Corndel DevOps Engineering Programme

in association with Softwire

Module 3: Project Exercise

Corndel Digital.

DA7E

Module 3

Project Exercise

For this exercise, we are going to be continuing with the to-do app that we have been working on.

So far, we've focused on adding the basic functionality, but we've not thought about testing at all. We've got away with this so far - because our app is small and simple enough that we have been able to manually test all of the functionality - but would be better if we added some robust automated test coverage.

In this exercise, we will add 3 sets of tests to our project:

- Unit Tests: to check our logic and work through edge cases.
- Integration Tests: to check that our classes work together.
- System Tests: to check that user journeys through our app work, and that we integrate correctly with Trello.

Part 1: Organise the items

At the moment, we include all of our tasks in a single list of things to do. Let's try to reduce the clutter by:

- organising them by their status (e.g. have a section for 'To Do' and 'Doing')
- hiding tasks that were completed before today
    - we can have a 'show older tasks' button for when you do want to see everything
- preventing users from having too many 'doing' items at the same time

We are going to be making these changes using TDD, so that we can trust that they work and won't break going forward.

Step 1: Making a view model

So far, your app probably does something like this to pass the tasks directly to the HTML template.

```
render_template('index.html', items=items)
```

That's fine for simple cases, but it has a couple of disadvantages:

- it gets messy if there are lots of parameters to pass in
- there isn't anywhere to put logic (e.g. filtering our items, or finding a particular thing)

So we are going to collect together all of the information that our HTML template (often called a 'view') needs into a single class. This new class (often called a 'view model') is now a nice neat container that makes it really simple for the template to access all of the values it needs.

Our view model might look like:

```
class ViewModel:
    def __init__(self, items):
        self._items = items

    @property
    def items(self):
        return self._items
```

In `app.py`, we can now use this view model to render our template:

```
item_view_model = ViewModel(items)
render_template('index.html',
view_model=item_view_model)
```

Finally, we can update our template to use our new view model:

```
{% for item in view_model.items %}
    # do stuff
{% endfor %}
```

Note: *There's nothing special about our choice of names here. Our template references an object called `view_model`, so in our render method we pass in a matching keyword argument `view_model=`, but you can change this to whatever makes sense for your application.*

Submitting your work

We'll be using Pull Requests on GitHub in the same fashion as the last module to review this exercise submission.

If you experience issues, ask a tutor for help!

Try implementing this now and make sure that everything still works. At first this might not look like a improvement, but as we add functionality we will begin to see how this help us.

Step 2: Sort into ‘to do’, ‘doing’ and ‘done’

Before we start, let’s add pytest as a dependency of our project. (You can find more information about the dependency we want to install on PyPi - https://pypi.org/project/pytest/)

We can now use TDD to add functionality to our view model! Add some more properties that return:

- just the ‘to do’ items
- just the ‘doing’ items
- just the ‘done’ items

There are several ways to do this. It doesn’t matter too much what you pick, but make sure it’s well tested!

Once we’ve got that working and tested we can start using these new properties in our template! Edit your template so that the ‘To Do’, ‘Doing’ and ‘Done’ items are all shown in separate lists.

Don't forget to commit!

Make sure you use **`git commit`** to save your work regularly.

It's good practice, and lets you revert back to a previous revision if you find you've broken everything horribly!

Try to make your commit messages concise and descriptive. Using source control tools effectively is just as important as the code you submit.

Step 3: Hiding old completed tasks

The Trello API tells us when each card was last modified. For completed tasks in our app the last modification will always be marking it as done, so we can use this time to tell how long ago the task was completed.

We are now going to split up our ‘Done’ section as follows.

- If there are fewer than 5 completed tasks, just show all of them.

- Otherwise, initially only show tasks that have been completed today.
- Allow users to toggle viewing all of the other cards too.

Use TDD to add some new properties to your view model. We are going to need

- `show_all_done_items`: which will keep track of if we should show all the completed items, or just the most recent ones.
- `recent_done_items`: which will return all the tasks that have been completed today.
- `older_done_items`: which will return all of the tasks that were completed before today.

