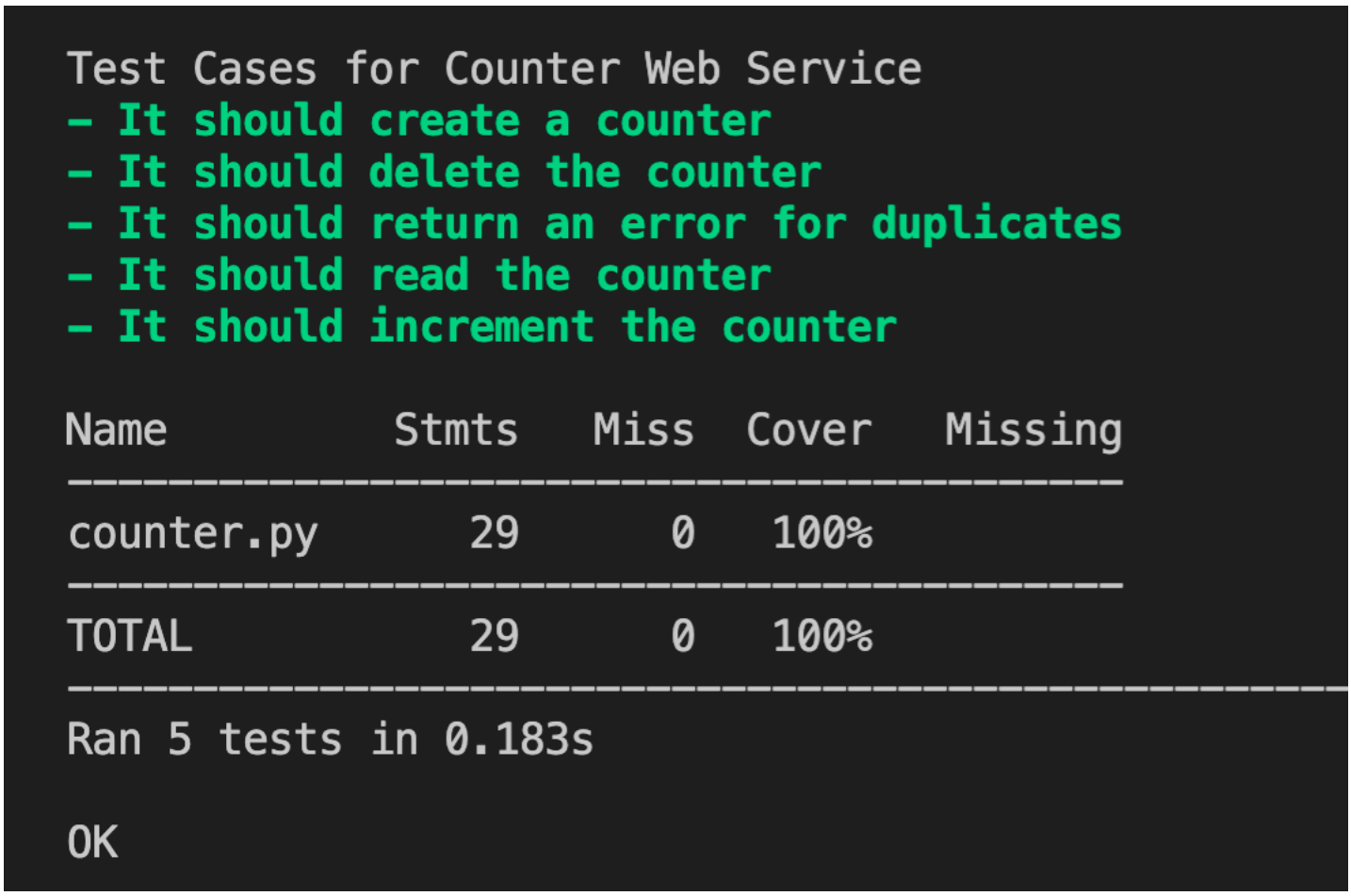
Notice that it does not provide any error handling. Again, if the counter didn't exist, you would need a new test case and then more code to check that the counter exists before deleting it.

Run the Tests

Run `nosetests` and make sure the test cases pass:

```
nosetests
```

The results should look like this:



Conclusion

Congratulations! You just completed the **Practicing Test Driven Developments** lab.

Hopefully, you now understand how easy it is to write test cases for code that doesn't exist yet. You use the application requirements to define the behavior that you are testing, and then you write the code that behaves that way.

You may have noticed that you only tested the “happy paths,” the paths where everything goes right. Every one of those functions had a “sad path” where the counter didn’t exist. Every one of those functions would blow up in production unless you write more code to handle conditions like the counter not existing. See if you can write more test cases on your own to test what happens when you try to update, read, or delete a counter that doesn’t exist. Then write the code to make those test cases pass.

You now have the tools to practice true test driven development. Your next challenge is to use these tools in your projects. Try writing the test cases first on your next project, then write the code to make them pass. Then make the code and the test cases more robust until you are confident that the code is working as designed.

Author(s)

John J. Rofrano