Test Driven Development Activity: SHEEP COMPENDIUM

In this activity, you will learn how to create test cases for FastAPI applications using Pytest. We will be creating a farmer's inventory of their favorite sheep, specifying their name, breed, and sex.

Step 1: Install Dependencies

Open your terminal and run the following command to install the necessary dependencies:

```
pip install fastapi uvicorn pytest httpx
```

Step 2: Setting Up Our Project Structure

1. First, create a new Project called "sheep_compendium."
2. Inside the project folder, create the following directories:
    - `models`
    - `tests`

3. **In each folder, create an empty file named:**

`__init__.py`

This helps Python identify each of these folders as packages.

Step 3: Writing Our First Test Case

In test-driven development, we write test cases before the actual feature is implemented. But before we do that, let's set up our FastAPI app.

1. **Inside the root directory, add the following file.**

   `main.py`

2. **Add the following code to `main.py`:**

   a.
   ```python
   from fastapi import FastAPI

   app = FastAPI()
   ```

   b. This code initializes a basic FastAPI application. It imports the necessary components for creating the app and defining data models.

3. Now, create a file named **test_main.py** in the *tests* folder.
4. Enter the following code into **test_main.py.**

```python
# Import TestClient to simulate API requests
from fastapi.testclient import TestClient

# Import the FastAPI app instance from the controller module
from main import app

# Create a TestClient instance for the FastAPI app
client = TestClient(app)
```

5. **Then, add the following function to the code.**

```
# Define a test function for reading a specific sheep
def test_read_sheep():
    # Send a GET request to the endpoint "/sheep/1"
    response = client.get("/sheep/1")

    # Assert that the response status code is 200 (OK)
    assert response.status_code == 200

    # Assert that the response JSON matches the expected data
    assert response.json() == {
        # Expected JSON structure
        "id": 1,
        "name": "Spice",
        "breed": "Gotland",
        "sex": "ewe"
    }
```

   a. **Explanation**
      i. In this function, we will send a GET request to the server to retrieve a sheep with a specific ID.
      ii. We will use the **assert** statement to determine whether or not certain conditions are met. For example, we see the command:
         1. *assert response.status_code == 200*
            a. This will check whether the request was successful. A status code of 200 means that everything went as expected.
      iii. Next, we determine what we expect to receive if we send a GET request for the sheep that has an ID of 1. We expect a JSON structure with certain attributes, like the name of the sheep being *Spice*.

Spice is a gorgeous Gotland sheep that Professor Wiktor encountered in the NJ Sheep and Fiber festival in September 2024. Here's a picture of the beautiful animal, so you know who we are working with.

Step 4: Running our First Test Case

1. Now, run the following command in your terminal:

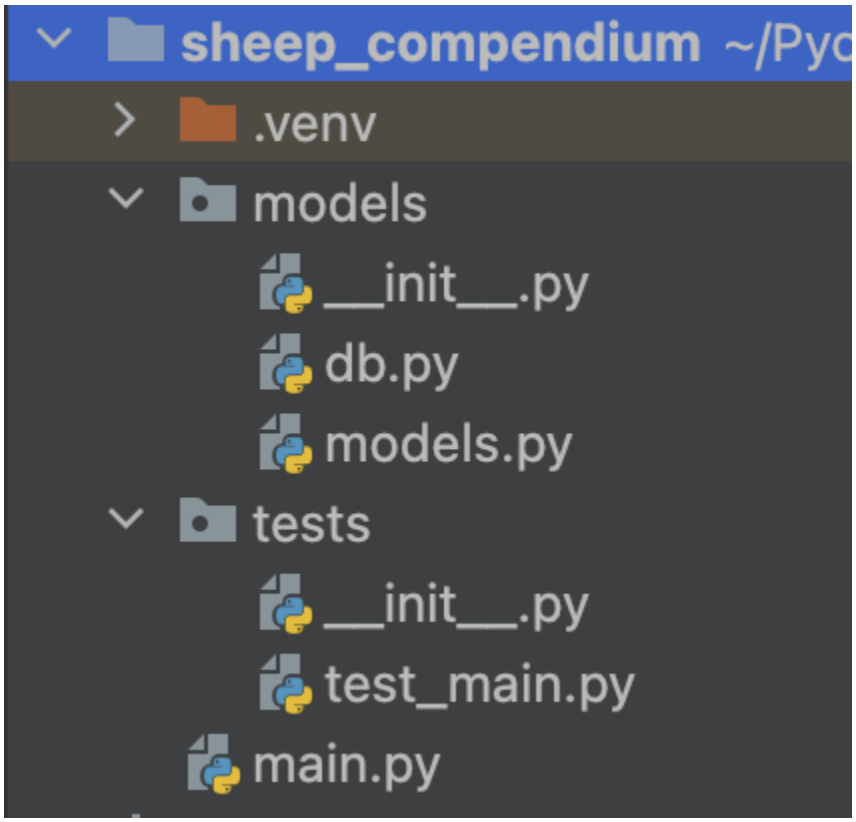
```
(.venv) (base) sandrawiktor@Lotus sheep_compendium % pytest
```

