Lab: Develop a RESTful Service Using Test-Driven Development

Estimated time needed: 90 minutes

Welcome to the **Develop a RESTful Service Using Test-Driven Development** hands-on lab. In this lab, you will begin to build the service that you will eventually deploy to OpenShift. You will follow the plan that you created in the **Agile Planning** lab, and use good test-driven development techniques to drive the design. You will also use the coverage tool to ensure you get at least **95%** test coverage.

Objectives

In this lab, you will follow the plan from your Kanban board, write test cases for the code you "wish you had," create several REST API endpoints to make those test cases pass, perform unit testing with Nose and Coverage, achieve 95% code coverage, and conduct a sprint review to demonstrate that your REST service works.

Note: Important security information

Welcome to the Cloud IDE with OpenShift. This is where all your development will take place. It has all the tools you will need to use Docker for deploying a PostgreSQL database.

It is important to understand that the lab environment is **ephemeral**. It only lives for a short time before it is destroyed. It is imperative that you push all changes made to your own GitHub repository so that it can be recreated 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 environment for any purpose.

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 be 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.

Note on screenshots

Throughout this lab, you will be prompted to take screenshots and save them on your device. You will need these screenshots either to answer graded quiz questions or will be required for submission under **Option 1: AI-Graded Submission and Evaluation** or **Option 2: Peer-Graded Submission and Evaluation**. Your screenshot must have either the .jpeg or .png extension.

To take screenshots, you can use various free screen-capture tools or your operating system's shortcut keys. For example:

- **Mac:** you can use `Shift + Command + 3` (⇧ + ⌘ + 3) on your keyboard to capture your entire screen, or `Shift + Command + 4` (⇧ + ⌘ + 4) to capture a window or area. They will be saved as a file on your desktop.

- **Windows:** You can capture your active window by pressing `Alt + Print Screen` on your keyboard. This command copies an image of your active window to the clipboard. Next, open an image editor, paste the image from your clipboard to the image editor, and save the image.

Initialize development environment

Because the Cloud IDE in the OpenShift environment is ephemeral, it 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's 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}/devops-capstone-project.git
cd devops-capstone-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 is not already open.

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 for the `{your_github_account}` placeholder 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/devops-capstone-project.git
cd devops-capstone-project
bash ./bin/setup.sh
```

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

4. Finally, use the `exit` command to close the current terminal. The environment will not 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:

   2025-12-10 15:09:14 Wednesday

   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 3.9:

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

You are now ready to start working.

Project overview

You have been asked by the customer account manager at your company to develop an Account microservice to keep track of the customers on your e-commerce website. In the **Agile Planning** lab, you created a plan to do just that. Since it is a microservice, it is expected to have a well-formed REST API that other microservices can call. This service initially needs the ability to create, read, update, delete, and list customers.

You have also been informed that someone else has started on this task and has already developed the database model and a Python Flask-based REST API with an endpoint to **create** a customer account. You should already have this code from when you created your own repo for the **Agile Planning** lab. If you have not completed that lab, please stop and do so now.

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 accounts `[{...}]`	`GET /accounts`
Create	POST	201_CREATED	An account as json `{...}`	`POST /accounts`
Read	GET	200_OK	An account as json `{...}`	`GET /accounts/{id}`
Update	PUT	200_OK	An account as json `{...}`	`PUT /accounts/{id}`
Delete	DELETE	204_NO_CONTENT	`""`	`DELETE /accounts/{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 1: Implement Your First User Story

It is now time to implement the plan that you created in the previous lab. You will start by moving the user story from the top of your `Sprint Backlog` to the `In Progress` column and assigning it to yourself. Then you will read the story to understand what is required and create a branch in the lab environment to begin working on it. You will start by writing a test case for the code you "wish you had" and then writing the code to make the test case pass. You will do this for each story in the Sprint Backlog until the backlog is empty.

If you ranked your Sprint Backlog correctly, the first story at the top should be **Set up the development environment** with a label of `technical debt`. It is time to pay this debt. In addition to the work you just did to get the lab environment ready, you should also configure your `setup.cfg` file to have your `nosetests` display in color by default when running your test suite.

Recall from the **Introduction to TDD course** that the command to show color and run coverage with `nosetests` is:

```
nosetests -vv --with-spec --spec-color --with-coverage --cover-erase --cover-package=service
```

You must place these flags in the `setup.cfg` file so that all you have to do is run `nosetests`, and all of those flags will be active.

Your task

1. Take the story titled "**Set up the development environment**" from the top of the `Sprint Backlog`, move it to the `In Progress` column, and **assign it** to *yourself*.
2. Change into the project directory `devops-capstone-project`
3. Create a new branch called `dev-setup` to begin working on the story.

   ▼ Click here for a hint.

   ```
   git checkout -b {branch name here}
   ```

▼ Click here for the answer.

