Setting up nginx with Docker

Introduction

I wanted to learn about how to use the popular proxy server nginx for handling "reverse proxy" duties on a Ruby project I was working on. So I decided the easiest thing to do in order to play around with nginx (as I work on a MacBook Pro laptop), would be to install and run it (and my back-end application) within Docker containers.

> Note: I did some preliminary Googling and sadly didn't find anything straightforward that demonstrated this relatively simple requirement (i.e. run nginx and have it proxy requests to a back-end service). So I decided it would be best to write about it myself

Now if you're unsure of what a reverse proxy does, it simply takes *in* traffic (i.e. users requesting a website domain) and proxies those requests onto another service (the service could be external to the host server or it could be running on the same box - which is the scenario I have - and typically the service being proxied to isn't publically available to the internet).

The nginx server has many features, such as load balancing, caching and serving static files (to name but a few). When used as a reverse proxy it can also be useful for handling SSL termination (i.e. SSL is an expensive operation and so the proxy server will authenticate the provided credentials - allowing access - but terminate the SSL requirement at that point before directing traffic onto the other protected service).

In this post we'll be primarily focusing on using nginx as a reverse proxy, although I also demonstrate how to serve static files. But the focus will be reverse proxying and not demonstrating other features within nginx. I'll also not be explaining Docker and how it works (I'm assuming you've used Docker in some form or another previously).

Lastly, I'm going to take you on the same journey I took while setting this all up; so rather than work through a perfect scenario you'll get to see some of the errors I stumbled across along the way. If you'd prefer to just read the (small amount of) code then the next section is for you.

Just give me the code

github.com/integralist/docker-examples/nginx

Setting up nginx

So to begin with, I went to Docker Hub and found the official nginx Docker image. It suggested the easiest thing to do to get started was to download and run the image; so knowing not a lot about nginx, that's exactly what I did:

```
docker run --name nginx-container -P -d nginx
```

> Note: you can use the `-p <host_port>:<container_port>` option instead of `-P` if you're not going to be running multiple nginx instances (again, for the moment I was just following the recommendation from Docker Hub)

Once the container was running I attempted to `curl` the endpoint (something like `http://localhost:<port_number_container_mapped_to>`) but I found this didn't work; and by that I mean it didn't return a recognisable nginx home page as was suggested it would. This was the first trip-up I made.

The reason this didn't work is because (for me) using Boot2Docker on a Mac - rather than being on a pure Linux machine capable of running Docker natively - means that localhost is the Mac and not the Docker host (which is the Boot2Docker VM). So to resolve that issue I needed to curl the ip of the Boot2Docker host. To get the ip simply run `boot2docker ip`.

So your curl command (depending on what OS you're running on) should look something like:

```
curl http://$(<docker_host_ip>:<port_number_container_mapped_to>
```

For me this was:

```
curl http://$(boot2docker ip):32781
```

Executing this resulted in the following output:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Mounting our own static files

Now at this point having a standard nginx welcome page isn't very useful. We want to add our own nginx configuration and to have it serve up our own static files. To achieve that with Docker we'll need to use Volumes.

But before I show the command for that, I need to show you the project directory I have right now:

```
.
├── Dockerfile
├── Gemfile
├── Gemfile.lock
├── app.rb
├── nginx.conf
└── html
    └── test.html
```

This isn't the final directory structure mind you, but it's what I initially started out with. You'll see as we move on I had to add a little more structure to the project. But for now, this is what we've got.

I'll show the contents of these files now, most of them are really small anyway:

Dockerfile

```
FROM ruby:2.1-onbuild
CMD ["ruby", "app.rb"]
```

Gemfile

```
source "http://rubygems.org/"

gem "sinatra"
```

app.rb

```
require "sinatra"

set :bind, "0.0.0.0"

get "/" do
  "Hello World"
end

get "/foo" do
  "Foo!"
end
```

nginx.conf

```
user nobody nogroup;
worker_processes auto;          # auto-detect number of logical CPU cores

events {
  worker_connections 512;       # set the max number of simultaneous connectior
}

http {
  server {
    listen *:80;                # Listen for incoming connections from any inte
    server_name "";             # Don't worry if "Host" HTTP Header is empty or
    root /usr/share/nginx/html; # serve static files from here
  }
}
```

html/test.html

```
<h1>Hey there!</h1>
<p>Here is my test HTML file</p>
```

Mounting the files

Let's see the syntax structure of the command I was looking to run:

```
docker run --name nginx-container \
  -v /path/to/static/files/on/host:/usr/share/nginx/html:ro \
  -v /path/to/conf/on/host:/etc/nginx/nginx.conf:ro \
  -P -d nginx
```