If everything is set up correctly, you should get the following result:

```
=================================== FAILURES ===================================
_____________________________ test_read_sheep _____________________________

    def test_read_sheep():
        # Send a GET request to the endpoint "/sheep/1"
        response = client.get("/sheep/1")

        # Assert that the response status code is 200 (OK)
>       assert response.status_code == 200
E       assert 404 == 200
E        +  where 404 = <Response [404 Not Found]>.status_code

tests/test_main.py:17: AssertionError
=========================== short test summary info ============
FAILED tests/test_main.py::test_read_sheep - assert 404 == 200
=========================== 1 failed in 0.30s ===============
```

Looks like we failed our test case! Well, it's because we didn't actually implement our functionality. So, we'll go ahead and do that next.

Checkpoint: *Your file structure should currently look like this.*



Troubleshooting:

If you are getting the following error:

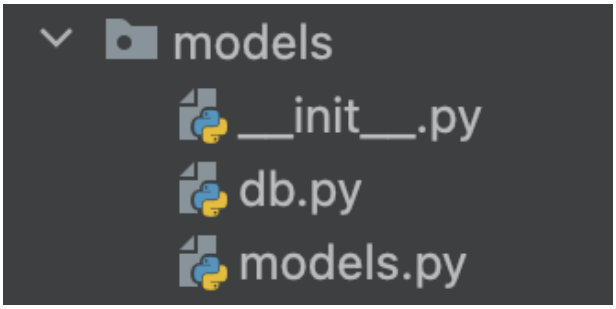
```
tests/test_main.py:2: in <module>
    from main import app
E   ModuleNotFoundError: No module named 'main'
```

Add the following to the top of your test_main.py file:

```
import sys
import os
# Add the project root (sheep directory) to sys.path
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__),
'..')))
```

Step 5: Creating an In-Memory Database of Sheep

1. Start by creating two new files in the models folder as such:



   a. db.py
      i. We'll create a simple in-memory database to store information about sheep. Instead of setting up a full SQL database, we'll use a Python dictionary to simulate database behavior, allowing us to focus on the Test-Driven Development (TDD) process.
   b. models.py
      i. In `models.py`, we'll define the `Sheep` class.
         1. This class will represent each individual sheep, with attributes such as `id`, `name`, `age`, `breed`, and any other details we want to store about each sheep.
      ii. This separation of `db.py` and `models.py` will help us keep our database logic isolated from our data models
         1. promoting modularity and maintainability.
2. Now, in our `models.py` file, we will define the Sheep class in Pydantic by using the following code.
   a.

```python
from pydantic import BaseModel


class Sheep(BaseModel):
    id: int
    name: str
    breed: str
    sex: str
```

