

# COSC2759 Tutorial/Lab Week 7

## Goals of this lab

In this tutorial we will be exploring configuration management and deep diving on Ansible to configure a virtual machine in the cloud. We will also be building a docker image and looking at how we can use the image on your local machine.

## Using Ansible to manage a Virtual Machine

The scaffolding for this task can be found at this url: <https://github.com/ashley-mallia/cosc2759-lab-7-solution>. The first section of the lab uses LAB7-CODE-1.

## Configure AWS Credentials

As we will be connecting to AWS, please refresh your credentials as your credentials from week 6 will have expired. If you do not recall how to do this, please refer to week 6 lab sheet.

## Deploy infra

1. Open the folder in your IDE and open the terraform.tfvars file inside the infra directory.

It should look like this:

```
public_key = ""
```

2. We need to add in the public key value - if you do not recall how to do this, please refer to week 6 lab sheet.

If you are not using the default name (id\_rsa) then you will have an extra step when running the ansible playbook in later steps.

3. Save your file and open the infra directory in your terminal. We will now deploy the environment and server we will be working against with the below commands:  
\$> cd infra

```
$> make init
```

```
$> make up
```

If you do not have make installed, run these commands instead:

```
$> cd infra
```

```
$> terraform init
```

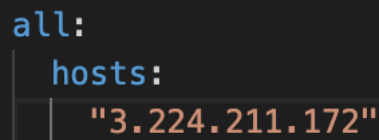
```
$> terraform apply -auto-approve
```

Wait until the server is deployed and verify that it was deployed properly by looking at the EC2 service dashboard in the AWS console.

## Using Ansible to configure Apache

4. Now that your server is up and running, we will start configuring Ansible.

Get the public IP of the EC2 VM we just created from the terraform outputs and add it to the inventory.yml file in the root directory.



```
all:
  hosts:
    "3.224.211.172"
```

5. Verify that we can connect to the server with ansible by running an empty playbook.

```
$> ansible-playbook -i inventory.yml -u ec2-user playbook.yml
```

If you used a public key other than id\_rsa in step 4, you must add the --public-key flag to your command:

```
$> ansible-playbook -i inventory.yml --public-key=<name_of_key> -u ec2-user
playbook.yml
```

The output should look something like this:

```

→ ansible git:(setup) * ansible-playbook -i inventory.yml -u ec2-user playbook.yml

PLAY [Demo of ansible capability and scripting] *****

TASK [Gathering Facts] *****
The authenticity of host '3.224.211.172 (3.224.211.172)' can't be established.
ECDSA key fingerprint is SHA256:ZH1lf4l5nwze8+R2/V9yCUTJKUb5JOCTAVofbNd//Gc.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
[WARNING]: Platform linux on host 3.224.211.172 is using the discovered Python interpreter at /usr/bin/python, but f
See https://docs.ansible.com/ansible/2.9/reference_appendices/interpreter_discovery.html for more information.

ok: [3.224.211.172]

PLAY RECAP *****
3.224.211.172      : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

```

you will get asked to confirm your connection as it is a new host. Answer yes to this.

6. Now that we can connect, let's start working on building out the playbook. Open the playbook.yml file
7. First task we will add to the playbook is to install apache, we will use the "package" module to install the yum package.

Create a new yaml list item under tasks and name it "Ensure apache is installed", make sure you run this install as root. The package name is "httpd" and we want the state to be "present":

```

- name: Ensure apache is installed
  become: true
  package:
    name: httpd
    state: present

```

8. This will install the package, but it won't enable and start the service, so we will do that next. To do that we use the "service" module:

```

- name: Ensure Apache is running
  become: true
  service:
    name: httpd
    state: started
    enabled: yes