In the above example, you can see I'm using the `-v` flag to mount my static files directory to `/usr/share/nginx/html` as well as mounting my own nginx configuration file into the container at `/etc/nginx/nginx.conf` .

> Note: `:ro` sets the volumes to be "read only"

So for me, a working example looked like the following:

```
docker run --name nginx-container \
  -v $(pwd)/html:/usr/share/nginx/html:ro \
  -v $(pwd)/nginx.conf:/etc/nginx/nginx.conf:ro \
  -P -d nginx
```

> Note: the host path has to be absolute, so tweak it as necessary (I used `pwd` to make the command shorter)

If you want to debug things, then you can run the container not as a daemon (`-d`) but with an interactive tty (`-it`) and drop yourself inside of a bash shell:

```
docker run -it --name nginx-container \
  -v $(pwd)/html:/usr/share/nginx/html:ro \
  -v $(pwd)/nginx.conf:/etc/nginx/nginx.conf:ro \
  -P nginx bash
```

Once the container is running and the `nginx.conf` and static files are mounted as a Volume, you can verify that nginx is serving the static files by trying to hit localhost on port `80` (as mentioned earlier: if you're on a Mac using Boot2Docker like me, then you'll need to access localhost via the Boot2Docker VM ip address instead):

> Note: don't forget the port number I've used will be different for you (get yours from `docker ps`)

```
curl http://$(boot2docker ip):32781
```

This will display a `403` error page, which is to be expected because we've not mounted a folder that has a `index.html` file. Our mounted directory only contained a `test.html` file, so if we try to access that instead, then we'll see it'll be loaded without a problem:

```
curl http://$(boot2docker ip):32781/test.html
```

If you were following along at home then you would have noticed that this returns the following HTML content:

```
<h1>Hey there!</h1>
<p>Here is my test HTML file</p>
```

If we try the exact same command again, but this time add the `-I` flag (e.g. `curl -I <url>` - this returns just the HTTP headers), we'll see that this is indeed being served by nginx:

```
HTTP/1.1 200 OK
Server: nginx/1.9.3
Date: Sat, 01 Aug 2015 17:27:01 GMT
Content-Type: text/html
Content-Length: 53
Last-Modified: Sat, 01 Aug 2015 14:05:59 GMT
Connection: keep-alive
ETag: "55bcd247-35"
Accept-Ranges: bytes
```

Dynamically updating containers without restart

Because we've mounted the `html` directory as a volume, we can now create an `index.html` file for nginx to use as the root page to load when you request `http://$(boot2docker ip):32781/` ; and in doing this it'll take effect immediately without requiring us to restart the container. Once we've created the file inside the `html` directory, we can then make the relevant curl request to see nginx serve up the html file.

Just so you know, the `index.html` file I created looks like this:

```
<h1>Welcome</h1>
<p>This is my home page</p>
```

Setting up the Ruby application

The Ruby application we'll be creating is a super simple Sinatra web application with two routes defined:

1. `/`
2. `/foo`

One of the first things you need to do is to make sure you run `bundle install` so you have a `Gemfile.lock` generated, otherwise the base Ruby image will complain. The reason it complains is because the base Docker image is using an `onbuild` version of the Ruby image, and what this means is that it follows a "convention over configuration" model where by it assumes you have three files available (`app.rb` , `Gemfile` and `Gemfile.lock`) for it to automatically copy into the built Docker image for you.

