Exercise 4

coursys.sfu.ca/2021fa-cmpt-470-d1/pages/Exercise4

In this exercise, you will explore the basic ideas of the configuration management setup we will have for “deploying” web apps.

You should **complete either one of** the (option 1) virtual machine or (option 2) container methods.

Option 1: Vagrant VMs

The first task is to get set up so you can run virtual machines (which will soon contain the configuration we define). See the **Vagrant instructions** for this. (You can play with it to make sure it's working but don't worry much about setting up a VM yet: we will do that in the next step.)

Video: Introduction to Vagrant and VMs

Repository Setup

Create a new repository in GitLab for this exercise. Again, be sure to add the instructor and TA as developers on the repo: ggbaker, lirongl, gna17.

Instead of following the provided instructions (to create a README file), **see the project repository instructions**. This will guide you through making a repository that uses the provided Vagrant project template to get some examples and basic directory structure.

Once you have done that, you should be able to get the VM running:

```
vagrant up
```

Sharing with the VM

When you have a Vagrant VM, you can always SSH into it like this:

```
vagrant ssh
```

You will be logging in as the user “vagrant”.

There are a few ways more ways to interact with the VM set up by the provided `Vagrantfile`:

- Port 80 on the VM is forwarded to 8080 on your computer. That means that the VM's web server is reachable as http://localhost:8080/. (Or it will be once there's a web server installed on the VM to reach.)

- The project folder (repository root, with the `Vagrantfile` in it) is shared to the VM at `/home/vagrant/project`. You can see the project files from inside in the VM there.

That means we have the code from the repository inside the VM, will be able to run a web server (and whatever else) on the VM, and then access the web server from the outside and see our code running.

Configuration Management

Now we are at a point that we can start to do configuration management. That is, we can put recipes in our repository that will set up everything on our VM the way it needs to be to get our system running.

Choose a configuration management tool: likely Chef, but you are also welcome to use Puppet or Fabric if you wish. There is some advice on the configuration management page on the choice. You may also want to coordinate with your group members: it probably makes sense to be using the same tools.

Video: Introduction to configuration management

You will find a directory for Chef with a simple configuration example. The provided code: configures the Ubuntu package manager; install the packages `wget` and `ntp`; puts the NTP config file from the repository onto the server at `/etc/ntp.conf`; restart the NTP server so it recognises the new config.

Our Configuration 1: web server

In your repository, **create a directory `webroot` and an HTML file `index.html`** in there. We want this directory to be the root directory of our web server: content of the `index.html` doesn't matter as long as you can recognize it when it's sent by the server.

The things that need to be done in the configuration management code are:

1. Install the Nginx web server (Ubuntu package `nginx`). [Or choose another web server if you want, but my instructions will be Nginx-based.]
2. Set the directory `webroot` as the document root on the web server in the Nginx configuration files.

Web Server Package

You should be able to modify the examples (in the `chef` directory) to **get the web server package (`nginx`) installed** as part of our configuration.

With the VM is running, as you change the configuration code in the repository you can re-run it on the VM:

```
vagrant provision
```

Do that now to get `nginx` installed. You should then be able to access http://localhost:8080/ and see a default page.

Web Server Config

For the web server configuration, you probably want to use the default config file installed with the package (on the VM) as a template to create your own in your repository. The file you need to change in Nginx is `/etc/nginx/sites-available/default`. The easiest way is to have a look at the default. In the VM:

```
cat /etc/nginx/sites-available/default
```

... copy-and-paste that to `cookbooks/baseconfig/files/default/nginx-default`.

Update the `default` file as part of your configuration: the version from your repository needs to be modified as appropriate, and copied to the correct location when the VM is provisioned (with a `cookbook_file` directive in your recipe).

In that file, you'll have to change the `root` directive to point to the full path of your `webroot` directory.

When the configuration files change, **Nginx needs to be reloaded** so it notices. Add that to your configuration recipe as well. If you have problems with Nginx not starting (or restarting), have a look at `/var/log/nginx/error.log`.

If all has gone well, you should now be able to visit http://localhost:8080/ and see the HTML file you created above.

Our Configuration 2: database server

The second thing we have to do is install our database server. That will require:

1. Install the Postgres server (Ubuntu package `postgresql`). [Or choose another DB server if you want, but my instructions will be for Postgres.]
2. Create a database for our system to use, and a non-admin database user who can work with it.