```

9. Now we will run the playbook to test the two tasks we have added:

```
$> ansible-playbook -i inventory.yml -u ec2-user playbook.yml
```

output should look like this

```
→ ansible git:(setup) x ansible-playbook -i inventory.yml -u ec2-user playbook.yml

PLAY [Demo of ansible capability and scripting] *****

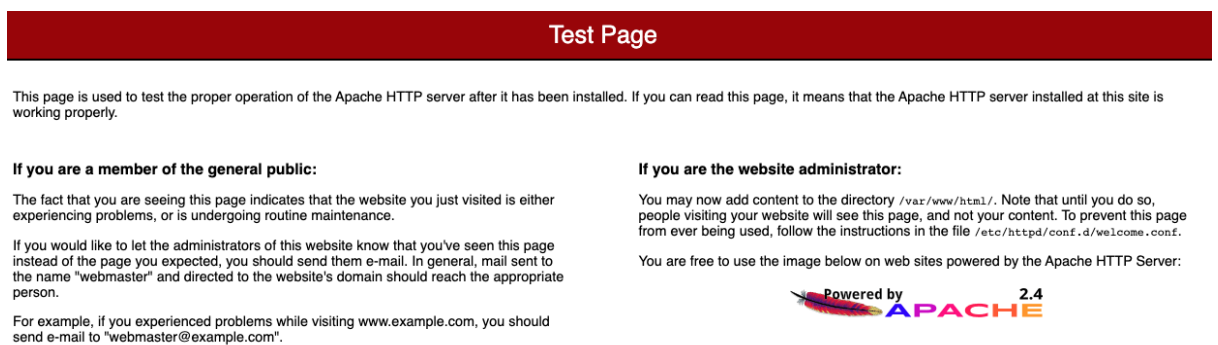
TASK [Follow Link (Cmd + click) on host 3.224.211.172 is using the discovered Python interpreter] *****
[WAR] See https://docs.ansible.com/ansible/2.9/reference_appendices/interpreter_discovery.html f
ok: [3.224.211.172]

TASK [Ensure apache is installed] *****
changed: [3.224.211.172]

TASK [Ensure apache service is running] *****
changed: [3.224.211.172]

PLAY RECAP *****
3.224.211.172 : ok=3 changed=2 unreachable=0 failed=0 skipped=0
```

10. Open the IP in your browser to make sure apache was installed properly. Ensure that the URL is using http:// not https:// - Does anyone know why it has to be http?



11. The test page tells us that we need to add some content to /var/www/html to remove the test page.

To do that we will use the "shell" module run a quick script to write some content to a file.

Add a new step in the play

```
- name: Add test page for website
  become: true
  shell:
    cmd: "echo \"Hello World \" > /var/www/html/index.html"
```

12. Run the playbook again and check the public ip in your browser to verify that we now can't see the test page anymore, and it has been replaced by a page saying

"hello world"

```
$> ansible-playbook -i inventory.yml -u ec2-user playbook.yml
```

```
→ ansible git:(setup) x ansible-playbook -i inventory.yml -u ec2-user playbook.yml

PLAY [Demo of ansible capability and scripting] *****

TASK [Gathering Facts] *****
[WARNING]: Platform linux on host 3.224.211.172 is using the discovered Python interpreter
See https://docs.ansible.com/ansible/2.9/reference_appendices/interpreter_discovery.html
ok: [3.224.211.172]

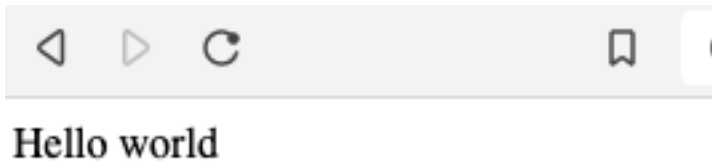
TASK [Ensure apache is installed] *****
ok: [3.224.211.172]

TASK [Ensure apache service is running] *****
ok: [3.224.211.172]

TASK [Add test page for website] *****
changed: [3.224.211.172]

PLAY RECAP *****
3.224.211.172      : ok=4    changed=1    unreachable=0    failed=0    skipped=0
```

Browser:



## Using templates

We now have apache deployed and configured on our server, and we have replaced the test page with a simple "hello world" page.