Once you've done that, you can now build the Docker image:

```
docker build -t my-ruby-app .
```

Once the image is built, you can run your Ruby application like so:

```
docker run --name ruby-app -p 4567:4567 -d my-ruby-app
```

You'll notice we're not using a dynamic port range because the Sinatra app we've created explicitly binds to port `4567` and so I'm exposing that specific port to the Boot2Docker VM. We can then access this application directly with the following curl command:

```
curl http://$(boot2docker ip):4567/

# Results in...

Hello World%
```

Now if we want to see the HTTP Headers coming back:

```
curl -I http://$(boot2docker ip):4567/

# Results in...

HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Content-Length: 11
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
```

```
X-Frame-Options: SAMEORIGIN
Server: WEBrick/1.3.1 (Ruby/2.1.6/2015-04-13)
Date: Sun, 02 Aug 2015 09:20:47 GMT
Connection: Keep-Alive
```

Linking your app to nginx

OK, so we're nearing the finish line now. We've got a working nginx container and a working Ruby container. We need to gel them together by making a request and having nginx proxy the request on to the Ruby container (if the request was aimed at that) or have nginx serve a static file if the request was relevant to that.

To get this to work with Docker, we have two options:

1. Host file
2. ENV variables

Both options require a custom Dockerfile to be used for setting up nginx and both have about the same amount of complexity in different ways.

But to understand these options, we need to be sure we understand what Docker does when linking containers. So to clarify, when linking a container (Ruby) with another container (nginx), Docker adds the ip of the linked container (Ruby) into the other container (nginx). It does this by updating the container's `/etc/hosts` file with a new entry that looks something like the following (note the last line of the file output):

```
host.conf  hostname   hosts
root@a4a3bde52f8e:/# cat /etc/hosts
172.17.0.19     a4a3bde52f8e
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.18     app 3435f6926e83 ruby-app
```

You can test this yourself by using the following command, which interactively jumps into a running version of the nginx container:

```
docker run --rm -it \
  --name nginx-container \
  --link ruby-app:app \
  -P nginx bash
```

Once inside the container, run `cat /etc/hosts` to get the above output (or something similar to it at least - your ip will likely be different to mine).

While you're still inside the temporarily running nginx container, we can clarify the second option which is the ENV variable setup Docker handles. When linking a container, Docker will add details of the linked container as environment variables (as well as adding a new entry to the `/etc/hosts` file). So while you're still inside the running nginx container, execute the following command:

```
env | grep APP | sort
```

> Note: we're `grep` ing for the phrase `APP` as that's what we specified in our `docker run` command (i.e. `--link ruby-app:app`)

We should now see the following output (or something very similar):

```
APP_ENV_BUNDLER_VERSION=1.10.6
APP_ENV_BUNDLE_APP_CONFIG=/usr/local/bundle
APP_ENV_GEM_HOME=/usr/local/bundle
APP_ENV_RUBY_DOWNLOAD_SHA256=1e1362ae7427c91fa53dc9c05aee4ee200e2d7d8970a891c5b
APP_ENV_RUBY_MAJOR=2.1
APP_ENV_RUBY_VERSION=2.1.6
APP_NAME=/nginx-container/app
APP_PORT=tcp://172.17.0.18:4567
APP_PORT_4567_TCP=tcp://172.17.0.18:4567
APP_PORT_4567_TCP_ADDR=172.17.0.18
APP_PORT_4567_TCP_PORT=4567
APP_PORT_4567_TCP_PROTO=tcp
```

Notice that Docker has capitalised the link name (e.g. `app` is now `APP`) and has used the convention of `<LINK_NAME>_<ENV_VAR_FROM_LINKED_CONTAINER>` .

Choosing an option?

With this knowledge secured in our minds, we can now understand what we need to do for each option (i.e. host file vs environment variable). For the host file option we can manipulate our `nginx.conf` file manually (before building the image) to use a named `upstream` server that references the `--link` name we provided, as part of the `docker run` command.

Where as with the environment variables option we would need to manipulate the `nginx.conf` file *dynamically* at run time. So within our `Dockerfile` we would add a shell script, which would be executed by the `CMD` statement within the `Dockerfile` . This shell script would run and use something like the `sed` command to replace a placeholder reference within our `nginx.conf` file (likely we'd use the `APP_PORT` environment variable and do some regular expression parsing for the ip address). Once the placeholder had been replaced with the appropriate container ip, we would manually start nginx using `service nginx start`.

Out of the two options, I think I'll go with the "host file" one. So let's run through the steps...