Don't forget to commit your work often, making it easier to back-track if you run into problems.

Once all this logic is complete, it should be quite simple to edit the template to add in our new feature. Let's do this now.

Note: *Take a look at the Summary and Details HTML tags if you get stuck with the display/hide logic: https://www.w3schools.com/tags/tag_summary.asp*

Part 2: Adding Integration Tests

Our unit tests are a great start, but we aren't done yet! It would be great to add some integration tests to check that all of our classes work together as we expect them too. For this set of tests, we are going to mock our interactions with the Trello API, but test all of the rest of our code to check that each of our endpoints work.

Step 1: Starting our Test App

For our integration tests, we are going to need to be able to programmatically start our application. This 'test app' should be as close as possible to the real thing, except that we are going to want to tweak some of the configuration (eg it won't need have a real `API_KEY` for Trello, because we are planning on mocking that interaction)

To do this, we are going to make 2 changes to our application:

- (If you haven't already) Move the code that constructs the application to a `create_app`function in `app.py`. This makes it easy for our test code to spin up a new application whenever it needs

```
def create_app():
    app = Flask(__name__)
    app.config.from_object('app_config.Config')

    # All the routes and setup code etc

    return app
```

- Create a new `.env.test` file to contain all of the environment variables for our integration tests. This will need all the same variables as the development `.env` file, but with a few differences
    - fake, non-private values for our API Keys etc.
    - this one *should* be included in the source control
    - the addition of a `FLASK_SKIP_DOTENV=True` variable (which will prevent flask from automatically switching back to the development config.)

Step 2: Create the Test Client

Flask provides an inbuilt 'test_client' that we can use to test our application. To use it, we need to create a new app, and then use the app to create a test client.

```
@pytest.fixture
def client():
    # Use our test integration config instead of the 'real' version
    file_path = find_dotenv('.env.test')
    load_dotenv(file_path, override=True)

    # Create the new app.
    test_app = app.create_app()

    # Use the app to create a test_client that can be used in our tests.
    with test_app.test_client() as client:
        yield client
```

Step 3: Patch the call to the Trello API

Let's start by testing our index endpoint. We can use the test client to make a call to our main page.

```
def test_index_page(mock_get_requests, client):
    response = client.get('/')
```

But… this will fail. Our app will try to make an HTTP request to Trello, but isn't correctly configured to do that. In this test we don't want to actually make a request to the Trello API (as we'd like our tests to be fast, work offline and have no external dependencies). Therefore we are going to need to 'patch' the code so that instead of making real requests to Trello, we just return a pre-defined response. We want to include as much as possible of our code in the test as possible though, so we should try to patch the smallest thing we can. In this case, it means that we probably want to patch the call to `requests.get`.

Tip: It's really important that the mocked request is realistic (i.e. that it has the same format as the real request). If you aren't certain what the response should look like, try making the request in postman and checking what data is returned.

Try this now so that we can so that our tests no longer break.

Tip: It's really important that the mocked request is realistic (i.e. that it has the same format as the real request). If you aren't certain what the response should look like, try making the request in postman and checking what data is returned.

Step 4: Add some assertions!

The last thing to do is assert that the response we get is the correct one. This is a little tricky as the response is quite a complex object, but in this case we can keep it simple.

Write some assertions that check that the response includes the text of the tasks names that our fake Trello response returned.

Stretch Goal: Extend to the other endpoints!

If you have time, try testing the other endpoints too.

Remember to think about what's most important in each case. For example often we may want to include an assertion that the correct call was made to the Trello API.

Part 3: Selenium

The final thing to check is that with our whole system working together, our user is able to perform the actions we would like them to make.

For this section we will also need to install the selenium python package.

Step 1: Create functions to create and delete a Trello board

For this test, we are going to want to make real calls to Trello, but that presents another issue. If we run this test against our normal board, then the content that is already on that board might affect our tests, and running our tests might also mess up all the tasks we have on our board. Instead, we would rather run our tests against a new board, created just for that test.

Let's start by creating 2 functions, one to create a new Trello board and one to delete it.

Tip: We can load in environment variables from our `.env` file using the `load_dotenv` function from the `dotenv` package.