Next, we will look at using templating to allow us to use variables to add more complicated files with replaceable values.

We often do this to configure applications that we are deploying.

13. Create a new folder to keep our template files in called "templates":

```
$> mkdir templates
```

14. Create a template file to replace the index.html file we created with the shell file:

```
$> touch templates/index.html.tpl
```

15. Add some basic HTML to the template file to allow us to upload this and add values by setting variables in the playbook:

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    Hello {{ name }}!
  </body>
</html>
```

The values wrapped in {{ ... }} are template variables that will get replaced with values that we add to the ansible playbook.

16. Next, we'll add the step in the playbook to apply the template and upload the file to the server. We'll use the template module to do this.

Open playbook.yml again and add this step:

```
- name: Ensure index.html is deployed as per template
  become: true
  vars:
    name: "Tom"
    title: "Tutorial week 7"
  template:
    src: index.html.tpl
    dest: "/var/www/html/index.html"
```

The "vars" group matches the variables we set in the template file.

src: is the location of the template file locally (remember, this should be placed in a templates folder where the playbook.yml is located).

dest: is the destination where the file will be copied to.

17. Run the playbook again, and verify that the page has been updated:

```
$> ansible-playbook -i inventory.yml -u ec2-user playbook.yml
```

```
→ ansible git:(setup) ✖ ansible-playbook -i inventory.yml -u ec2-user playbook.yml

PLAY [Demo of ansible capability and scripting] *****

TASK [Gathering Facts] *****
[WARNING]: Platform linux on host 3.224.211.172 is using the discovered Python interpreter
See https://docs.ansible.com/ansible/2.9/reference_appendices/interpreter_discovery.html

ok: [3.224.211.172]

TASK [Ensure apache is installed] *****
ok: [3.224.211.172]

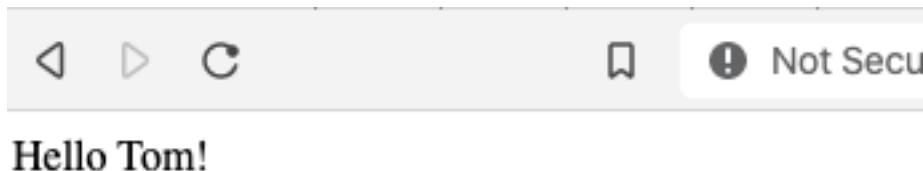
TASK [Ensure apache service is running] *****
ok: [3.224.211.172]

TASK [Add test page for website] *****
changed: [3.224.211.172]

TASK [Ensure index.html is deployed according to template] *****
changed: [3.224.211.172]

PLAY RECAP *****
3.224.211.172      : ok=5    changed=2    unreachable=0    failed=0    skipped=0
```

Browser



18. Since we have a template in place now, we don't need the shell module anymore, so let's delete that step and run the ansible playbook again - nothing should have changed as ansible will only make changes when it is required, e.g., if we change any variables or the template file.

```

→ ansible git:(setup) ✖ ansible-playbook -i inventory.yml -u ec2-user playbook.yml

PLAY [Demo of ansible capability and scripting] *****

TASK [Gathering Facts] *****
[WARNING]: Platform linux on host 3.224.211.172 is using the discovered Python interpreter
See https://docs.ansible.com/ansible/2.9/reference_appendices/interpreter_discovery.html
ok: [3.224.211.172]

TASK [Ensure apache is installed] *****
ok: [3.224.211.172]

TASK [Ensure apache service is running] *****
ok: [3.224.211.172]

TASK [Ensure index.html is deployed according to template] *****
ok: [3.224.211.172]