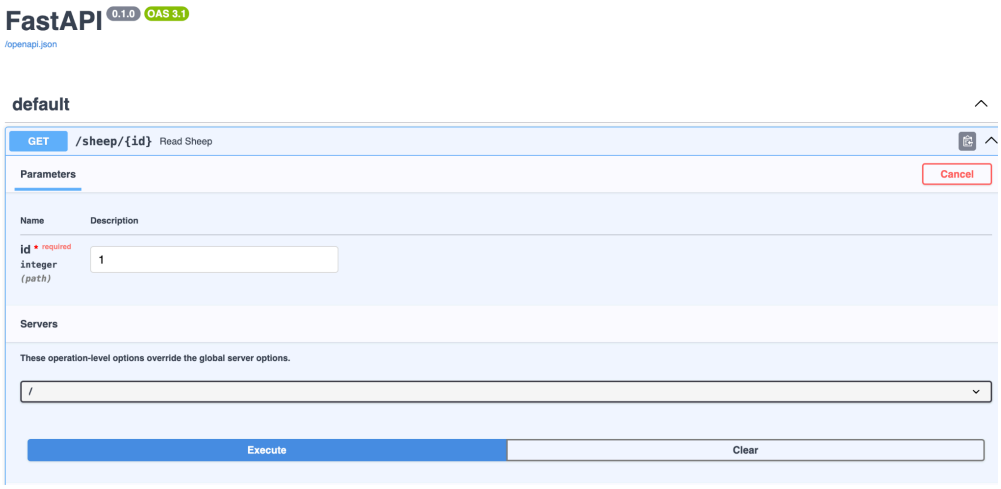
i. This way, we can promote modularization. The models.py file will manage the data structures and provide validation functionality, while the db.py file will just manage data storage and retrieval.

3. Now, in db.py, we will add the code to create our in-memory database.

```python
from models.models import Sheep
from typing import Dict


1 usage
class FakeDB:
    def __init__(self):
        self.data: Dict[int, Sheep] = {}

    def get_sheep(self, id: int) -> Sheep:
        return self.data.get(id)


db = FakeDB()
db.data = {
    1: Sheep(id=1, name="Spice", breed="Gotland", sex="ewe"),
    2: Sheep(id=2, name="Blondie", breed="Polypay", sex="ram"),
    3: Sheep(id=3, name="Deedee", breed="Jacobs Four Horns", sex="ram"),
    4: Sheep(id=4, name="Rommy", breed="Romney", sex="ewe"),
    5: Sheep(id=5, name="Vala", breed="Valais Blacknose", sex="ewe"),
    6: Sheep(id=6, name="Esther", breed="Border Leicester", sex="ewe")
}
```

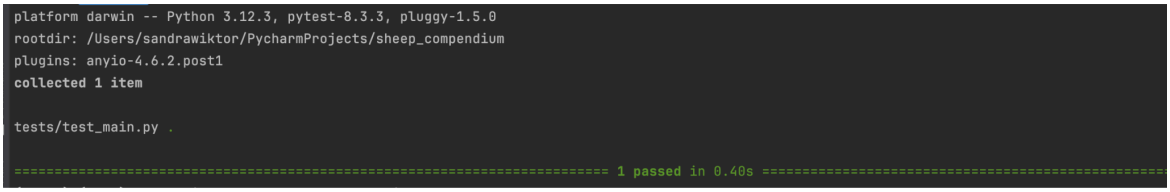
a. This setup creates a small mock database of sheep that can be accessed by their IDs using `db.get_sheep(id)`. The `FakeDB` class and `data` dictionary provide an easy way to retrieve information about each sheep by their unique identifier, simulating basic database operations for testing or prototyping purposes.

Esther the Border Leicester is pleased you got this far.

4. Now, update `main.py` to add the route for retrieving a sheep by ID with a GET request.
    a.
```python
from fastapi import FastAPI
from models.db import db
from models.models import Sheep

app = FastAPI()

@app.get( path: "/sheep/{id}", response_model=Sheep)
def read_sheep(id: int):
    return db.get_sheep(id)
```
5. Now, run the following command in the terminal:
    a.
```
uvicorn main:app --reload
```
6. Go to the link: http://127.0.0.1:8000/docs
7. Send a request to the GET endpoint to access the sheep with ID=1 using the 'Try it Now' feature in the Swagger UI interface.
    a.


8. It should return the following:
    a.


9. Now, go back to your terminal, and write pytest again.

a. 

b. Your test should pass!

Submission

1. [10pts] Go to test_main.py. Add a test case that checks if we can update sheep data. To complete this activity, fill in the TODOs.

```
# Define a test function for adding a new sheep
def test_add_sheep():
    # TODO: Prepare the new sheep data in a dictionary format.


    # TODO: Send a POST request to the endpoint "/sheep" with the new sheep data.
    #  Arguments should be your endpoint and new sheep data.

    # TODO: Assert that the response status code is 201 (Created)

    # TODO: Assert that the response JSON matches the new sheep data

    # TODO: Verify that the sheep was actually added to the database by retrieving the new sheep by ID.
    #  include an assert statement to see if the new sheep data can be retrieved.
```

   a. **Note: when you send new sheep data to the POST request, use the keyword "json=*sheep_data_name*"

Let's add a new sheep to test. Please choose one of the following sheep to add information about. The first sheep is a suffolk, the second sheep is an F1, and the third is a babydoll. Feel free to use your own sheep pictures instead.

