

A panel indicates that *DELETE has been initiated...* You may close this message box now.

53. Select the **X** in the top-right corner to close the panel.

© 2021 Amazon Web Services, Inc. and its affiliates. All rights reserved. This work may not be reproduced or redistributed, in whole or in part, without prior written permission from Amazon Web Services, Inc. Commercial copying, lending, or selling is prohibited.  
LAB 8.1

## Lab 8.1: Migrating a Web Application to Docker Containers

### Lab overview and objectives

In this lab, you will migrate a web application to run on Docker containers. The application is installed directly on the guest operating systems (OSs) of two Amazon Elastic Compute Cloud (Amazon EC2) instances. You will migrate the application to run on Docker containers.

After completing this lab, you should be able to:

- Create a Dockerfile.
- Create a Docker image by using a Dockerfile.
- Run a container from a Docker image.
- Interact with and administer your containers.
- Create an Amazon Elastic Container Registry (Amazon ECR) repository.
- Authenticate the Docker client to Amazon ECR.
- Push a Docker image to Amazon ECR.

### Duration

This lab will require approximately **60 minutes** to complete.

### AWS service restrictions

In this lab environment, access to AWS services and service actions might be restricted to the ones that are needed to complete the lab instructions. You might encounter errors if you attempt to access other services or perform actions beyond the ones that are described in this lab.


### Scenario

The café owners have noticed how popular their gourmet coffee offerings have become. Customers cannot seem to get enough of their cappuccinos and lattes. Meanwhile, the café owners have been challenged to consistently source the highest quality coffee beans. Recently, the owners learned that one of their favorite coffee suppliers wants to sell her company. Frank and Martha jumped at the opportunity to buy the company.


The acquired coffee supplier runs an inventory tracking application on an AWS account. Sofia has been tasked to understand how the application works and then create a plan to integrate the application into the café's existing application infrastructure.

In this lab, you will again play the role of Sofia, and you will work to migrate the application to run on containers.

The following diagram shows the architecture that was created for you in AWS at the *beginning* of the lab:

 Coffee website original architecture

By the *end* of this lab, you will have migrated the application and the backend database to run as Docker containers, as shown in the following diagram:

 Coffee website future architecture

You will register these two containers in Amazon ECR to make them available to deploy as needed.

Let's get started!

## Accessing the AWS Management Console

1. At the top of these instructions, choose Start Lab to launch your lab.

A **Start Lab** panel opens, and it displays the lab status.

**Tip:** If you need more time to complete the lab, choose the **Start Lab** button again to restart the timer for the environment.

2. Wait until you see the message *Lab status: ready*, then close the **Start Lab** panel by choosing the **X**.

3. At the top of these instructions, choose AWS.

This opens the AWS Management Console in a new browser tab. The system will automatically log you in.

**Tip:** If a new browser tab does not open, a banner or icon is usually at the top of your browser with a message that your browser is preventing the site from opening pop-up windows. Choose the banner or icon and then choose **Allow pop ups**.

4. Arrange the AWS Management Console tab so that it displays along side these instructions. Ideally, you will be able to see both browser tabs at the same time so that you can follow the lab steps more easily.

**Tip:** If you would like the lab instructions to display across the entire browser window, you can hide the terminal in the browser panel by unchecking the Terminal checkbox in the top right.

## Task 1: Preparing the development environment

In this first task, you will configure your AWS Cloud9 environment to support the work that you will do in the rest of the lab.

5. Connect to the AWS Cloud9 integrated development environment (IDE).

- From the **Services** menu, search for and select **Cloud9**.

Notice the existing IDE, which is named **Cloud9 Instance**.

- For that IDE, choose **Open IDE**.

The AWS Cloud9 IDE loads in a new browser tab.

6. Download and extract the files that you will need for this lab.

- In the same terminal, run the following command:

```
xxxxxxxxxx
```

```
wget https://aws-tc-largeobjects.s3.us-west-2.amazonaws.com/CUR-TF-200-ACCDEV-2-91558/06-lab-containers/code.zip -P /home/ec2-user/environment
```

- The `code.zip` file is downloaded to the AWS Cloud9 instance. The file is listed in the left navigation pane.

- Extract the file:

```
unzip code.zip
```

**Note:** You will use the downloaded and extracted files later in this lab.

7. Run a script that ensures you can get full credit when you choose to submit this lab. It also will upgrade the version of Python and the AWS CLI that are installed on the Cloud9 instance.

- Set permissions on the script so that you can run it, then run it:

```
xxxxxxxxxx
```

```
chmod +x ./resources/setup.sh && ./resources/setup.sh
```

- Verify that the script completed without error.