PLAY RECAP *****
3.224.211.172      : ok=4    changed=0    unreachable=0    failed=0    skipped=0

```

19. Let us change variables – this time passing them in from the command line.

you can use the “-e” flag to pass in variables. This will override any variables set in the playbook.

This is a good way to use the same ansible playbook in multiple environments when you run them in a CI/CD pipeline

\$> ansible-playbook -i inventory.yml -u ec2-user -e name=Pete playbook.yml

```

→ ansible git:(setup) ✖ ansible-playbook -i inventory.yml -u ec2-user -e name=Pete playbook.yml
[WARNING]: Found variable using reserved name: name

PLAY [Demo of ansible capability and scripting] *****

TASK [Gathering Facts] *****
[WARNING]: Platform linux on host 3.224.211.172 is using the discovered Python interpreter at /usr
See https://docs.ansible.com/ansible/2.9/reference_appendices/interpreter_discovery.html for more
ok: [3.224.211.172]

TASK [Ensure apache is installed] *****
ok: [3.224.211.172]

TASK [Ensure apache service is running] *****
ok: [3.224.211.172]

TASK [Ensure index.html is deployed according to template] *****
changed: [3.224.211.172]

PLAY RECAP *****
3.224.211.172      : ok=4    changed=1    unreachable=0    failed=0    skipped=0    rescued=0

```



Browser:



**Hello Pete!**

20. You can also use variables inside the playbook, like the destination path that we place the file.

Add a new section at the top of the playbook and add a variable called "file\_location" that we will use in the template step to tell ansible where to copy the template

```
- name: Demo of ansible capability and scripting
  hosts: all
  vars:
    file_location: "/var/www/html/index.html"
  tasks:
    - name: Ensure apache is installed
```

21. In the template step, we need to update the dest parameter with the variable. This is done like in the template file

```
- name: Ensure index.html is deployed as per template
  become: yes
  template:
    dest: "{{ file_location }}"
    src: index.html.tpl
```

Take note of the double quotation marks around the reference to the variable. This is a quirk of using yaml, and is required for this to work.

22. Run the ansible playbook command again (without overriding the name parameter, so we can verify that it worked)

```
$> ansible-playbook -i inventory.yml -u ec2-user playbook.yml
```

```

→ ansible git:(setup) ✗ ansible-playbook -i inventory.yml -u ec2-user playbook.yml

PLAY [Demo of ansible capability and scripting] *****

TASK [Gathering Facts] *****
[WARNING]: Platform linux on host 3.224.211.172 is using the discovered Python interpreter
See https://docs.ansible.com/ansible/2.9/reference_appendices/interpreter_discovery.html
ok: [3.224.211.172]

TASK [Ensure apache is installed] *****
ok: [3.224.211.172]

TASK [Ensure apache service is running] *****
ok: [3.224.211.172]

TASK [Ensure index.html is deployed according to template] *****
changed: [3.224.211.172]

PLAY RECAP *****
3.224.211.172 : ok=4    changed=1    unreachable=0    failed=0    skipped=

```

Browser



Hello Tom!

## Clean up

Let us clean up the EC2 instance

```
$> cd infra
```

```
$> make down
```

# Building and interacting with a Docker Container

In the first half of this tutorial, we will be looking at docker containers and how we interact with them.

In the directory is LAB7-CODE-2/week7-tute.zip.

Extract the zip file to where you would like to work on the app and change into the directory. Once in there we will set up a git repository to make sure we can restore things if we accidentally do the wrong thing

1. Initialise the git repository and add all the files to the initial commit

```
$> git init  
$> git add .  
$> git commit -m "Initial commit"
```

## Building a docker container

First, we will build a new docker container from an alpine image, and install all the dependencies we need to be able to run an application on the containers

We will create a new docker image based on an existing image that already has nodejs installed: Node:12.2.0-alpine.

2. Create the docker file

```
$> touch Dockerfile
```

3. Open the file in VS code and add the following line to the top of the file

