Corndel DevOps Engineering Programme

in association with Softwire

Module 7: Project Exercise

Corndel Digital.

DAAS

Module 7

Project Exercise Brief

In this exercise, you will set up continuous integration for your app using Travis CI. You'll set up a Travis CI job which will build your app and run your tests.

Setup

Check out your code from the previous exercise, it'll form the starting point for this exercise.

Python, Docker and Pipelines

Running your tests on Travis CI is going to involve a lot of dependencies. You'll need a Python binary (of the correct version), the standard library, poetry, third-party Python code, plus selenium and a webdriver that can run in "headless" mode (i.e. without a GUI).

That's a lot, and you shouldn't rely on a CI/CD tool to provide a complex dependency chain like this. Instead, we'll use docker to build, test and deploy our application. Travis CI won't even need to know it's Python code! This has a few advantages:

- Our Travis CI configuration will be much simpler
- It's easier to move to a different CI/CD tool in future
- We have total control over the build and test environment via our Dockerfile

If you completed the stretch goal in your Module 5 exercise (creating a "test" docker image that can run your unit, integration and end-to-end tests), then you've already got everything you need and can move on to Part 1.

If not, follow the instructions in **Appendix A** (Testing in Docker) to get a docker test image configured.

Part 1: Set up Travis CI for your repository

Travis CI is set up to work well with GitHub but for it to work you need to enable it for any repository you want to use it for. Instructions are here: https://docs.travis-ci.com/user/tutorial/#to-get-started-with-travis-ci-using-github. Follow the first three steps for signing up with GitHub, making sure you activate Travis CI for your project exercise repository.

Once you've done that Travis CI should be enabled for your repository. This means that if you push a `.travis.yml` file to your repository Travis CI will attempt to run the build defined in that file.

Travis CI is free for public repositories. For private repositories you can run 100 builds for free, after that you need to pay. However 100 builds should be enough to complete this exercise.

Part 2: Make Travis CI build your code

Step 1: Add a basic config file

We now want to add a `.travis.yml` file which defines what the Travis job should do.

The main thing you need to define in your `.travis.yml` file is the `script` section which specifies which commands should be run for the build.

This would be an example of a very simple `.travis.yml` file:

```
script:
- echo "hello world"
```

All this will do is print "hello world". Try adding this to your repository, push the code and check that the build runs. To find the output of your build go to https://travis-ci.com/github/GitHubUsername/RepositoryName (replace `GitHubUsername` with your GitHub username and replace `RepositoryName` with the name of your repository).

Step 2: Improve the config file

We now want to add to the config file so that it actually runs the docker build instead of just printing "hello world". To do this replace the command in the `script` section with one which will run the docker build with the dockerfile you added earlier on this exercise (or as part of the stretch goals in exercise 5).

Part 3: Make Travis test the code

In exercise 3 you should have added some unit tests, integration and end-to-end tests to your project. We now want to update the Travis config so that it will run those tests.

Step 1: Make Travis run the unit tests

Add another command to the `script` section in `.travis.yml` which will run the unit tests. You should run the tests with `docker`. (Don't try to install Python and all your project dependencies in Travis - you should rely on your docker container to encapsulate and use the dependencies.)

Step 2: Make Travis run the integration tests

Expand the `script` section to also run the integration tests. This will be a bit more complicated than the unit tests as you will need to set values for the Trello environment variables to be able to run them, using the `.env.test` file.

Step 3: Make Travis run the E2E tests

Expand the `script` section once again, this time adding in your end-to-end tests. Unlike your integration tests, these will probably need real live Trello credentials. Make sure you handle the secrets properly! Normally you would set these in the `.env` file, however for the Travis job it will be easier to set them as environment variables. So that you don't commit those credentials in git you should encrypt the variables before you set them in the `.travis.yml` file. Travis makes this quite easy, see https://docs.travis-ci.com/user/environment-variables/#defining-encrypted-variables-in-travisyml.

Part 4: Update the build settings

Step 1: Update the build frequency

By default the Travis job will run every time something is pushed to a branch. It will also build every time a pull request is opened or updated. In that case it will build not what's on the branch of the pull request, but instead build the result of merging the branch into the target branch. This is very useful as it will stop you from merging something which causes the build to fail on your target branch.

In general building both branches and pull requests is useful, as it tells you both if something is wrong with the branch and if something would be wrong with the target branch once a pull request has been merged. However disabling builds for branches can be an option if builds are being too slow. Try changing the settings for your Travis job so that it will only run for pull requests.

Step 2: Enable auto cancelling builds

By default Travis will run a build each time commits are added. This is usually what you want, however this can in some cases mean you're doing more builds than is necessary. For example if you push a commit with some changes, then push a second commit before the build for the first commit has started. In this case Travis will run the build for the first commit first, then the build for the second. Running the build for intermediate commits could have small benefits such as narrowing down the cause of a build failure. Let's say that reducing the time taken matters more for this pipeline.

Change the settings for your Travis job so that it will cancel waiting builds if a new commit has been pushed to that branch/pull request.