Add a screenshot of your completed code below.

```
.py   tests\__init__.py   main.py   test_main.py   models.py   db.py

    }

#Define a test function for adding a new sheep
def test_add_sheep():  new *
    #TODO: Prepare the new sheep data in a dictionary format
    new_sheep_data = {
        "name": "Chowder",
        "breed": "Suffolk",
        "sex": "ewe"
    }
    #TODO: Send a POST request to the endpoint "/sheep" with the new sheep data.
    #Arguments should be your endpoint and new sheep data
    response = client.post( url: "/sheep", json=new_sheep_data)
    #TODO: Assert that the response status code is 201 (Created)
    assert response.status_code == 201
    #TODO: Assert that the response JSON matches the new sheep data
    assert response.json() == {
        "id": 2,
        "name": "Chowder",
        "breed": "Suffolk",
        "sex": "ewe"
    }
    #TODO: Verify that the sheep was actually added to the database by retrieving the
    # include an assert statement to see if the new sheep data can be retrieved.
    response = client.get("/sheep/2")
    assert response.status_code == 200
    assert response.json() == {

48:81   CRLF   UTF-8   4 spaces   Python 3.13 (sheep_compendium)
```

```
.py   tests\__init__.py   main.py   test_main.py ×   models.py   db.py

28  def test_add_sheep():  new *
32          "breed": "Suffolk",
33          "sex": "ewe"
34      }
35      #TODO: Send a POST request to the endpoint "/sheep" with the new sheep data.
36      #Arguments should be your endpoint and new sheep data
37      response = client.post( url: "/sheep", json=new_sheep_data)
38      #TODO: Assert that the response status code is 201 (Created)
39      assert response.status_code == 201
40      #TODO: Assert that the response JSON matches the new sheep data
41      assert response.json() == {
42          "id": 2,
43          "name": "Chowder",
44          "breed": "Suffolk",
45          "sex": "ewe"
46      }
47      #TODO: Verify that the sheep was actually added to the database by retrieving the
48      # include an assert statement to see if the new sheep data can be retrieved.
49      response = client.get("/sheep/2")
50      assert response.status_code == 200
51      assert response.json() == {
52          "id": 2,
53          "name": "Chowder",
54          "breed": "Suffolk",
55          "sex": "ewe"
56      }
```

2. Once you finish writing your test case, write Pytest to verify that we set it up correctly. It should fail with the following message.

```
tests/test_main.py:42: AssertionError
=========================== short test summary info ============================
FAILED tests/test_main.py::test_add_sheep - assert 404 == 201
============================== 1 failed, 1 passed in 0.34s ==============================
```

3. Now, we will work on creating our endpoint to add sheep. First, we modify db.py. Please add the following code:

a.

```python
def add_sheep(self, sheep: Sheep) -> Sheep:
    # Check if the sheep ID already exists
    if sheep.id in self.data:
        raise ValueError("Sheep with this ID already exists")
    # Add the new sheep to the database
    self.data[sheep.id] = sheep
    return sheep
```

i. **Note:** Notice how the parameters of add_sheep are different from the update_sheep. This parameter expects an instance of the `Sheep` class, which contains various attributes (such as `id`, `name`, `breed`, and `sex`).

4. Now, in main.py, we add the endpoint to add sheep.

a.

```python
@app.post( path: "/sheep/", response_model=Sheep, status_code=status.HTTP_201_CREATED)  # S
def add_sheep(sheep: Sheep):
    # Check if the sheep ID already exists to avoid duplicates
    if sheep.id in db.data:
        raise HTTPException(status_code=400, detail="Sheep with this ID already exists")

    # Add the new sheep to the database
    db.data[sheep.id] = sheep
    return sheep  # Return the newly added sheep data
```