```
git checkout -b dev-setup
```

4. Edit the `setup.cfg` file and add the flags in the example above, under the `[nosetests]` stanza.

▼ Click here for a hint.
Each flag will be a `name=value` pair, where the name is the name of the flag without the `--`. If the flag has no value, use `=1`. For example: `--verbosity 2` would be `verbosity=2` and `--with-spec` would be `with-spec=1`.
▼ Click here to check your answer.
The complete [nosetests] stanza in `setup.cfg` should look like this:

```
[nosetests]
verbosity=2
with-spec=1
spec-color=1
with-coverage=1
cover-erase=1
cover-package=service
```

5. Run `nosetests` just to be sure that the file is being read correctly and does not cause an error. If it does, fix it.

6. Use the `git commit -am` command to commit your changes with the message `"added nose arguments"`, and the `git push` command to push those changes to your repository.

   Note: You will be prompted to set up your git user and email the first time you push:

```
git config --local user.name "{your GitHub name here}"
git config --local user.email {your GitHub email here}
```

▼ Click here for a hint.

```
git commit -am "{message here}"
git push --set-upstream origin {branch name here}
```

▼ Click here for the answer.

```
git commit -am "added nose arguments"
git push --set-upstream origin dev-setup
```

7. Create a pull request on GitHub to merge your changes into the `main` branch, and since there is no one else on your team, accept the pull request, merge it, and delete the branch.

8. Finally, go to your kanban board and move your story into the `Done` column to show it has been completed.

Now that you have established your `setup.cfg` file, you can get better test output from Nose without including the parameters in your command.

Evidence

For Option 1: AI-Graded Submission and Evaluation

1. Open the **public GitHub repository URL** of the `setup.cfg` file, which contains the configuration for **nosetests**, **coverage report**, **Flake8**, and **Pylint**. Copy and save the **public URL** you will submit this URL as evidence.

2. Open your **Kanban board** and take a screenshot showing the story **"Setting up the development environment"** moved to the **Done** column.
   Save the screenshot as **`rest-techdebt-done.jpeg`** (or **`rest-techdebt-done.png`**).

For Option 2: Peer-Graded Submission and Evaluation

1. Open your `setup.cfg` file in the Cloud IDE and take a screenshot showing the contents of the file.
   Save it as **`rest-setupcfg-done.jpeg`** or **`rest-setupcfg-done.png`**.

2. Open your **Kanban board** and take a screenshot showing the story **"Setting up the development environment"** in the **Done** column.
   Save it as **`rest-techdebt-done.jpeg`** or **`rest-techdebt-done.png`**.

Congratulations! You have just completed your first story.

Reference: RESTful Service

Here are some hints on the RESTful behavior of each of your stories. Use these to carry out the testing and implementation in the following exercises.

List

- List should use the `Account.all()` method to return all of the accounts as a `list` of `dict` and return the `HTTP_200_OK` return code.
- It should never send back a `404_NOT_FOUND`. If you do not find any accounts, send back an empty list (`[]`) and `200_OK`.

Read

- Read should accept an `account_id` and use `Account.find()` to find the account.
- It should return an `HTTP_404_NOT_FOUND` if the account cannot be found.
- If the account is found, it should call the `serialize()` method on the account instance and return a Python dictionary with a return code of `HTTP_200_OK`.

Update

- Update should accept an `account_id` and use `Account.find()` to find the account.
- It should return an `HTTP_404_NOT_FOUND` if the account cannot be found.
- If the account is found, it should call the `deserialize()` method on the account instance passing in `request.get_json()` and call the `update()` method to update the account in the database.
- It should call the `serialize()` method on the account instance and return a Python dictionary with a return code of `HTTP_200_OK`.

Delete

- Delete should accept an `account_id` and use `Account.find()` to find the account.
- If the account is not found, it should do nothing.
- If the account is found, it should call the `delete()` method on the account instance to delete it from the database.
- It should return an empty body `""` with a return code of `HTTP_204_NO_CONTENT`.

Use these hints to write your test cases first, and then write the code to make the test cases pass.

Exercise 2: Create a REST API with Flask