- Create a new directory structure that looks like the following (you can grab the exact files from github.com/integralist/docker-examples/nginx):

```
.
├── docker-app
│   ├── Dockerfile
│   ├── Gemfile
│   ├── Gemfile.lock
│   └── app.rb
├── docker-nginx
│   ├── Dockerfile
│   └── nginx.conf
└── html
    ├── index.html
    └── test.html
```

- Build the images (we'll see what these look like in a moment) for the Ruby app and nginx
- Run the Ruby app container
- Run the nginx container and link it to the running Ruby container
- Verify everything works

Other than moving existing files (related to the building of the Ruby Docker image) inside of a `docker-app` folder, the two biggest changes are the addition of a new `Dockerfile` inside the `docker-nginx` folder and the following updated `nginx.conf` file:

```
user nobody nogroup;
worker_processes auto;          # auto-detect number of logical CPU cores

events {
  worker_connections 512;       # set the max number of simultaneous connectior
}

http {
  upstream app {
    server app:4567;            # app is automatically defined inside /etc/host
  }

  server {
    listen *:80;                # Listen for incoming connections from any inte
    server_name "";             # Don't worry if "Host" HTTP Header is empty or
    root /usr/share/nginx/html; # serve static files from here
    location /app/ {            # catch any requests that start with /app/
      proxy_pass http://app;    # proxy requests onto our app server (i.e. a di
    }
  }
}
```

The aim of the new configuration is to accept any requests to `/app/` and proxy them onto our application server. So if we were to request `/app/` we'd want nginx to proxy it to the Sinatra `/` route; and if we were to request `/app/foo` then we'd expect nginx to proxy it to the Sinatra `/foo` route.

The new `Dockerfile` we've created inside the `docker-nginx` folder will have the following content:

```
FROM ubuntu

# install nginx
RUN apt-get update && apt-get install -y nginx
RUN rm -rf /etc/nginx/sites-enabled/default

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log

EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```

> Note: some bright chap by the name of Steven Jack suggested setting `ENTRYPOINT` in the Dockerfile to `nginx` and have the two options (`-g` and `daemon off`) left in the `CMD` . Doing this means if you wanted to pass other options to nginx you wouldn't have to duplicate all of your CMD

So from the root of our project directory, let's complete the next step and build our new Docker images (technically you don't have to rebuild the Ruby application as there has been no changes to the `Dockerfile` , but just for completion I'll demonstrate the build command again).

First let's rebuild the Ruby Docker image:

```
docker build -t my-ruby-app ./docker-app
```

Now let's build our new custom nginx Docker image:

```
docker build -t my-nginx ./docker-nginx
```

The third step in our list was to run the Ruby container:

```
docker run --name ruby-app -p 4567:4567 -d my-ruby-app
```

Let's also verify that it's running (the following command should return back "Hello World"):

```
curl http://$(boot2docker ip):4567/
```

Now onto our fourth step, which was to run our custom nginx container whilst linking it to our already running ruby container:

```
docker run --name nginx-container \
  -v $(pwd)/html:/usr/share/nginx/html:ro \
  -v $(pwd)/docker-nginx/nginx.conf:/etc/nginx/nginx.conf:ro \
  --link ruby-app:app \
  -P -d my-nginx
```

The final step is to verify that everything worked as expected (i.e. we should be able to make a request to our Boot2Docker VM's localhost and have it proxy the request through to our Ruby application server). But let's do it in stages, so the first stage is to hit the root of localhost:

> Note: in the following example you'll need to get the dynamically allocated port number for the nginx container. You can do this by running `docker ps` and extracting the port number from the output provided

```
curl http://$(boot2docker ip):<dynamic_port_number>
```

This `curl` request will result in the following output to stdout...

```
<h1>Welcome</h1>
<p>This is my home page</p>
```

The second stage is to hit the `/test.html` endpoint for localhost and make sure nginx is still serving back our static files correctly:

```
curl http://$(boot2docker ip):<dynamic_port_number>/test.html
```

This `curl` request will result in the following output to stdout...

