

ROB N

 menu

Dockerising a Perl application

The latest fancy tool for deploying things is Docker. I've been following it for about a year with some interest, but after deploying GitLab with Docker last week in about two hours (no exaggeration), I figured it was time to see how to drive this thing. I did this by “dockerising” a simple but non-trivial Perl application.

The point of all this

As I've been working on this I realised what Docker gives everyone, including the Perl community, and I wanted to put it up here so that if you don't care about the details, you still hear this.

If you remember back far enough you'll remember that Perl got big originally because if you wanted to make a simple web script, your hosting provider probably already had a `cgi-bin` dir and Perl with its venerable `CGI` module, so it was really easy. And then later we lost because every hosting provider had PHP, and you'd install some random forum or blog application by just copying the entire application directory to the web space (often via FTP), and we didn't have a Perl embedded in the web server so we couldn't compete.

This changes all that. Lots of places have Docker already, and both Amazon and Google have committed to supporting it on their clouds. We now have a way to get our applications to everyone on the biggest hosting providers with a single command. For application deployment this levels the playing field. If you want to use Perl for your webapp and get it out to lots of people, you no longer need to

worry about deployment problems. That's a game changer for us.

So do this. Dockerise your Perl application. Its not hard to do and helps you get it out in front of more people.

The application: towncrier

I didn't write about it when I made it, so let me briefly introduce the application.

A few months ago we decided that we'd prefer to have a nice status dashboard to report the state of our services rather than the blog we've had for years. Since its such a simple style of application it looks like most people just build their own, so there wasn't a lot out there to choose from. I spent some time with Stashboard, a standalone dashboard from Twilio. This was pretty much exactly what I wanted but it only works on Google App Engine, which at the time couldn't let me a specific subdomain (ie status.fastmail.fm) at it. I spent some time looking at GAE compatibility layers for other cloud providers but it all got more fiddly than I wanted. I really just wanted something that I could deploy and forget.

The obvious solution: write my own. I figured that it wouldn't take long to port Stashboard to a stack I'm at least a little bit familiar with, so I spent my evenings for the next couple of weeks writing a fairly straightforward Dancer app. Its not hugely complicated, but does demonstrate a simple webapp with an object store, a REST API and a RSS feed. I'm sure there's a few things that aren't done "right", but good working code beats perfect non-existent code any day.

I promptly released it and deployed it. I chose not to put the whole application on CPAN for reasons I won't go into here (its a whole post really), but deploying it still isn't hard. Pull the git repo, install the dependencies from CPAN, then start it up. As far as I know nobody else has done that, but that's ok since its now serving its purpose.

The thing that has surprised me since is how handy its been to have a simple but non-trivial application around to try out new ideas. Its been a really nice platform for experimenting with API construction, object datastores, web authentication, etc. And this week its been useful for experimenting with deployment tools. I set myself the goal of making it possible to deploy towncrier using Docker.

Carton for Perl dependency installation

So before I get into Docker proper, I need to take a short detour to a nice tool to help get Perl dependencies up and running on stuff that isn't installed through CPAN. That tool is Carton, and if you're a Ruby person you'll understand it by

knowing that its Perl's version of Bundler.

The short story of Carton is that you add a `cpanfile` to your project that lists all the packages it depends on. Later you run `carton install` and they all get installed alongside the application (using `local::lib`) by whatever means necessary (typically either from CPAN or from git repositories). Its not so different to what `cpanm` does but without installing into the system Perl and without requiring other Perl build stuff to be present (ie `Makefile.PL`). Once that's done, you run things with `carton exec foo.pl` and it'll set up Perl's library path appropriately before it runs the program.

The reason all of this is interesting for Docker is that after we clone the repository we can fetch all of its dependencies with a single command. That lets us keep all the dependency info out of the Docker config proper.

No more info on Carton for now. Go and read more about it. Its a good tool to have in the box.

Preparing a Docker image

This just covers the basics for what we need to do. Docker's documentation is pretty good for everything else, especially once you understand its model.

You drive Docker with a `Dockerfile`, which is basically a script that tells Docker what actions to take. Docker starts from a base image, typically a complete Linux distro install image. Then it runs each command, snapshotting the image state after each one. At the end of the process you're left with an image that contains the base image and your changes. You can then run this image, copy it somewhere, redeploy it, publish it for others to use, and so on. The snapshot at each step means you can roll the image back or partially rebuild it. Its great while you're creating your `Dockerfile`.