It is now time to implement the rest of the stories. Since you may have ranked them in a different order than is outlined here, you might implement them in the order that you ranked them. However, since both Update and Delete require that you can Read an account, it is strongly recommended that Read is ranked higher than Update or Delete. The List story is independent of any other and can be implemented anytime after Create, which is already implemented.

If you are unfamiliar with Flask, note that all the routes for the accounts service are the same; only the method changes. The function and route to create an account are already provided in the sample code, which you can use while running the tests.

The structure of the existing code is covered in the `README.md` file in your repository. It is recommended that you refer to this so that you know where things are, but you will mostly be working with `tests/test_routes.py` and `service/routes.py`.

Your task

It is time to implement the next four stories. Here is the general workflow:

1. Get the next highest-ranked story to work on.
2. Create a branch to work in.
3. Implement a test case that asserts the correct behavior.
4. Implement the code to make the test case pass.
5. Maintain code coverage of 95% or better.
6. Make a pull request to merge your changes.
7. Update the kanban board by moving your story to **Done**.
8. Take a screenshot to document your progress.

Here is a more detailed breakdown using "Read an Account" as the story. Be sure to check the next page for hints on the requirements for each of these REST APIs.

Task 1: Select the Next Story to Work On

1. Go to your kanban board, take the next story from the top of the `Sprint Backlog`, and move it to the `In Progress` column.

2. Open the story and assign it to **yourself**.

3. Read the story to understand what you need to implement.

Task 2: Create a Branch

1. Since you are working in branches, you must pull the latest changes from the `main` branch to stay up-to-date as you merge each story.

   The steps are:

   ```
   git checkout main
   git pull
   git branch -d {old_branch_name}
   git checkout -b {new_branch_name}
   ```

   This will switch to the main branch, pull the latest changes, delete your old branch, and create a new branch.

Task 3: Write a Test Case and watch it fail

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 outlined in the acceptance criteria of the story.

For example, if the story is **Read an account from the service**, then you might do the following:

Open **test_routes.py** in IDE

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

`devops-capstone-project/tests/test_routes.py`

1. Create a test case called `test_read_an_account(self)`.
2. Make a `self.client.post()` call to accounts to create a new account, passing in some account data.
3. Get back the account id that was generated from the `json`.
4. Make a `self.client.get()` call to `/accounts/{id}` passing in that account id.
5. Assert that the return code was `HTTP_200_OK`.
6. Check the `json` that was returned and `assert` that it is equal to the data that you sent.
7. Run `nosetests` and watch it fail because there is no code yet.

▼ Click here for a hint.
Here is starter code to test read an account:

```
def test_get_account(self):
    """It should Read a single Account"""
    account = self._create_accounts(1)[0]
    # make a call to self.client.post() to create the account
    # assert that the resp.status_code is status.HTTP_200_OK
    # get the data from resp.get_json()
    # assert that data["name"] equals the account.name
```

▼ Click here to check your solution.
This is a complete test case for read an account:

```
def test_get_account(self):
    """It should Read a single Account"""
    account = self._create_accounts(1)[0]
    resp = self.client.get(
        f"{BASE_URL}/{account.id}", content_type="application/json"
    )
    self.assertEqual(resp.status_code, status.HTTP_200_OK)
    data = resp.get_json()
    self.assertEqual(data["name"], account.name)
```

Task 4: Write the code to make the test case pass

Once you have a test case, you can begin to write the code to make it pass. Assuming that you are working on the "read an account" story, you might do the following:

Open **routes.py** in IDE

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

`devops-capstone-project/service/routes.py`

1. Create a Flask route that responds to the `GET` method for the endpoint `/accounts/<id>`.
2. Create a function called `read_account(id)` to hold the implementation.
3. Call the `Account.find()`, which will return an account by `id`.
4. Abort with a return code `HTTP_404_NOT_FOUND` if the account was not found.

5. Call the `serialize()` method on an account 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.
Here is a starter code for the REST API to read an account:

```
######################################################################
# READ AN ACCOUNT
######################################################################
@app.route("/accounts/<int:account_id>", methods=["GET"])
def get_accounts(account_id):
    """
    Reads an Account
    This endpoint will read an Account based the account_id that is requested
    """
    app.logger.info("Request to read an Account with id: %s", account_id)
    # use the Account.find() method to find the account
    # abort() with a status.HTTP_404_NOT_FOUND if it cannot be found
    # return the serialize() version of the account with a return code of status.HTTP_200_OK
    return {the account as json here + 200}
```

▼ Click here to check your solution.
This is a complete REST API implementation for reading an account:

```
######################################################################
# READ AN ACCOUNT
######################################################################
@app.route("/accounts/<int:account_id>", methods=["GET"])
def get_accounts(account_id):
    """
    Reads an Account
    This endpoint will read an Account based the account_id that is requested
    """
    app.logger.info("Request to read an Account with id: %s", account_id)
    account = Account.find(account_id)
    if not account:
        abort(status.HTTP_404_NOT_FOUND, f"Account with id [{account_id}] could not be found.")
    return account.serialize(), status.HTTP_200_OK
```

Task 5: Check your 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 an account that was not found, so you will need to write another test case that reads an account with an account 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_account_not_found(self)`.
2. Make a `self.client.get()` call to `/accounts/{id}` passing in `0` as the account id.
3. Assert that the return code was `HTTP_404_NOT_FOUND`.
4. Run `nosetests` and fix the code in `routes.py` until it passes.

▼ Click here for a hint.
Here is starter code for an account not found:

```
def test_get_account_not_found(self):
    """It should not Read an Account that is not found"""
    # send a self.client.get() request to the BASE_URL with an invalid account number (e.g., 0)
    # assert that the resp.status_code is status.HTTP_404_NOT_FOUND
```

▼ Click here to check your solution.
This is a complete test case for reading an account that was not found:

```
def test_get_account_not_found(self):
    """It should not Read an Account that is not found"""
    resp = self.client.get(f"{BASE_URL}/0")
    self.assertEqual(resp.status_code, status.HTTP_404_NOT_FOUND)
```

Task 6: Make a pull request

Now, you are ready to push the code back to GitHub and make a pull request.

1. Run `nosetests` to make sure that all of the tests pass. If they are not, fix them.
2. Use `git commit -am "{commit message here}"` to commit your changes.
3. Use `git push --set-upstream origin {your branch name}` to push your changes to GitHub.
4. Create a pull request on GitHub to merge your changes into the `main` branch.
5. Since there is no one else on your team, accept the pull request, merge it, and delete the branch.

Task 7: Update the Kanban board

Your final task is to update the kanban board to let everyone else on the team know that you are done.

1. Move your story into the `Done` column to show it has been completed.

Evidence

For both **Option 1: AI-Graded Submission and Evaluation** and **Option 2: Peer-Graded Submission and Evaluation**.
Take **screenshots** of your **Kanban board** after each story is moved to the **Done** column. Name the screenshots as follows:

- `read-accounts.jpeg` (or `read-accounts.png`) — for the story **"Read an account from the service."**
- `list-accounts.jpeg` (or `list-accounts.png`) — for the story **"List all accounts in the service."**
- `update-accounts.jpeg` (or `update-accounts.png`) — for the story **"Update an account in the service."**
- `delete-accounts.jpeg` (or `delete-accounts.png`) — for the story **"Delete an account from the service."**

Wash, Rinse, Repeat

Now, go back to **Task 1** and work on the next story until all of the stories are implemented. On the following page, you will find the remaining hints and solutions. You may refer to them while writing your code.

Hints and Solutions

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

List

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

▼ Click here for a hint.
Here is starter code to test the list for all accounts:

```
    def test_get_account_list(self):
        """It should Get a list of Accounts"""
        self._create_accounts(5)
        # send a self.client.get() request to the BASE_URL
        # assert that the resp.status_code is status.HTTP_200_OK
        # get the data from resp.get_json()
        # assert that the len() of the data is 5 (the number of accounts you created)
```

▼ Click here to check your solution.
This is a complete test case for list all accounts:

```
    def test_get_account_list(self):
        """It should Get a list of Accounts"""
        self._create_accounts(5)
        resp = self.client.get(BASE_URL)
        self.assertEqual(resp.status_code, status.HTTP_200_OK)
        data = resp.get_json()
        self.assertEqual(len(data), 5)
```

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

▼ Click here for a hint.
Here is starter code for the REST API for list all accounts:

```
######################################################################
# LIST ALL ACCOUNTS
######################################################################
@app.route("/accounts", methods=["GET"])
def list_accounts():
    """
    List all Accounts
    This endpoint will list all Accounts
    """
```

```
    app.logger.info("Request to list Accounts")
    # use the Account.all() method to retrieve all accounts
    # create a list of serialize() accounts
    # log the number of accounts being returned in the list
    # return the list with a return code of status.HTTP_200_OK
    return {list of accounts as json here + 200}
```

▼ Click here to check your solution.
This is a complete REST API implementation for list all accounts:

```
######################################################################
# LIST ALL ACCOUNTS
######################################################################
@app.route("/accounts", methods=["GET"])
def list_accounts():
    """
    List all Accounts
    This endpoint will list all Accounts
    """
    app.logger.info("Request to list Accounts")
    accounts = Account.all()
    account_list = [account.serialize() for account in accounts]
    app.logger.info("Returning [%s] accounts", len(account_list))
    return jsonify(account_list), status.HTTP_200_OK
```

Update

Write a test for the **Update** function:

▼ Click here for a hint.
Here is starter code to test update an account:

```
def test_update_account(self):
    """It should Update an existing Account"""
    # create an Account to update
    test_account = AccountFactory()
    # send a self.client.post() request to the BASE_URL with a json payload of test_account.serialize()
    # assert that the resp.status_code is status.HTTP_201_CREATED
    # update the account
    # get the data from resp.get_json() as new_account
    # change new_account["name"] to something known
    # send a self.client.put() request to the BASE_URL with a json payload of new_account
    # assert that the resp.status_code is status.HTTP_200_OK
    # get the data from resp.get_json() as updated_account
    # assert that the updated_account["name"] is whatever you changed it to
```

▼ Click here to check your solution.
This is a complete test case for update an account:

```
def test_update_account(self):
    """It should Update an existing Account"""
    # create an Account to update
    test_account = AccountFactory()
    resp = self.client.post(BASE_URL, json=test_account.serialize())
    self.assertEqual(resp.status_code, status.HTTP_201_CREATED)
    # update the account
    new_account = resp.get_json()
    new_account["name"] = "Something Known"
    resp = self.client.put(f"{BASE_URL}/{new_account['id']}", json=new_account)
    self.assertEqual(resp.status_code, status.HTTP_200_OK)
    updated_account = resp.get_json()
    self.assertEqual(updated_account["name"], "Something Known")
```

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

▼ Click here for a hint.
Here is a starter code for the REST API for update an account:

```
######################################################################
# UPDATE AN EXISTING ACCOUNT
######################################################################
@app.route("/accounts/<int:account_id>", methods=["PUT"])
def update_accounts(account_id):
    """
    Update an Account
    This endpoint will update an Account based on the posted data
    """
    app.logger.info("Request to update an Account with id: %s", account_id)
    # use the Account.find() method to retrieve the account by the account_id
    # abort() with a status.HTTP_404_NOT_FOUND if it cannot be found
    # call the deserialize() method on the account passing in request.get_json()
    # call account.update() to update the account with the new data
    # return the serialize() version of the account with a return code of status.HTTP_200_OK
    return {account as json + 200}
```

▼ Click here to check your solution.
This is a complete REST API implementation for update an account:

```
######################################################################
# UPDATE AN EXISTING ACCOUNT
######################################################################
@app.route("/accounts/<int:account_id>", methods=["PUT"])
def update_accounts(account_id):
    """
    Update an Account
    This endpoint will update an Account based on the posted data
    """
    app.logger.info("Request to update an Account with id: %s", account_id)
    account = Account.find(account_id)
    if not account:
        abort(status.HTTP_404_NOT_FOUND, f"Account with id [{account_id}] could not be found.")
    account.deserialize(request.get_json())
    account.update()
    return account.serialize(), status.HTTP_200_OK
```

Delete

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

▼ Click here for a hint.
Here is starter code to test delete an account:

```
def test_delete_account(self):
    """It should Delete an Account"""
    account = self._create_accounts(1)[0]
    # send a self.client.delete() request to the BASE_URL with an id of an account
    # assert that the resp.status_code is status.HTTP_204_NO_CONTENT
```

▼ Click here to check your solution.
This is a complete test case for delete an account:

```
def test_delete_account(self):
    """It should Delete an Account"""
    account = self._create_accounts(1)[0]
    resp = self.client.delete(f"{BASE_URL}/{account.id}")
    self.assertEqual(resp.status_code, status.HTTP_204_NO_CONTENT)
```

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

▼ Click here for a hint.
Here is starter code for the REST API for delete an account:

```
######################################################################
# DELETE AN ACCOUNT
######################################################################
```

```
@app.route("/accounts/<int:account_id>", methods=["DELETE"])
def delete_accounts(account_id):
    """
    Delete an Account
    This endpoint will delete an Account based on the account_id that is requested
    """
    app.logger.info("Request to delete an Account with id: %s", account_id)
    # use the Account.find() method to retrieve the account by the account_id
    # if found, call the delete() method on the account
    # return and empty body ("") with a return code of status.HTTP_204_NO_CONTENT
    return {empty string + 204}
```

▼ Click here to check your solution.
This is a complete REST API implementation for delete an account:

```
######################################################################
# DELETE AN ACCOUNT
######################################################################
@app.route("/accounts/<int:account_id>", methods=["DELETE"])
def delete_accounts(account_id):
    """
    Delete an Account
    This endpoint will delete an Account based on the account_id that is requested
    """
    app.logger.info("Request to delete an Account with id: %s", account_id)
    account = Account.find(account_id)
    if account:
        account.delete()
    return "", status.HTTP_204_NO_CONTENT
```

Error Handlers

It is important to also test the error handlers to make sure they are working properly. Here are some suggestions for doing this:

Method Not Allowed

To cause a method not allowed error, simply make a GET, POST, PUT, or DELETE on an endpoint that doesn't support that HTTP method.

▼ Click here for a hint.
Here is the starter code for testing a method that is not allowed:

```
def test_method_not_allowed(self):
    """It should not allow an illegal method call"""
    # call self.client.delete() on the BASE_URL
    # assert that the resp.status_code is status.HTTP_405_METHOD_NOT_ALLOWED
```

▼ Click here to check your solution.
Here is the implementation for testing a method that is not allowed:

```
def test_method_not_allowed(self):
    """It should not allow an illegal method call"""
    resp = self.client.delete(BASE_URL)
    self.assertEqual(resp.status_code, status.HTTP_405_METHOD_NOT_ALLOWED)
```

Exercise 3: Run the REST service

In this exercise, you will run the service and ensure you can access it locally.

Your Task

In the bash terminal, use the `flask db-create` command to refresh the database.

```
flask db-create
```

Use the `make run` command to start the service with the new database.

```
make run
```

Launch the application in a web browser by pressing the **Launch Account Service** button below:

Launch Account Service

Results

You should get back the following web page:

You are now ready to test your running service.

Exercise 4: Sprint Review

Now that all of the implementation is done, it is time to demo your new service at the **Sprint Review** meeting. The product owner and stakeholders are excited to see what you have implemented.

In this exercise, you will use the `curl` command to make REST calls for the Create, Read, Update, Delete, and List functions of the service that you have created. With the service running, open a second Bash terminal and enter the following curl commands to make REST calls.

Demo Create an Account

1. Enter the following command to create an account:

```
curl -i -X POST http://127.0.0.1:5000/accounts \
-H "Content-Type: application/json" \
-d '{"name":"John Doe","email":"john@doe.com","address":"123 Main St.","phone_number":"555-1212"}'
```

   Check the account id that was returned. It should have been `1` but if it was not, substitute the account id that was returned for all of the remaining calls.

Evidence

For Option 1: AI-Graded Submission and Evaluation:
Copy and paste the **cURL command** and its **JSON output** demonstrating the **CREATE** function of an account, and save it in a text file named `rest-create-done` for the final project submission and evaluation.

For Option 2: Peer-Graded Submission and Evaluation:
Take a **screenshot** of the output demonstrating the **CREATE** function of an account and save it as `rest-create-done.png` or `rest-create-done.jpeg` for the Peer Assignment.

Demo List All Accounts

1. Enter the following command to list all accounts:

```
curl -i -X GET http://127.0.0.1:5000/accounts
```

Evidence

For Option 1: AI-Graded Submission and Evaluation:
Copy and paste the **cURL command** and its **JSON output** demonstrating the **LIST** function of an account, and save it in a text file named `rest-list-done` for the final project submission and evaluation.

For Option 2: Peer-Graded Submission and Evaluation:
Take a **screenshot** of the output demonstrating the **LIST** function of an account and save it as `rest-list-done.png` or `rest-list-done.jpeg` for the Peer Assignment.

Demo Read an Account

1. Enter the following command to read the account:

```
curl -i -X GET http://127.0.0.1:5000/accounts/1
```

Evidence

For Option 1: AI-Graded Submission and Evaluation:
Copy and paste the **cURL command** and its **JSON output** demonstrating the **READ** function of an account, and save it in a text file named `rest-read-done` for the final project submission and evaluation.

For Option 2: Peer-Graded Submission and Evaluation:
Take a **screenshot** of the output demonstrating the **READ** function of an account and save it as `rest-read-done.png` or `rest-read-done.jpeg` for the Peer Assignment.

Demo Update an Account

1. Enter the following command to update the account:

```
curl -i -X PUT http://127.0.0.1:5000/accounts/1 \
-H "Content-Type: application/json" \
-d '{"name":"John Doe","email":"john@doe.com","address":"123 Main St.","phone_number":"555-1111"}'
```

Note: You are sending a new phone number. Check that the phone number returned is `555-1111`.

Evidence

For Option 1: AI-Graded Submission and Evaluation:
Copy and paste the **cURL command** and its **JSON output** demonstrating the **UPDATE** function of an account, and save it in a text file named `rest-update-done` for the final project submission and evaluation.

For Option 2: Peer-Graded Submission and Evaluation:
Take a **screenshot** of the output demonstrating the **UPDATE** function of an account and save it as `rest-update-done.png` or `rest-update-done.jpeg` for the Peer Assignment.

Demo Delete an Account

1. Enter the following command to delete the account:

```
curl -i -X DELETE http://127.0.0.1:5000/accounts/1
```

Evidence

For Option 1: AI-Graded Submission and Evaluation:
Copy and paste the **cURL command** and its **JSON output** demonstrating the **DELETE** function of an account, and save it in a text file named `rest-delete-done` for the final project submission and evaluation.

For Option 2: Peer-Graded Submission and Evaluation:
Take a **screenshot** of the output demonstrating the **DELETE** function of an account and save it as `rest-delete-done.png` or `rest-delete-done.jpeg` for the Peer Assignment.

You have completed the demo of the REST calls you implemented, and the product owner has agreed that they are all done as expected.

Move Stories to Closed/Check if moved to Done:

- If you are using Zenhub kanban, move all of your stories from the `Done` column to the `Closed` column.
    - If you are using GitHub kanban, check if all user stories are moved to the `Done` column. Please note that in GitHub Kanban, when stories are moved to `Done` column, they are automatically closed due to GitHub's built-in feature.

Conclusion

Congratulations! You have completed creating your first sprint for the capstone project. You have implemented your Account microservice, moved stories along the kanban board, made pull requests to merge your code back into the main branch, and moved your stories to the `Done` column.

Next Steps

In a real Agile team, you would conduct a sprint retrospective. Do not deny yourself this valuable ceremony.

Pause for a moment and think about:

- What went right?
- What went wrong?
- What would you change for the next sprint?

Write these reflections down somewhere so that you do not forget.
Reflecting on your performance is critical for becoming a high performer, and feedback is always welcome, even if it is just from yourself.

Author(s)

Tapas Mandal
John J. Rofrano

Other Contributor(s)