```
FROM node:12.2.0-alpine
```

You now have a fully functional docker file, that you can use to build a new image with. However, you are not installing anything new or making any changes.

4. Next, we will tell docker where to work from by setting the working directory. This will create a new directory, and any files we create, or copy will be in this directory.

Add this line to the file

```
WORKDIR /usr/app
```

This will create a new directory called /usr/app and any commands from now on will be executed from that directory.

5. Since alpine does not have a shell client installed, we will add that next. This will allow us to connect to the container later and debug or review what has been done. We do that early on, so we don't have to reinstall these things over and over. Remember how docker only runs commands that have changes?

Add this line to the docker file:

```
RUN apk add --no-cache bash
```

apk is the package manager on alpine, and --no-cache is saying to don't store any temporary files that will make the image bigger, as we don't need it.

6. Let's build the image and see what happens.

We will tag the image with tute7:latest so we can find it again later:

```
$> docker build -t tute7:latest .
```

First run will take a bit of time to install the bash command, then if you run it again it should be very fast. That is the caching that docker does to help speed up future builds.

7. Review your local images and verify that the image was tagged properly:

```
$> docker images
```

You should see something like this:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
tute7	latest	e6383279af38	13 seconds ago	81.7MB

8. Next, lets install the application into the image, so we can run it inside he container.

Add these lines to the file:

```
COPY package*.json ./

RUN NODE_ENV=production npm install

COPY . .
```

What these lines do, is they copy across just the package.json and the package-lock.json files, then it runs npm install to install all the node\_modules required for the application. And finally, it copies the rest of the files across.

The reason for this approach, is that docker is smart enough to realise when a file on the local drive has changed and will re-run any of the layers (lines in the docker file) that has a change.

We don't want to run npm install every time a js file changes, only when the package files change. So, we are being a bit clever here and minimising the amount of re-build we must do, and in turn speeding up the build.

9. Build the image again:

```
$> docker build -t tute7:latest .
```

You should see an output like this:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
tute7	latest	1328dc1c451e	16 seconds ago	126MB

The image is now a little larger, since we added more files/

10. Let us run the docker image and explore it by connecting using bash:

```
$> docker run -it tute7:latest /bin/bash
```

```
→ tuteapp git:(master) X docker run -it tute7:latest /bin/bash
bash-4.4#
```

check which folder you're in:

```
$> pwd
```

List all the files in the directory:

```
$> ls -al
```

```
bash-4.4# pwd
/usr/app
bash-4.4# ls -al
total 468
drwxr-xr-x  1 root    root      4096 Apr 19 04:05 .
drwxr-xr-x  1 root    root      4096 Feb 12 02:51 ..
drwxr-xr-x  4 root    root      4096 Apr 19 03:28 .qawolf
-rw-r--r--  1 root    root        99 Apr 19 03:28 .sequelizerc
-rw-r--r--  1 root    root     1865 Apr 19 03:28 README.md
-rw-r--r--  1 root    root     1172 Apr 19 03:28 app.js
-rw-r--r--  1 root    root      337 Apr 19 03:28 app.json
drwxr-xr-x  2 root    root      4096 Apr 19 03:28 bin
drwxr-xr-x  2 root    root      4096 Apr 19 03:28 config
drwxr-xr-x  2 root    root      4096 Apr 19 03:28 migrations
drwxr-xr-x  2 root    root      4096 Apr 19 03:28 models
drwxr-xr-x 320 root    root    12288 Apr 19 03:58 node_modules
-rw-r--r--  1 root    root   400333 Apr 19 03:28 package-lock.json
-rw-r--r--  1 root    root     1600 Apr 19 03:32 package.json
drwxr-xr-x  4 root    root      4096 Apr 19 03:28 public
drwxr-xr-x  2 root    root      4096 Apr 19 03:28 routes
drwxr-xr-x  4 root    root      4096 Apr 19 03:28 test
drwxr-xr-x  2 root    root      4096 Apr 19 03:28 views
```

You can see all the files have been copied across/

11. This connection works in the same way as an SSH connection, so when we are finished we exit using the exit command:

```
$> exit
```

12. If we run the docker image with no command right now, it will exit immediately. You can test this with the following command:

```
$> docker run tute7:latest
```

13. Check for running containers with docker ps:

```
$> docker ps
```

```
→ tuteapp git:(master) ✗ docker run tute7:latest
→ tuteapp git:(master) ✗ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
NAMES					

14. Let's fix that by adding a command to the docker file .If you look in the package.json file, one of the scripts listed is "start". We will execute that script to run the app.