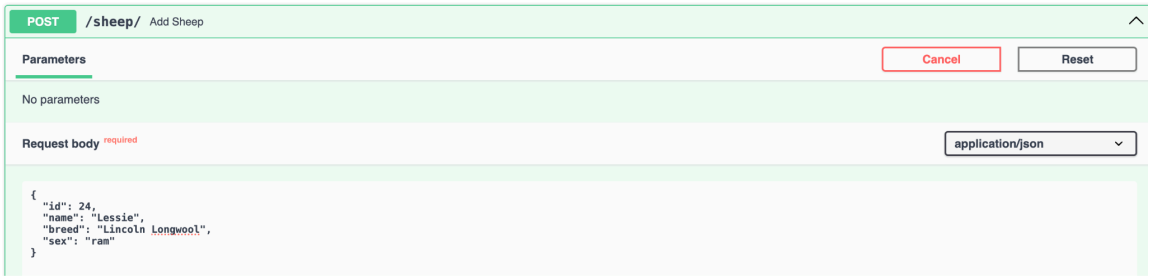
b.

```python
from fastapi import FastAPI, HTTPException, status
from models.db import db
from models.models import Sheep

app = FastAPI()
```

i. Add the *status* module to the imports.

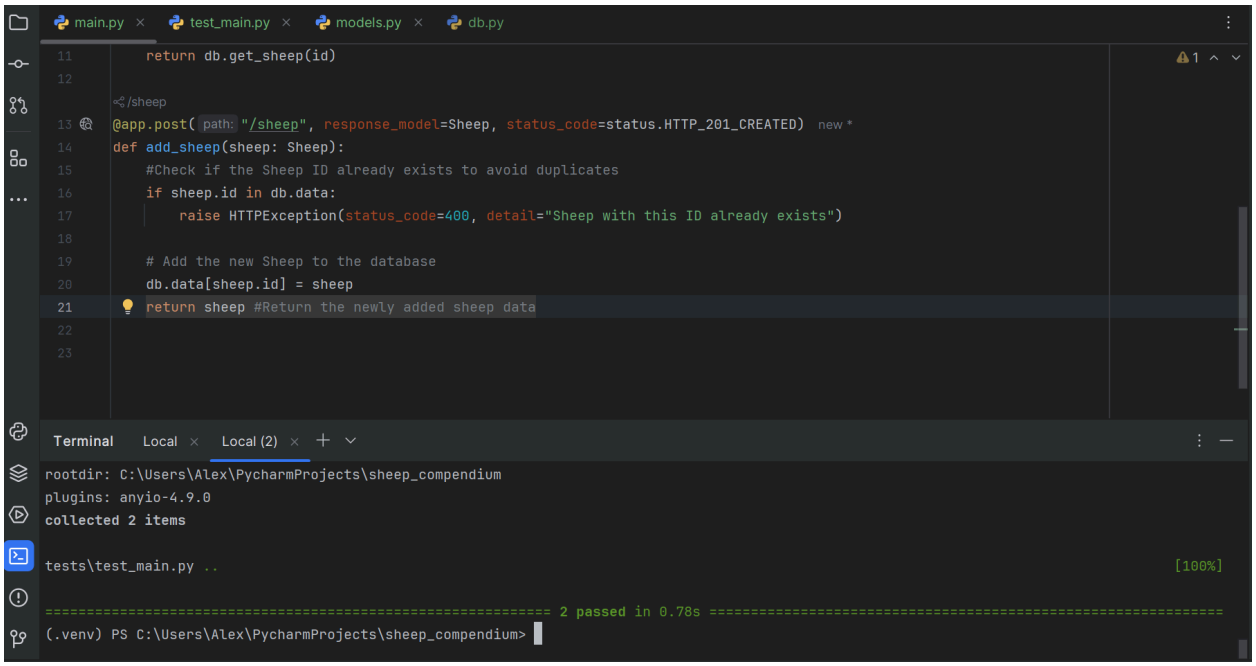
5. Send a request to the POST endpoint that adds a sheep.

a.



6. [5pts] Now, in the terminal, run the *pytest* command again. Take a screenshot of your passed test.

