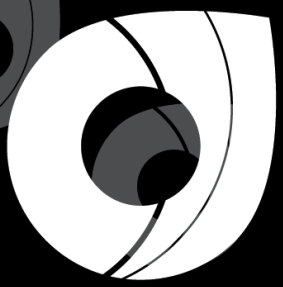
Corndel DevOps Engineering Programme

in association with Softwire

Module 8: Project Exercise



Corndel Digital.

DAC3

Module 8

Project Exercise Brief

In the previous Module you created a CI pipeline using Travis CI that builds and tests your code on every push. If you managed the stretch goals, it might also provide you with build alerts via slack or email.

In this exercise you will expand this CI pipeline to publish build artefacts. You'll then extend the pipeline further, practicing **continuous deployment** to Heroku (a hosting platform). You'll end up with a fully automated pipeline from source code to public deployment.

Part 1: Build Artefacts

First you'll need to expand your Travis CI pipeline to publish production docker images to a new repository on Docker Hub. A public docker registry (such as Docker Hub) is a great place to share build artefacts for open-source projects, as it's extremely easy for anyone running docker to download, run and integrate with your app.

Step 1: Docker Login

Before you can push images to Docker hub, you need to log in. On your local machine you can simply run `docker login` and log in interactively, but you'll need to handle this slightly differently on a CI server.

- Add your docker hub username and password as two new config values in your CI pipeline. The password must be secret (encrypt it!), but the username isn't sensitive.
- Run `docker login` non-interactively, using your environment variables to supply the username and password (see https://docs.docker.com/engine/reference/commandline/login/).

Step 2: Build and Push

Expand your CI pipeline to build production images and push them to docker hub. Keep in mind that your pipeline should only push production images from the main branch. You don't want to publish build artefacts from in-development feature branches!

To recap, the basic commands for building and pushing your application to Docker Hub should look like:

```
$ docker build --target <my_build_phase> --tag <image_tag> .
$ docker push <image_tag>
```

where `<image_tag>` has the format `<user_name>/<image_name>:<tag>`.

Make sure you set appropriate image tags! The most recent production image needs to be tagged `latest`. If you want to keep older images - often good practice - you'll need to tag each build uniquely. Teams often tag images with the git commit hash so they are easily identifiable. You can access the commit hash via the `TRAVIS_COMMIT` environment variable (Travis provides a few default env variables, see: https://docs.travis-ci.com/user/environment-variables/#default-environment-variables).

It's important to keep `.travis.yml` organised as you expand it. That helps keep it maintainable for the future. Before you start, we recommend reviewing Travis CI's job lifecycle. Which stage should each operation live in? You can also move some commands to separate shell script files and reference these files from `.travis.yml`.

Part 2: Manual Heroku Deployment

Now you have production images, it's time to deploy them. In this stage you'll perform an initial manual deployment, and later we'll automate future deployments using Travis CI.

Step 1: Create & Configure Heroku App

First, create a new Heroku app either using the web interface or CLI. You should already have a Heroku account from this Module's workshop. If not, set one up now - it's free. Make a note of the app's name.

Once you've created an app, you'll need to provide it with the production environment variables your code needs to run. That will include your trello API credentials, and the board ID you want to use. *Warning*: The board you use will be visible to the public internet. If your current version contains any private or otherwise sensitive data you should create a new production-only board and use that ID instead.

You can set config values via the Heroku CLI; for example, the following snippet uploads the TRELLO_API_KEY stored in your `.env` file:

```
$ heroku config:set `cat .env | grep TRELLO_API_KEY`
```

Step 2: Push an Image to Heroku

Heroku can't deploy images from docker hub directly, but instead uses its own (private) docker registry. You need to push your image there, then tell Heroku to deploy it.

```
# Get the latest image from Docker Hub (built by your CI pipeline)
$ docker pull <user_name>/<image_name>
# Tag it for Heroku
$ docker tag <user_name>/<image_name> registry.heroku.com/
<heroku_app_name>/web
# Push it to Heroku registry
$ docker push registry.heroku.com/<heroku_app_name>/web
```

For this to work you need to be logged in via the Heroku CLI. You've probably already done this, but if not try running `heroku login`.

Step 3: Release it!

Pushing an image to Heroku doesn't release it immediately. Instead, you'll need to use the `heroku container:release web` command. You can run `heroku open` to automatically open the web app in your browser.

