Dockerized Postgresql Development Environment

2015-03-24

My local development environment is kind of a mess. I am running OSX and use a variety of techniques to run the projects I work on. For example, I have local postgresql, redis, and memcache servers running to support some projects. Every once in a while I run into issues where a version of the service I am running for one project is not compatible with the version I need on another project. This problem can be solved with virtualized, per-project development environments using a tool like Vagrant.

On most of my recent projects, this is actually how I do things. But, not on all of them. I certainly could go back and create Vagrant environments for projects that depend on local resources but haven't found the time.

I recently installed Linux on my laptop and, rather than install local services like postgresql for development purposes, I decided I would run them in Docker containers. In this post, I will describe how I setup postgresql to run in a Docker container with (mostly) persistent storage.

By default, when using the official postgres docker image, you can lose [1] your postgresql data, including users, databases or other objects, if you ever remove the container. Since my workflow often involves removing and recreating containers, I wanted a way to ensure that I could easily start a container with a pre-configured postgresql server. I didn't want to have to recreate roles or databases every time I removed or recreated my postgresql container.

The Data Volume Container

To accomplish this, you can use a data volume container. This is a container that can be used to keep persistent data for non-persistent containers or to share data between multiple containers. Here is the command I used to create a data volume container for my postgresql database:

```
docker create -v /var/lib/postgresql/data --name postgres9.3.6-data busybox
```

I use the `-v` option to specify the name of the volume. This should match the name of the volume that your postgresql image mounts for its data and configuration. In the case of the official postgres image that I am using in this example, you can see that it is */var/lib/postgresql/data*.

I also use `--name` to give a descriptive name for the data volume container so you can easily see it when running `docker ps -a`.

Note that I version the name of my containers since I sometimes need to run multiple different versions of postgresql.

Finally, I am basing the container on busybox which is an extremely lightweight image.

The Postgresql Container

Now, let's create the postgresql container.

```
docker run --name local-postgres9.3.6 -e POSTGRES_PASSWORD=asecurepassword -d --
volumes-from postgres9.3.6-data postgres:9.3.6
```

Again, I am using `--name` to specify a versioned name for my container. I am using an environment variable (`-e`) to set the password for the *postgres* user. This is specific to the official postgresql image — see the documentation for more options. `-d` tells the docker to run the container as a daemon. You'll want to use a better password when running the above command.

The part we are interested in is the `--volumes-from` option. This tells the container to mount the */var/lib/postgresql/data* volume from the *postgres9.3.6-data* data volume container that we created in the previous step. Finally, I am using the *9.3.6* tag of the *postgres* image to launch the container.

Making Changes to Postgresql

We can now use *psql* to configure our postgresql server. Run the following command to establish a connection:

```
docker run -it --link local-postgres9.3.6:postgres --rm postgres:9.3.6 sh -c 'exec
psql -h "$POSTGRES_PORT_5432_TCP_ADDR" -p "$POSTGRES_PORT_5432_TCP_PORT" -U
postgres'
```

This command is lifted straight from the postgresql image documentation. It uses `--link` to enable the connection to the postgresql instance running in the container we just created. As you can see, this command is pretty unwieldy — you can use a shell alias to make it more manageable if you need to run it often.

Now we can go ahead and make changes to the postgresql server. For example, let's create a role and a database:

```
postgres=# CREATE ROLE myapp WITH CREATEDB LOGIN PASSWORD 'secret';
CREATE ROLE
postgres=# CREATE DATABASE myapp_development;
CREATE DATABASE
postgres=# \q
```

That data and configuration is being persisted in a shared data volume and will be persisted as long we don't remove the *postgres9.3.6-data* container we created in the first step. For example, I can stop and remove my *local-postgres9.3.6* container, recreate it (using the `--volumes-from` option) and still see that the user and database still exists.

```
docker stop local-postgres9.3.6

docker rm -v local-postgres9.3.6

docker run --name local-postgres9.3.6 -e POSTGRES_PASSWORD=asecurepassword -d --
volumes-from postgres9.3.6-data postgres:9.3.6

docker run -it --link local-postgres9.3.6:postgres --rm postgres:9.3.6 sh -c 'exec
psql -h "$POSTGRES_PORT_5432_TCP_ADDR" -p "$POSTGRES_PORT_5432_TCP_PORT" -U
postgres -l'
```

You should see the *myapp_development* database that we created in the list of databases returned by that last command.

Exposing Postgresql to the Host

You may have noticed that whenever we started the *local-postgres9.3.6* container that we were never exposing the service to the host machine. This means that postgresql would only ever be accessible by other containers started using the `--link` option. You may want to be able to connect to the dockerized postgresql instance from applications on the host — for example, a locally-installed copy of *psql* or a Rails application running outside of a Docker container. To do this, you can run the following:

```
docker run --name local-postgres9.3.6 -p 5432:5432 -e
POSTGRES_PASSWORD=asecurepassword -d --volumes-from postgres9.3.6-data
postgres:9.3.6
```

Make sure you have stopped and removed the *local-postgres9.3.6* container if you had it running previously. Otherwise, you'll get a name conflict and the container won't start.

Now, you can connect to that instance from applications on your host:

```
psql -h localhost -p 5432 -U postgres
Password for user postgres:
psql (9.3.6)
Type "help" for help.

postgres=#
```

Note, I am exposing *5432* directly on the host. If you want to start multiple instances of postgresql (different versions, for example), you'll want to adjust the ports to prevent conflicts.

And, that's it. Now, I have a very flexible development environment setup. I can run multiple different versions of postgresql for different applications without too much of a hassle. And, it's a little lighter weight than using virtual machines with Vagrant. You can use this same technique for other services that might benefit from persistent storage. For example, I do the same thing with Redis servers.

I am really just getting started with using Docker and containers in general. I think it is pretty exciting technology. I'd love to hear from you - have you bought into Docker and how are you using it in your day-to-day work? Please email me and let me know!

Are you interested in topics like this? Drop your email below to receive regular updates.

1. Technically, your data will not be lost unless you use the *-v* flag when you remove the container. For example,

   ```
   docker rm -v mypostgrescontainer
   ```

   If you neglect to include the `-v` flag when removing the container, the data will remain in a dangling volume and, while still present on your host's file system, not very easy to work with and not automatically mounted when you start a new postgresql container. There is some work going on to make working with volumes easier. ↩

devopsdocker
Troubleshooting AWS Elastic Beanstalk Errors

Securing a Server with Ansible