Step 2: Creating the test app.

We can now create a pytest fixture that will run before and after each test. Its job is to create the new Trello board, and then build and start our app in a background thread. After the test is completed, we can then stop that background thread and delete the Trello board. It should look something like this:

```python
import os
from threading import Thread

@pytest.fixture(scope='module')
def test_app():
    # Create the new board & update the board id environment variable
    board_id = create_trello_board()
    os.environ['TRELLO_BOARD_ID'] = board_id

    # construct the new application
    application = app.create_app()

    # start the app in its own thread.
    thread = Thread(target=lambda: application.run(use_reloader=False))
    thread.daemon = True
    thread.start()
    yield app

    # Tear Down
    thread.join(1)
    delete_trello_board(board_id)
```

Step 3: Add a fixture for the selenium driver

The selenium driver is how we control the browser that is going to load our app. You will find that there are Selenium drivers for most major browsers, but the example below uses Firefox, because its quite easy to get working. Before we start, you will need to download the Gecko Driver executable and place it in the root of your project - the selenium driver just uses this under the hood.

You can then create a fixture for your driver as follows:

```python
from selenium import webdriver

@pytest.fixture(scope="module")
def driver():
    with webdriver.Firefox() as driver
        yield driver
```

Tip: Use a separate directory for your Selenium Tests to your other tests. Selenium tests are much slower, so it helps to be able to run them separately. By putting them in separate directories we can e.g. run `pytest tests` to run the unit and integration tests in the ‘tests’ directory, or `pytest tests_e2e` to run the selenium tests in the ‘tests_e2e’ directory.

Troubleshooting!

Getting the driver to work can be a bit of a pain! If it isn't working, try these things

- you'll need the browser (FireFox in this case) installed as well as the driver.
- the browser and the driver will need to be the same version
- the driver must be on the `$PATH`
- on Mac, you may get a permissions error. If you do, try the steps explained here: https://stackoverflow.com/questions/60362018/macos-catalinav-10-15-3-error-chromedriver-cannot-be-opened-because-the-de

Step 4: Write the tests!

We are now ready to write our tests! Check everything is working with a test similar to this:

```
def test_task_journey(driver, test_app):
    driver.get('http://localhost:5000/')

    assert driver.title == 'To-Do App'
```

And then extend your test to check that you can create a new item, start it, complete it and mark it as incomplete.

Stretch Goals

Code Coverage

Have you tested everything you should test? Are all of the branches covered? Let's run a code coverage report to check.

Are there are any areas that still need some work? See if you can get the line and branch coverage up to 100%.

VCR

One of the tricks with our integration test was making sure our mock response matched the real response from the Trello API. It is really important to get this right and to ensure that it remains up to date, as otherwise our integration test is meaningless.

To help with this we could choose to use a library like VCR (https://pypi.org/project/vcrpy/). The idea is to allow you to record real responses from the API, and then reuse these responses for future tests. Here's how it works:

- The first time you run a test, the actual calls to the API are made, just like they would if running the full application.
- The response for each of those requests is recorded and saved to a file.
- In future runs of the test, all HTTP requests are intercepted.
- And instead of making the 'real' request, VCR will just look up the saved response and use that.
- (You can trigger re-recording the responses just by deleting the currently stored ones)

This allows us to have really high quality test data, but retain our quick, reliable and consistent tests.

Have a go at changing your integration tests to implement this.

Cypress

Selenium is a great tool, but it can sometimes be a pain! In particular, you might find that

- it can be tricky to debug when things don't work as you expect
- it can be tricky to get Selenium to wait correctly for things that aren't quite instant

Cypress is a more modern tool that aims to solve some of the difficulties of using Selenium. It provides features such as automatically waiting for your assertions and providing a visual snapshot of your application after every step of your test - allowing you to easily walk through and debug everything that happened.

Cypress is a JavaScript library, so you won't be able to write the tests in Python. However the documentation is good and the syntax is clear, so you should be able to pick it up without too much trouble.

You can learn more about it here - https://docs.cypress.io/

Have a go at converting your Selenium tests to Cypress. How do they compare? Which do you prefer?