Make sure your app works on Heroku. If it doesn't, there are a couple of things to check:

- Have you set all the necessary Trello config variables?
- Does your Dockerfile comply with the restrictions listed by Heroku? Note that Heroku requires your app to listen on a port defined by the `$PORT` environment variable - you may need to adjust your Dockerfile.

Part 3: Continuous Deployment

In Part 1 you expanded your CI pipeline to produce production-ready docker images, and in Part 2 you learnt how to deploy these images to your Heroku environment from the command line. Now it's time to combine those two.

Expand your Travis CI pipeline into a continuous deployment pipeline that:

- Automatically builds and deploys your main branch to Heroku
- Also publishes the Docker images to Docker Hub

Test it out! Push a small change to `main` branch, wait for the pipeline to complete, then check it's visible on the live site.

Stretch Goals & Hints

(Stretch Goal) Pipeline Optimisation

Now your CI/CD pipeline is running, you should review its performance. Look at the Travis CI web interface and note down:

1. How long does the CI/CD pipeline take to run?
2. How long does each *stage* of the pipeline take? You might want to consider:
    1. Travis CI overheads (e.g. configuring a VM)
    2. Docker installation
    3. Building docker images
    4. Running the test suite
    5. Pushing docker images
    6. Executing Heroku CLI commands
3. Do all builds take the same time?
    1. Are identical, repeated builds fast?

Depending on how you've written your Dockerfile and Travis CI pipeline, it's likely that your pipeline takes a few minutes (probably less than 5), but is far from instant. The build time will probably be dominated by one or two steps of the pipeline.

A pipeline that takes a few minutes is not terrible. Many businesses can work with this, but it is far from ideal from a DevOps perspective. Can you think of any ways to make it faster? Have a go.

(Hint: Think about how you can avoid re-building the same docker layers every time)

(Stretch Goal) IaaS vs PaaS/SaaS

In this exercise you have used Heroku and Travis CI to deliver a functional, and free, CI/CD pipeline. Travis CI is a kind of Software as a Service (SaaS), while Heroku is a good example of Platform as a Service (PaaS). They are relatively high-level kinds of cloud DevOps tools that it quick and easy to get up and running, provided you are happy with decisions these providers have made to build their platforms.

Sometimes teams and businesses have particular requirements that make SaaS and PaaS solutions unsuitable or overly costly. They will instead look to IaaS (Infrastructure as a Service) companies, who can provide cloud-based virtual machines on which a team can deploy whatever software they like. IaaS solutions are far more flexible and have typically have a smaller pricetag than SaaS/PaaS solutions, but are much more time-intensive to set up, and might require long-term effort to maintain and update.

For this stretch goal we don't want you to code anything, but consider how you would implement the equivalent CI/CD functionality using IaaS only - i.e. you are provided with some cloud-based virtual machines and networking infrastructure only.

- What software would you need?
- How would you maintain it?
- Are there convenient off-the-shelf packages you can deploy, or would you need to write custom applications to mimic PaaS/IaaS offerings?

Hints

- The `docker login` command has the `--password-stdin` option to read a password from STDIN. This is more secure than passing it as a direct argument. For example `echo $PASSWORD | docker login --username $LOGIN --password-stdin` will log in to Docker hub using credentials read from environment variables.
- Dockerfile `ENTRYPOINT` and `CMD` array syntax does not support variable substitution, so won't work with `$PORT`. You'll need to use the string format, or write a separate entrypoint script.
- You may encounter issues creating Python virtual environments in docker containers running on Heroku. If so, adjust your Dockerfile to run poetry without a virtualenv (e.g. `RUN poetry config virtualenvs.create false --local && poetry install`). There's no need to create a virtualenv inside docker: the container already provides an isolated Python environment.
- Travis has a Heroku deployment option, but this doesn't work for Docker deployments. Instead, use the `script` deployment option and write your own commands.
- You can set Travis CI's `deploy` stage to only trigger on a specific branch (see https://docs.travis-ci.com/user/deployment/script/).
- `heroku login` expects a user to interact with a browser, and is not suitable for use in a CI pipeline. Fortunately, you can log in to the heroku container registry directly with `docker login`, following the documentation here.
- One way to avoid re-building docker layers unnecessarily is to pull your old images from a registry (e.g. docker hub) at the start of a pipeline. The docker daemon can then re-use layers from these images if you identify them as a reliable source using the `docker build --cache-from` option.