```
<h1>Hey there!</h1>
<p>Here is my test HTML file</p>
```

Looking good. OK the third and final stage is to now try to hit the `/app/` endpoint for localhost and make sure nginx is proxying the request through to our Ruby backend application server (and also sending the result back again!):

```
curl http://$(boot2docker ip):32785/app/
```

This `curl` request will result in the following output to stdout...

```
<!DOCTYPE html>
<html>
<head>
  <style type="text/css">
  body { text-align:center;font-family:helvetica,arial;font-size:22px;
    color:#888;margin:20px}
  #c {margin:0 auto;width:500px;text-align:left}
  </style>
</head>
<body>
  <h2>Sinatra doesn&rsquo;t know this ditty.</h2>
  <img src='http://app/__sinatra__/404.png'>
  <div id="c">
    Try this:
    <pre>get &#x27;&#x2F;app&#x2F;&#x27; do
  &quot;Hello World&quot;
end
</pre>
  </div>
</body>
</html>
```

Ewww? OK so we can see nginx *is* proxying through to our Ruby application server, but the server doesn't seem to know how to handle the request `/app/` ? Maybe it's something to do with the extra forward slash on the end of the request? If I change it to `/app` instead we'll see what that does:

```
curl http://$(boot2docker ip):32785/app
```

This `curl` request will result in the following output to stdout...

```
<html>
<head><title>301 Moved Permanently</title></head>
<body bgcolor="white">
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx/1.4.6 (Ubuntu)</center>
</body>
</html>
```

Hmm, ok that's not much better. So what's going on here? Well, we're seeing the `301` redirect because Sinatra is redirecting anything without a forward slash. So ok, but what about the "Sinatra doesn't know this ditty" error?

The final solution

The cause of the problem is because we've misconfigured nginx. The solution to the issue requires us to know a *very* subtle detail about how nginx location blocks work, which is:

If you don't put a forward slash `/` at the end of the upstream name, then you'll find nginx passes the request exactly as it was made (i.e. `/app/`) through to the backend service, as apposed to passing it as just `/` .

For example, we had set the proxy to pass to `http://app;` which meant a request to `/app/` was being passed to Sinatra as `/app/` and a request to `/app/foo` was being passed on to Sinatra as `/app/foo` ; and as you can see from the Sinatra application code, we have no such routes defined.

What we should do instead is change the proxy pass value to `http://app/;` (notice the extra forward slash just before the closing semi-colon).

By putting a `/` after the upstream name means it acts more like nginx's `alias` directive which remaps requests for you. If I kept it as `http://app;` , then I would've needed to have added an `/app/` route to my Ruby application. But the additional forward slash remaps `/app/` so it's passed to Sinatra as `/` (we have a route for that) as well as remapping `/app/foo` to just `/foo` (we have a route for that too!).

So with this in mind you'll see in GitHub I've updated the `nginx.conf` to look like the following (with additional comments to clarify the behaviour):

```
user nobody nogroup;
worker_processes auto;          # auto-detect number of logical CPU cores

events {
  worker_connections 512;       # set the max number of simultaneous connectio

}

http {
  upstream app {
    server app:4567;            # app is automatically defined inside /etc/host
  }

  server {
    listen *:80;                # Listen for incoming connections from any inte
    server_name "";             # Don't worry if "Host" HTTP Header is empty or
    root /usr/share/nginx/html; # serve static files from here
```

```
      location /app/ {              # catch any requests that start with /app/
        proxy_pass http://app/;     # proxy requests onto our app server (i.e. a di
                                    #
                                    # NOTE: If you don't put a forward slash / at t
                                    #       then you'll find nginx passes the reque
                                    #       Putting / after the upstream name means
                                    #       If I kept it as http://app; then I woul
      }
    }
  }
```

With this change made, we can now make the following requests successfully:

- `curl http://$(boot2docker ip):<dynamic_port_number>/app/`
- `curl http://$(boot2docker ip):<dynamic_port_number>/app/foo`

Conclusion

This was a bit of a whirlwind run through to getting a simple nginx reverse proxy set-up with Docker. You can obviously swap out the Ruby backend with whatever technology stack is more appropriate (e.g. Node, Clojure, Scala... whatever).

From here, if you're new to nginx (like I am) you can start to experiment with the many other features nginx provides. Enjoy!

Links

- Home
- About Me
- GitHub
- Twitter
- Resume