Add this line to the docker file and then rebuild the image:

```
CMD [ "npm", "start" ]
```

```
$> docker build -t tute7:latest .
```

15. Run the image again and see what happens now:

```
$> docker run -p 3000:3000 tute7:latest
```

```
→ tuteapp git:(master) ✗ docker run -p 3000:3000 tute7:latest

> rmit-app@0.0.0 start /usr/app
> NODE_ENV=production node ./bin/www

Unhandled rejection SequelizeConnectionRefusedError: connect ECONNREFUSED 127.0.0.1:5432
    at /usr/app/node_modules/sequelize/lib/dialects/postgres/connection-manager.js:121:20
    at Client.<anonymous> (/usr/app/node_modules/pg/lib/client.js:203:5)
    at Connection.emit (events.js:196:13)
    at Socket.<anonymous> (/usr/app/node_modules/pg/lib/connection.js:86:10)
    at Socket.emit (events.js:196:13)
    at emitErrorNT (internal/streams/destroy.js:91:8)
    at emitErrorAndCloseNT (internal/streams/destroy.js:59:3)
    at processTicksAndRejections (internal/process/task_queues.js:84:9)
```

Big error because we don't have a database backend yet.

16. Let's fix that by starting up a database server to run in the background that our application can interact with

```
$> docker run -p 5432:5432 --name db -e POSTGRES_PASSWORD=password -d postgres:10.7
```

If you don't have the mongo already downloaded, it will download it for you and then run it.

-d in the command says to run it in the background

-e in the command is to set an environment variable with the database password

-p in the command is mapping ports between localhost and the container, so we can connect to it

```

→ tuteapp git:(master) x docker run -d -p 27017:27017 mongo:4.0
Unable to find image 'mongo:4.0' locally
4.0: Pulling from library/mongo
fe703b657a32: Pull complete
f9df1fafd224: Pull complete
a645a4b887f9: Pull complete
57db7fe0b522: Pull complete
4f659f7c5a8a: Pull complete
b4b7ff548bb8: Pull complete
67a4fc8b9ac5: Pull complete
4a2afe62084c: Pull complete
665eda6efe5a: Pull complete
42f249afb878: Pull complete
da9817b98263: Pull complete
d52bfe96ec69: Pull complete
5f87759b290c: Pull complete
Digest: sha256:74a364c5a142b1887456289405b4428f6f0aa92c5955b4ff10e477ae2a7795b2
Status: Downloaded newer image for mongo:4.0
5ef003e1de7300c966dd7613bda6df339418fc99f7b970c3971f6dfde3f5eb91

```

17. Verify that the container is running:

```
$> docker ps
```

```

→ tuteapp git:(master) x docker ps

```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
b267dc8b8324	postgres:10.7	"docker-entrypoint.s..."	43 seconds ago	Up 41 seconds	0.0.0.0:5432->5432/tcp	db

18. Next, we need to create the database on the database server. We will execute a command on the database server itself:

```
$> docker exec db psql -U postgres -c "CREATE DATABASE app;"
```

docker exec allows us to do this, we execute the command:

```
psql -U postgres -c "CREATE DATABASE app;"
```

The database is now ready to go,

19. Before we can run the container again, we need to figure out how to configure the application to run against the other container.

Look in the readme file, there is a section on environment variables that we can send through to the container to configure it.