This is towncrier's `Dockerfile`:

```
# DOCKER-VERSION 0.3.4
FROM perl:latest
MAINTAINER Robert Norris rob@eatenbyagrue.org

RUN curl -L http://cpanmin.us | perl - App::cpanminus
RUN cpanm Carton Starman

RUN cachebuster=b953b35 git clone http://github.com/robn/towncrier.git
RUN cd towncrier && carton install --deployment

EXPOSE 8080

WORKDIR towncrier
```

```
CMD carton exec starman --port 8080 bin/app.pl
```

Lets go through it a line at a time.

```
FROM perl:latest
```

This is our starting point, which is the `latest` tag from the `perl` repository on Docker Hub. This is a pre-built image built from its own `Dockerfile`, which in turn is constructed from other images until eventually we find that its a Debian Jessie install at the bottom. That doesn't really matter much for our purposes since all we're interested in is a sane Perl environment, which this gives us. The `latest` tag moves as new versions of Perl are released, so building our `Dockerfile` gets us 5.20. That's good!

At this point, we have a complete OS install with Perl ready to go.

```
MAINTAINER Robert Norris rob@eatenbyagrue.org
```

Arranges for author information to be included in the image. This line is not at all important except to note that every command produces a new Docker snapshot image, even ones that don't appear to do anything interesting.

```
RUN curl -L http://cpanmin.us | perl - App::cpanminus
```

`RUN` runs the given command in a shell. This one will be familiar to a lot of people - it bootstraps `cpanminus`, the CPAN installer.

So now we have Perl with `cpanm` and we can install whatever we want.

```
RUN cpanm Carton Starman
```

Here we install two CPAN packages (and dependencies): `Carton`, so we can install `towncrier`'s dependencies later, and `Starman`, a high-performance Perl web server. There's other options, but this one I've used in production before and I know it well. `towncrier` doesn't depend on it directly so `Carton` won't install it, which is why we grab it explicitly here.

```
RUN cachebuster=b953b35 git clone http://github.com/robn/towncrier.git
```

This one is interesting. The `git clone` is entirely what you'd expect. But what's this `cachebuster` business?

Docker snapshots the image state after each line in the `Dockerfile`. If you rebuild the image and nothing has changed then it reuses the snapshot rather than rebuilding the entire image state. In this way you can make changes to later commands without having to rebuild everything which is great because the bigger changes (installing a whole OS, building a large application) are usually done earlier.

Whether or not an intermediate image gets rebuilt is entirely dependent on whether or not the corresponding line in the `Dockerfile` changes. If its the same, no rebuild takes place. This sucks if the line in question is a `git clone` because that line doesn't change even if there's upstream changes.

The `cachebuster` thing is my workaround. I just set a variable to some random value. If I want to force that line to rebuild, I just change it. I've made it the git short hash of the repo for my own reference, but it can be anything you want, just as long as its different each time.

This is a known issue that will be fixed in time. This workaround seems good for now.

```
RUN cd towncrier && carton install --deployment
```

By now you should know what's going on. We've just done a checkout, so we `cd` into it and run `carton` to install all the dependencies.

Once this has finished, we have all the software we need installed inside our image. There's just a few bits of final setup to go.

```
EXPOSE 8080
```

This tells Docker that the image will start something on port 8080 and that it should be prepared to expose that port outside the container. Note that this doesn't actually expose the port, just tells Docker that it could be exposed. Later when we start the container we can bind that to a host port.

```
WORKDIR towncrier
```

A simple one. It just tells Docker to automatically `cd` into this dir the for `RUN` and `CMD` commands (we'll see `CMD` in a moment).

```
CMD carton exec starman --port 8080 bin/app.pl
```

`CMD` is like `RUN` except the command isn't run now, during image construction, but rather when the container is started. Its your "application start" command. Here we're getting `carton` to set up the local Perl environment and then run `starman` under it, binding it to a port and running the towncrier application. This is the normal way to start a Dancer app.

And now our `Dockerfile` is ready, we can build it:

```
$ docker build -t robn/towncrier .
```

`.` for the `Dockerfile` in the current directory. I'm not sure why it takes a directory rather than the `Dockerfile` itself, but its not a big deal. `-t` lets us name our image, otherwise it'll get a hash for name. Including my name in the image name will help later when it comes time to distribute our image on Docker Hub.

Time passes. You can watch the progress if you like. I went and made coffee because that's what I do when I have to wait. You probably have your own waiting strategy. Do that.

Running the application

Now the fun bit: running it.

```
$ docker run --name=towncrier -it --rm -p 8080:8080 robn/towncrier
```

And you get the regular startup:

```
2014/07/16-21:45:34 Starman::Server (type Net::Server::PreFork) starting! pid(6)
Resolved [*]:8080 to [0.0.0.0]:8080, IPv4
Binding to TCP port 8080 on host 0.0.0.0 with IPv4
Setting gid to "0 0"
```

And now you can go to localhost:8080 and see towncrier's initial page. So that's working!