Stretch Goals & Hints

(Stretch) Make Travis send slack notifications

Update your config file so that it will send slack notifications with the build status.

(Stretch) Make Travis email on failure

If the build fails you want to be sure someone knows about it. Therefore update your job so that it will email you if it fails.

Hints

Docker

To use `docker` in the `.travis.yml` file see https://docs.travis-ci.com/user/docker/. Most importantly you need to add:

```
services:
- docker
```

to your `.travis.yml` file. Once you've done that you can use any docker command in the `script` section.

Build settings

To change the build settings go to https://travis-ci.com/github/GitHubUsername/RepositoryName/settings. This will have toggles for the build settings, such as whether to build for branches and/or pull requests, and whether to auto cancel builds.

Appendix A

Python Testing in Docker

For your CI pipeline, you need a docker image that can run `pytest`. i.e. you should be able to run test suites using `docker run`:

```
# Run tests
$ docker run <options> my-test-image tests
=========================== test session starts ===========================
platform linux -- Python 3.8.5, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
rootdir: /app
collected 25 items

my/test/directory/test_stuff.py                                       [100%]

=========================== 11 passed in 0.49s ============================
```

This document guides you through setting up dockerised testing. You'll mostly be expanding the Dockerfile you wrote as part of Module 5, but you may also need to adjust your Python test code to work smoothly in docker.

You may have done this setup as part of a Module 5 stretch goal. There's no need to repeat this work if your tests already run in Docker, however you may find it interesting to skim through and compare approaches.

Part 1: Add a new build stage

You already have a multi-step docker build with development and production stages. Now add a third stage for testing. Call it "test". In the end you'll have an outline that looks like the one below:

```
# Common setup steps
FROM python:3.8-slim-buster as base
...

# production build stage
FROM base as production
...

# local development stage
FROM base as development
...

# testing stage
FROM base as test
...

ENTRYPOINT ["poetry", "run", "pytest"]
```

You'll notice we've filled in the ENTRYPOINT instruction to launch pytest. You can do the same.

Part 2: Get Pytest Working

Your tests probably won't complete yet, but it's worth trying. Use your dockerfile as shown below. Note how we're passing an argument to pytest directly from the `docker run` command:

```
# Build it
$ docker build --target test --tag my-test-image .

# Run tests in the "tests" directory
$ docker run my-test-image tests

# Maybe your e2e tests are somewhere else? Run them:
$ docker run my-test-image <path_to_other_tests>
```

Depending on your existing Dockerfile, your image might fail at different stages. Look carefully at any error messages, how far does it get?

- Can docker run poetry?
- Can poetry launch pytest?
- Can pytest find your tests?
- Can your tests find your application code?

If you've got this far, congratulations! If not, take some time to get work through the steps above and get the test image to a stage where it launches pytest and finds your application code. Don't worry if the test cases fail - we'll come to that in a moment.

Part 3: Trello Credentials

Your test cases might fail because you've not passed the correct environment variables to the containers. If so, now's the time to fix that. You can load secrets from a `.env` file using the `--env-file` option, or using `-e` to set an env variable directly.

Make sure each `docker run` command (if you had multiple) receives the correct credentials.

Part 4: Unit and Integration Tests

Your unit and integration tests should now run in Docker. Make sure they do! If you hit any errors, now is the time to investigate and fix them.

Part 5: Selenium in Docker

You can run selenium end-to-end tests in your Docker container, but first you'll need to install a supported webdriver and browser. You can have a go at this yourself, but it can be a little fiddly.

To help, here's a a dockerfile snippet that installs the latest versions of chrome and the chromium webdriver:

```
# Install Chrome
RUN curl -sSL https://dl.google.com/linux/direct/google-chrome-
stable_current_amd64.deb -o chrome.deb &&\
    apt-get install ./chrome.deb -y &&\
    rm ./chrome.deb

# Install Chromium WebDriver
RUN LATEST=`curl -sSL https://chromedriver.storage.googleapis.com/
LATEST_RELEASE` &&\
    echo "Installing chromium webdriver version ${LATEST}" &&\
    curl -sSL https://chromedriver.storage.googleapis.com/${LATEST}/
chromedriver_linux64.zip -o chromedriver_linux64.zip &&\
    apt-get install unzip -y &&\
    unzip ./chromedriver_linux64.zip
```

With that in place, your Python code should be able to find and launch the chromium webdriver as part of your selenium end-to-end tests.

The pytest fixture below shows one approach to launching selenium with chromium within docker. If you hit any errors, try adding some of these options.

```
@pytest.fixture(scope='module')
def driver():
    opts = webdriver.ChromeOptions()
    opts.add_argument('--headless')
    opts.add_argument('--no-sandbox')
    opts.add_argument('--disable-dev-shm-usage')
    with webdriver.Chrome('./chromedriver', options=opts) as driver:
        yield driver
```

Congratulations, you can now run your unit, integration and end-to-end tests within docker! You can use this as a convenient way to run tests locally, or on any platform that supports docker. That includes Travis CI. Return to the main exercise.