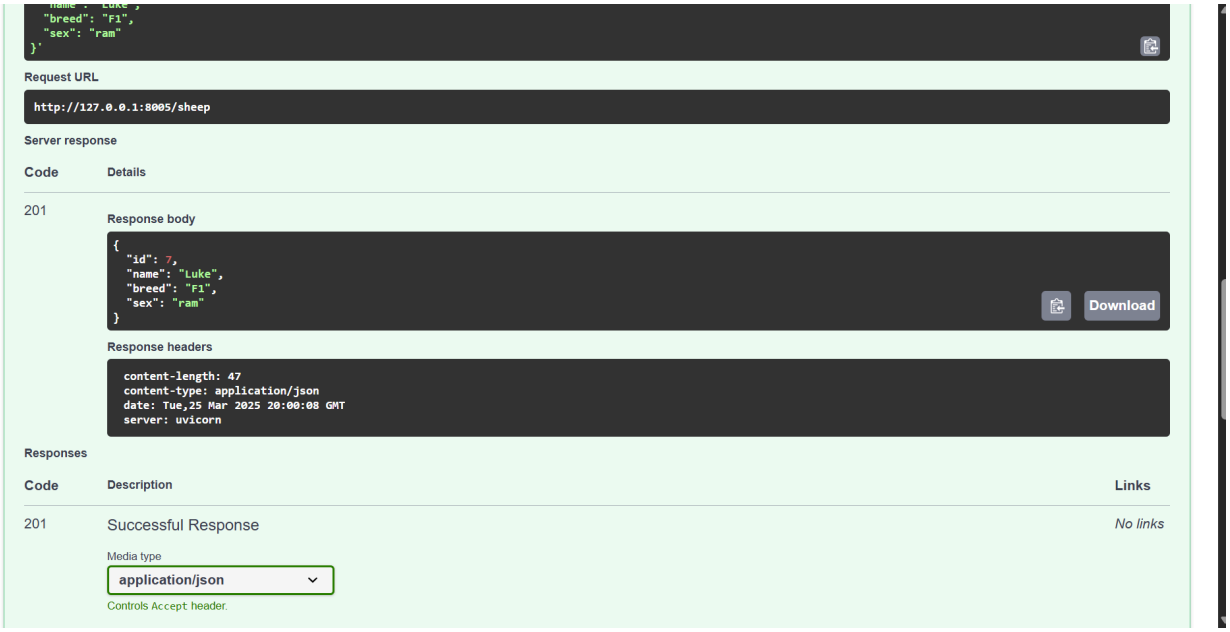
Add screenshot here.



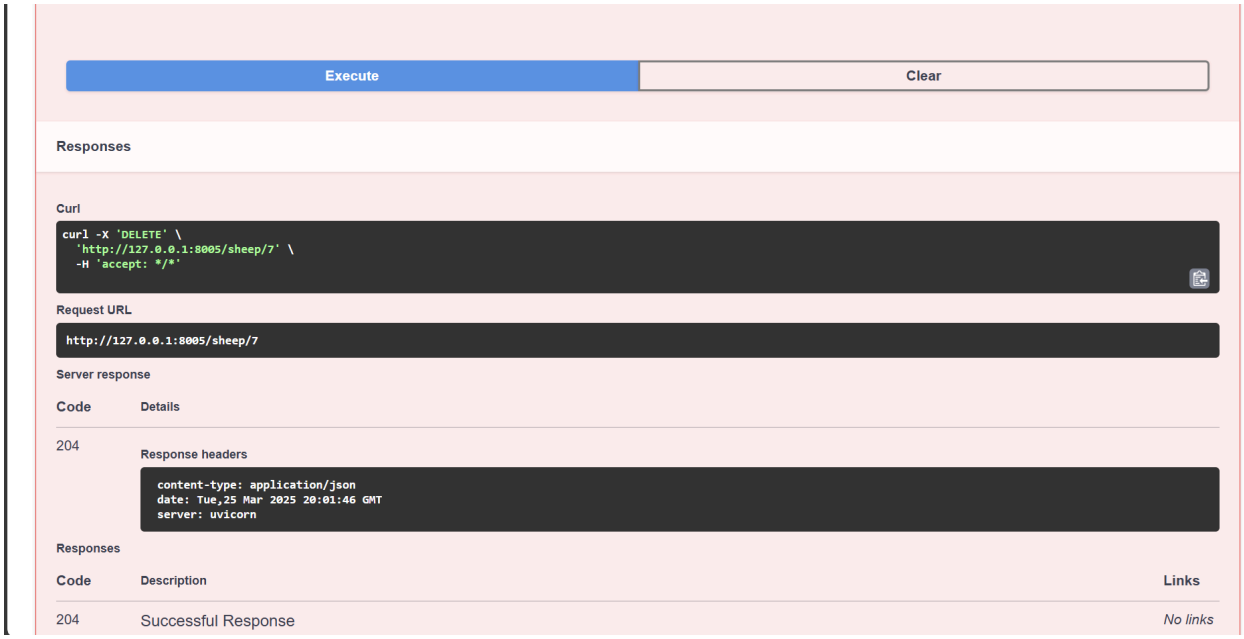
Extra Credit

- Complete the following endpoints:
    - Delete Sheep (+2pts)
        - Write working test case (+3pts)