The switches and options are pretty well documented. `-it --rm` is pretty standard for testing. `-p` hooks an `EXPOSE`d port inside the container to a port

on the host system. `80:8080` would put it on port 80 instead, even though the container thinks its still on 8080.

That's pretty much it for actually getting the thing running.

Configuring the application

So all this so far is good if you just want the defaults for the application, but chances are you don't. Even something as simple as towncrier requires at least one config parameter - setting the API admin password.

A Dancer application is normally configured with a `config.yml` file in the application's base directory. The problem is that the default config file is stuck inside the container, which is pretty much off-limits to us. Docker has a way to mount a host directory inside a container and we'll look at that in a moment. First we'll look at the preferred way to configure a Docker container, which is that old reliable favourite: environment variables.

You can set environment variables for the Docker `CMD` program with the `-e` switch to `docker run`. A simple example for towncrier is to set the page title, which normally comes from the `towncrier.title` config item. We can override that in the environment like so:

```
$ docker run --name=towncrier -it --rm -p 8080:8080 -e TOWNCRIER_TITLE="How's my
```

Reload the page and you'll see its changed. Great.

To do this you need to make sure you accept environment variables and use them to override your config. I use this pattern in Dancer:

```
my $title = $ENV{TOWNCRIER_TITLE} // config->{towncrier}->{title} // "Service sta
```

Most Dancer config parameters are settable from environment variables too, but its not documented. I'm going to submit a doc patch as soon as I get time, but for now you can look at the code for `load_default_settings` in `Dancer::Config`.

Not all configuration lends itself easily to environment though. One example is `Dancer::Plugin::Auth::Basic`, which takes a list of usernames and passwords from the config file and isn't configurable from the environment. I did consider sending them a patch but multi-valued fields are a bit tricky in environment variables, and since I currently only use one username and password for the admin API (all I need for now), I instead got rid of its config entirely and added a

small hook in the code:

```
use constant TOWNCRIER_ADMIN_USER =>
    $ENV{TOWNCRIER_ADMIN_USER} // config->{towncrier}->{admin_user} // "admin";
use constant TOWNCRIER_ADMIN_PASSWORD =>
    $ENV{TOWNCRIER_ADMIN_PASSWORD} // config->{towncrier}->{admin_password} // "s

hook before => sub {
    return unless request->path_info =~ m{^/admin/api/v1/};
    auth_basic
        realm    => 'api',
        user      => TOWNCRIER_ADMIN_USER,
        password => TOWNCRIER_ADMIN_PASSWORD;
};
```

And with that, we can set the username and password from the environment too.

Getting at the filesystem

There are times when you need to get at the filesystem. For towncrier there's two real places where you might want to do that. One is to be able to provide a config file from outside the container, and the other is to be able to get at the SQLite database for backups or whatever. Lets do both of these things.

First we just make a regular directory and put towncrier's config into it:

```
$ mkdir -p /opt/towncrier/data
$ curl -L https://raw.githubusercontent.com/robn/towncrier/master/config.yml -o /
```

Now we edit the config and change the location of the database:

```
database: /opt/towncrier/data/towncrier.sqlite
```

And then we just need to run the application telling Docker to make this directory available to the container, and then provide environment to tell the application where to find its config.

```
$ docker run --name=towncrier -it --rm -p 8080:8080 -v /opt/towncrier/data:/opt/t
```


towncrier creates its database on first access, so load the page and then:

```
$ ls /opt/towncrier/data  
config.yml  towncrier.sqlite
```

And that's it. Fiddle the config and restart, backup the database, whatever else you want. There's nothing special about these files; its just a union mount so they're literally the same files that are visible inside the container, with all the possible concurrency pain that that entails :)

Publishing your image

If you're building Docker images for your own use, that's as much as you need to do. If you want to give your ready-built image to other people though then you need to publish it. As noted, Docker Hub is the easiest place to do that. There's not much to it: create an account, create a repository, then login:

```
$ docker login  
Username: robn  
Password:  
Email: robn@eatenbyagrue.org
```

Tag your image with a repository name and tag:

```
$ docker tag a359fe81a0cc robn/towncrier:latest
```

And then push it:








```
$ docker push robn/towncrier:latest
```

And that's it. And now to get your app all I have to do is go to a Docker-aware hosting provider and do the same `docker run` line as we have above, and it will download the image and run it. One command to deploy and run an entire app. That's pretty amazing.



Rob N

I do things. Mostly computer things. Sometimes I write about them.

 Email  Twitter  Facebook  Google+  Github
 Stackoverflow  Steam

Dockerising a Perl application was published on July 15, 2014.

YOU MIGHT ALSO ENJOY

(VIEW ALL POSTS)

- Adding XMPP support to nginx
 - Phillip Hughes
 - Why I bought a Pebble
-

© 2015 Rob N. Powered by Jekyll using the Minimal Mistakes theme.