20. Now let's run the app container again with the correct configuration:

```
$> docker run -p 3000:3000 -e DB_HOSTNAME=db -e DB_USERNAME=postgres -e
```



DB\_PASSWORD=password -e DB\_NAME=app --link db tute7:latest

```
+ tuteapp git:(master) * docker run -p 3000:3000 -e DB_HOSTNAME=db -e DB_USERNAME=postgres -e DB_PASSWORD=password -e DB_NAME=app --link db tute7:latest
> rmit-app@0.0.0 start /usr/app
> NODE_ENV=production node ./bin/www
(node:16) DeprecationWarning: Using the automatically created return value from client.query as an event emitter is deprecated and will be removed in pg@7.0.
Please see the upgrade guide at https://node-postgres.com/guides/upgrading
```

We are using the `--link` flag in this command, to make the two containers talk to each other. This is a deprecated flag and will be removed soon. The recommended way is to set up a network and use that.

21. Go to `localhost:3000` in your browser and you can interact with the app.

## Simplifying with docker compose

These commands get long and annoying to type into the terminal every time you need to run a docker container. There is a tool to simplify this, it's called docker compose. Let us set up a docker compose file that stands up both the db and the web app together easily. It will also take care of standing up a network for you, so you don't have to rely on the deprecated `--link` flag.

22. Create a new file called `docker-compose.yml`:

```
$> touch docker-compose.yml
```

23. We are using version 3 of docker compose and we are defining services. There are a few other types that can be deployed with docker compose, but we will focus on services for this tutorial.

Please add the version and the services tag on the top of the file like this:

```
version: "3"
services:
```

24. Next, we will add 2 services, web and db:

```
services:
  web:
  db:
```

25. In the db service, we need to set up the configuration for that database:
- name is db
  - image is postgres:10.7
  - port to pass through is 5432
  - and the environment variable POSTGRES\_PASSWORD should be set to password

We do that like this:

```
container_name: db
image: postgres:10.7
restart: always
ports:
  - '5432:5432'
environment:
  - POSTGRES_PASSWORD=password
```

26. Next, we will set up the web app container with the variables from the run command:

```
image: tute7:latest
ports:
  - 3000:3000
environment:
  - DB_USERNAME=postgres
  - DB_PASSWORD=password
  - DB_HOSTNAME=db
  - DB_NAME=app
```

27. There is a slight issue we need to resolve before we are ready, our app expects the database to already be up and ready to receive connections before it is started.

You can see this in action yourself if you run the docker compose command now:

```
$> docker-compose up -d
```

```
$> docker ps
```

```
→ tuteapp git:(master) ✕ docker-compose up -d
Starting db ... done
Starting tuteapp_web_1 ... done
→ tuteapp git:(master) ✕ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                    NAMES
c457fc929fd2   postgres:10.7 "docker-entrypoint.s..." 7 minutes ago  Up 2 seconds  0.0.0.0:5432->5432/tcp   db
```

Only the database is running

28. To resolve this, we need to override the start command and add a wait, there is a startup.sh script in the app directory that we will use for this.

Update your docker-compose file to look like this:

```
web:
  image: tute7:latest
  depends_on:
    - db
  ports:
    - 3000:3000
  command: "/usr/app/startup.sh"
  environment:
    - NODE_ENV=production
    - DB_USERNAME=postgres
    - DB_PASSWORD=password
    - DB_HOSTNAME=db
    - DB_NAME=app
```

29. Now run docker-compose again

```
$> docker-compose up -d
```

you should see this and be able to access the web app on <http://localhost:3000>

```
→ tuteapp git:(master) ✕ docker-compose up -d
Starting db ... done
Starting tuteapp_web_1 ... done
```

30. To stop the containers, use this command:

```
$> docker-compose down
```

```
→ tuteapp git:(master) ✕ docker-compose down
Stopping tuteapp_web_1 ... done
Stopping db ... done
Removing tuteapp_web_1 ... done
Removing db ... done
Removing network tuteapp_default
```