8. Verify the version of AWS CLI installed.

- In the AWS Cloud9 Bash terminal (at the bottom of the IDE), run the following command:

```
xxxxxxxxxx
aws --version
```

The output should indicate that version 2 is installed.

9. Verify that the SDK for Python is installed.

- Run the following command:

```
xxxxxxxxxx
pip show boto3
```

**Note:** If you see a message about not using the latest version of pip, ignore the message.

## Task 2: Analyzing the existing application infrastructure

In this task, you will analyze the current application infrastructure.

10. Open the existing coffee supplier application in a browser tab.

- Return to the browser tab labeled **Your environments**, and navigate to the EC2 console.
- Choose **Instances**.

Notice that three instances are running.

- One instance is the AWS Cloud9 instance that you used in the previous task.
- The other two instances (MySQLServerNode and AppServerNode) support the application that you will containerize in this lab.
- Choose the **AppServerNode** instance, and copy the **Public IPv4 address** value.
- Open a new browser tab and navigate to the IP address.

The coffee suppliers website displays.

**Note:** The page uses `http://` instead of `https://`. Your browser might indicate that the site is not secure, because it does not have a valid SSL/TLS certificate. You can ignore the warning in this development environment.

11. Test the web application functionality.

- Choose **List of suppliers** and then choose **Add a new supplier**.
- Fill in all of the fields with values. For example:

- **Name:** Nikki Wolf
- **Address:** 100 Main Street
- **City:** Anytown
- **State:** CA
- **Email:** [nwolf@example.com](mailto:nwolf@example.com)
- **Phone:** 4155551212

- Choose **Submit**.

The **All suppliers** page displays and includes the record that you submitted.

- Choose **edit** and change the record (for example, modify the phone number).
- To save the change, choose **Submit**.

Notice that the change was saved in the record.

12. Analyze the web application code.

- A copy of the application code that is installed on the AppServerNode EC2 instance is also available in your AWS Cloud9 environment.
  - Return to the AWS Cloud9 instance browser tab.

- In the file browser in the left navigation pane, expand the **resources** directory, and then expand the **codebase\_partner** directory to see the application code.
- **Optional:** If you are interested to know how the code is configured on the AppServerNode, you can connect to the instance and view the code there as well. To do this, in the terminal next to these instructions, run the following commands, to connect to the EC2 instance and see the files that are installed (replace <public-ip-address> with the actual IPv4 address of the AppServerNode instance).

```

xxxxxxxxxx
ssh -i ~/.ssh/labsuser.pem ubuntu@<public-ip-address>

cd resources/codebase_partner

ls -l

```


- For this lab, it is not necessary for you to understand the details of how the application was built. However, the following details might be of interest to you:
  - The application was built with Express, which is a framework for building web applications.
  - The application runs on port 80 and is coded in node.js.
  - To install the application directly on the guest OS of the AppServerNode Ubuntu Linux EC2 instance, node.js and the node package manager (npm) were installed. Then, the code that you can see in **resources/codebase\_partner** was placed on the server, and the connection to the application's database was configured.

### Task 3: Migrating the application to a Docker container

In this task, you will migrate an application that is installed directly on the guest OS of an Ubuntu Linux EC2 instance to instead run in a Docker container. The Docker container is portable and could run on any OS that has the Docker engine installed.

For convenience, you will run the container on the same EC2 instance that hosts the AWS Cloud9 IDE that you are using. You will use this IDE to build the Docker image and launch the Docker container.

The following diagram shows the migration that you will accomplish in this task:

 Node application migrated to a container

13. Create a working directory for the node application, and move the source code into the new directory.

- In the AWS Cloud9 IDE, to create and navigate to a directory to store your Docker container code, run the following commands:

```

xxxxxxxxxx
mkdir containers
cd containers

```

- To create and navigate to a directory named **node\_app** inside of the **containers** directory, run the following commands:

```

xxxxxxxxxx
mkdir node_app
cd node_app

```

- To move the code base, which you copied earlier, into the new **node\_app** directory, run the following command:

```

xxxxxxxxxx
mv ~/environment/resources/codebase_partner ~/environment/containers/node_app

```

14. Create a Dockerfile.

**Note:** A *Dockerfile* is where you provide instructions to Docker to build an image. A Docker *image* is a template that has instructions to create a *container*.

- To create a new Dockerfile named **Dockerfile** in the **node\_app/codebase\_partner** directory, run the following command:

```

xxxxxxxxxx
cd ~/environment/containers/node_app/codebase_partner
touch Dockerfile

```

- In the left navigation pane, browse to and open the empty Dockerfile that you just created.
- Copy and paste the following code into the Dockerfile:

```

xxxxxxxxxx
FROM node:11-alpine

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

COPY . .

RUN npm install

EXPOSE 3000

CMD ["npm", "run", "start"]

```

- o Save the changes.