Installing the `postgresql` package can be done as before. Getting the database setup (for Postgres) basically involves running this shell command:

```
echo "CREATE DATABASE mydb; CREATE USER vagrant; GRANT ALL PRIVILEGES ON DATABASE
mydb TO vagrant;" | sudo -u postgres psql
```

We won't really be using the database for this exercise, but you should check that it's working (just after a `vagrant ssh`: this doesn't need to be in the Chef recipe):

```
vagrant ssh # to get in to the VM
psql mydb # start the DB shell
```

You can then create a table, just to make sure it's possible. (`\dt` lists tables in the database.)

```
\dt
CREATE TABLE things (id INT PRIMARY KEY NOT NULL, value INT);
\dt
```

Really Test It

It can be easy to make some quick change within your VM that hasn't been incorporated into your configuration code. The most complete way to test for this is to completely destroy the VM and recreate it using only the configuration:

```
vagrant destroy
vagrant up
```

You should be able to do that, wait, and go to http://localhost:8080/ and see the HTML page you created again. You should also be able to SSH in and access the database as before.

You can do a `vagrant destroy` when you're done with the exercise to clean up the VM that you probably don't need anymore.

Option 2: Docker Containers

The first task is to get set up so you can run containers with Docker and Docker Compose. See the Docker instructions for this.

Repository Setup

Create a new repository in GitLab for this exercise. Again, be sure to add the instructor and TA as developers on the repo: ggbaker, lirongl, gna17.

Instead of following the provided instructions (to create a README file), **see the project repository instructions**. This will guide you through making a repository that uses the provided Docker project template to get some examples and basic directory structure.

Once you have done that, you should be able to get the containers running:

```
docker-compose build && docker-compose up
```

If you visit http://localhost:8080 you should see a default welcome page from the web server install.

Exploring Images

If you want to see what's going on in one of your images, you can start one as a container and run a shell to poke around:

```
docker-compose run web bash
```

That was: **run** in the **web** service (defined in `docker-compose.yml`) the command **bash** (to start a shell).

Once you have a shell, you can look around and see that your project is mounted in `/code` and see what's in the default configuration for the image:

```
ls /code
cat /etc/nginx/conf.d/default.conf
```

Configuration

There are two places we can change what is happening with our collection of containers: any `Dockerfile`s we have, and in the `docker-compose.yml` file itself.

The `docker-compose.yml` controls which services are started, and how. The `Dockerfile`s describe how the images are built and run. Many things in the `Dockerfile` can be overridden in the `docker-compose.yml`.

Our Configuration 1: web server

In your repository, **create a directory `webroot` and an HTML file `index.html`** in there. We want this directory to be the root directory of our web server: content of the `index.html` doesn't matter as long as you can recognize it when it's sent by the server.

The site configuration for Nginx, `/etc/nginx/conf.d/default.conf`, needs to be replaced with one pointing to the correct web root.

Create a new configuration file in your repository with the document root set to `/code/webroot` (possibly by copying the output when you viewed it above and modifying).

Modify the web server `Dockerfile` so the new configuration file is **copied into the web server image** as part of the build process.

If you stop the services (control-C) and bring them up again, at http://localhost:8080 you should see the HTML file you created above.

```
docker-compose build && docker-compose up
```

Our Configuration 2: database server

The Postgres container comes with configuration options to set the database name, user, and password. It will automatically create the database (if it doesn't exist) and set up roles when it starts.

Modify the environment variables for you database containers so the database...

- name (for the default database) is `demodb`;
- user is `demo`;
- password is `secret`.

All of this can be done in the `docker-compose.yml`. The documentation for the image will probably be helpful.

Once you have your services running (`docker-compose up`), in another terminal, let's make sure we can connect to the database. First, check the name of the running database container:

```
docker container ls
```

It will probably be something like `docker_db_1`. Then run `psql` in the running container to get a database shell:

```
docker exec -it docker_db_1 psql -U demo -W demodb
```

... and type the password "secret". There's a lot in that command: the first part (`docker exec -it docker_db_1`) is the docker command that says you want to run a command in a particular container, and the end (`psql -U demo -W demodb`) is the actual command to run.

You can then create a table, just to make sure it's possible. (`\dt` lists tables in the database.)

```
\dt
CREATE TABLE things (id INT PRIMARY KEY NOT NULL, value INT);
\dt
```

Destroy and Rebuild

Let's double-check that if we destroy everything and restart, the services come up correctly:

```
docker-compose down && docker system prune -f
docker-compose build && docker-compose up
```

You should still be able to visit http://localhost:8080/ and connect to the database as before.

In the database, you might notice that your database has disappeared: it was stored in the container that was destroyed (by `docker-compose down`). You can make the data persistent with a volume mounted where Postgres stores its data, but aren't required to for this exercise.

Submission

Make sure you have added all of the files you created for this exercise to the Git repository (look at `git status` and do `git add` as necessary). Commit everything and create a Git tag, as you did last week.

Submit your Git tag through the CourSys activity Exercise 4.

Updated Mon Aug. 30 2021, 07:36 by ggbaker.