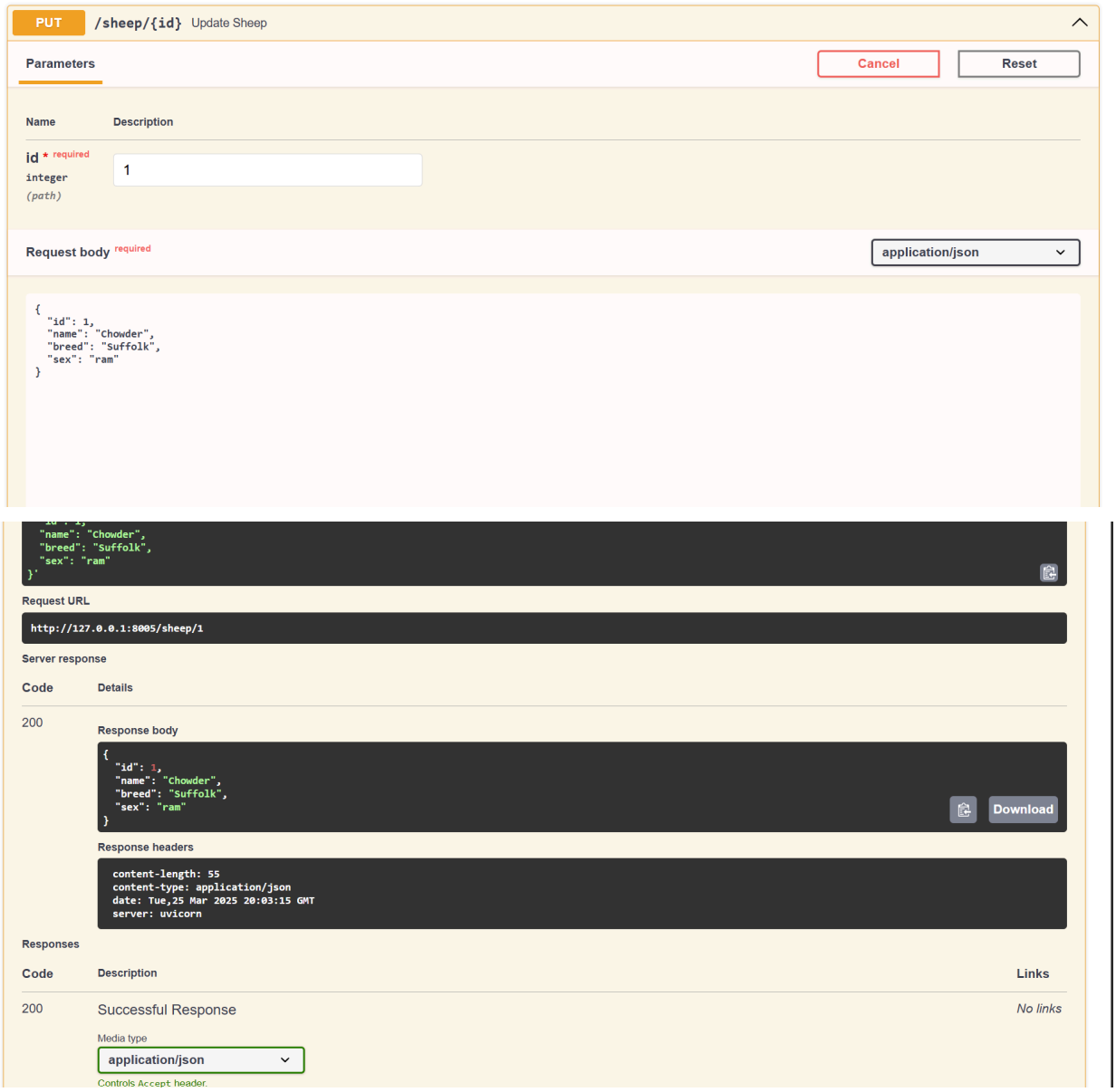
Added into the list first



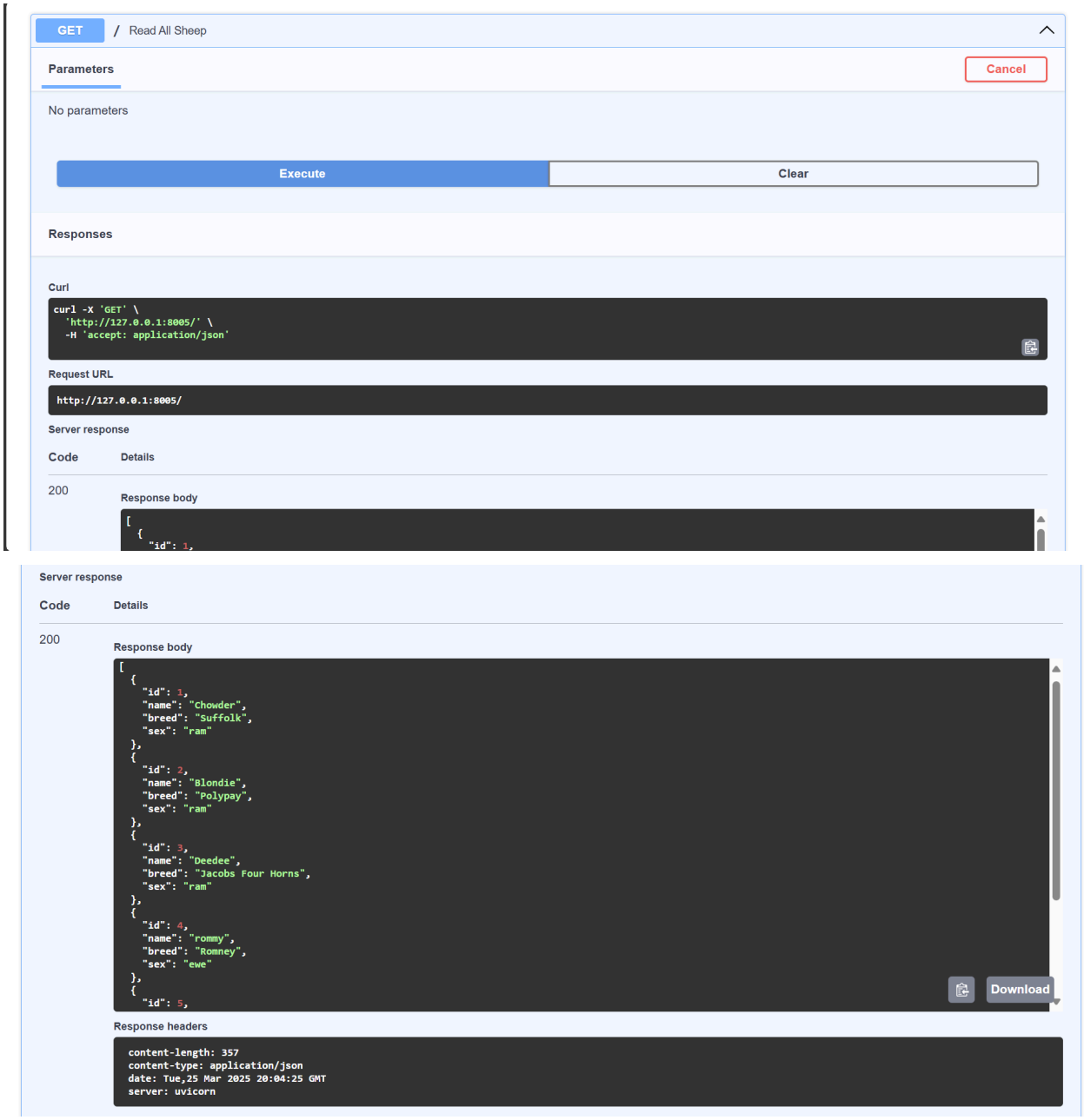
Then deleted it



- Update Sheep (+2pts)
  - Writing working test case (+3pts)



- Read all Sheep (+3pts)
  - Write working test case (+3pts)



- Submission:
  - GitHub repository link with all your code (+2)
  - Document with screenshots, demonstrations, and explanations of your code. (+5pts)

The Lincoln Longwool applauds your good work.