**Analysis:** This Dockerfile code specifies that an Alpine Linux distribution with node.js runtime requirements should be used to create the image. The code also specifies that the container should allow network traffic on TCP port 3000. Finally, the code specifies that the application should be run and started when the container launches.

## 15. Build an image from the Dockerfile.

- o In the AWS Cloud9 terminal, run the following command:

```

xxxxxxxxxx
docker build --tag node_app .

```

- o The output is similar to the following:

```

xxxxxxxxxx

Sending build context to Docker daemon  9.007MB

Step 1/7 : FROM node:11-alpine
11-alpine: Pulling from library/node
e7c96db7181b: Pull complete
0119aca44649: Pull complete
40df19605a18: Pull complete
82194b8b4a64: Pull complete
Digest: sha256:8bb56bab197299c8ff820f1a55462890caf08f57ffe3b91f5fa6945a4d505932
Status: Downloaded newer image for node:11-alpine
--> f18da2f58c3d

Step 2/7 : RUN mkdir -p /usr/src/app
--> Running in 768899b8cc6f

Removing intermediate container 768899b8cc6f
--> e3d5cc4cafd7

Step 3/7 : WORKDIR /usr/src/app
--> Running in c16e1d316a6c

Removing intermediate container c16e1d316a6c
--> 9557a073a12b

Step 4/7 : COPY . .
--> 373727287dc7

Step 5/7 : RUN npm install
--> Running in 4ef97681cff6

npm WARN coffee_api@1.0.0 No repository field.
audited 78 packages in 0.862s
found 0 vulnerabilities

Removing intermediate container 4ef97681cff6
--> 847f6f8474c5

Step 6/7 : EXPOSE 3000
--> Running in 39ff9456b6a8

Removing intermediate container 39ff9456b6a8
--> 1e7614a93ae1

Step 7/7 : CMD ["npm", "run", "start"]

```

```

--> Running in ff6310d7bdbd

Removing intermediate container ff6310d7bdbd
--> a5886f101e12

Successfully built a5886f101e12
Successfully tagged node_app:latest

```

**Note:** Ignore any minor warnings in the output.

16. Verify that the Docker image was created.

- o To list the Docker images that your Docker client is aware of, run the following command:

```

xxxxxxxxxx
docker images

```

- o The output is similar to the following:

```

x
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
node_app     latest    39501e06b5e9   6 seconds ago  84.5MB
node         11-alpine f18da2f58c3d   4 years ago    75.5MB

```

- o Notice that the **node\_app** line item was created only a few seconds ago.

17. Create and run a Docker container based on the Docker image.

- o To create and run a Docker container from the image, run the following command:

```

x
docker run -d --name node_app_1 -p 3000:3000 node_app

```

**Analysis:** This command launches a container with the name **node\_app\_1**, using the **node\_app** image that you created as the template. The **-d** argument specifies that it should run in the background and print the container ID. The **-p** specifies to publish container port 3000 to the host (AWS Cloud9 instance) port 3000.

- o The terminal returns the container ID. It is a long string of letters and numbers.
- o To view the Docker containers that are currently running on the host, run the following command:

```

x
docker container ls

```

18. Verify that the node application is now running in the container.

- o To check that the container is working on the correct port, run the following command:

```

x
curl http://localhost:3000

```

- o The webpage looks similar to the following:

x

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="/css/bootstrap.min.css">
  <link rel="stylesheet" href="/css/base.css">
  <title>Coffee suppliers</title>
</head>
<body>

<div class="container">
  <nav class="navbar navbar-expand-lg navbar-dark bg-dark">

```

```


<div><a class="navbar-brand page-title" href="/supplier">Coffee suppliers</a></div>
<div class="collapse navbar-collapse" id="navbarSupportedContent">
  <ul class="navbar-nav mr-auto">
    <li class="nav-item active">
      <a class="nav-link" href="/">Home</a>
      <a class="nav-link" href="/suppliers">Suppliers list</a>
    </li>
  </ul>
</div>
</nav>    <div class="container">
  <h1>Welcome</h1>
  <p>Use this app to keep track of your coffee suppliers</p>
  <p><a href="/suppliers">List of suppliers</a></p>
</div>
</div>
<script src="/js/jquery-3.6.0.min.js"></script>
<script src="/js/bootstrap.min.js"></script>
</body>
</html>

```

**Analysis:** This demonstrates that the application is running and available on TCP port 3000. However, you want to interact with this as a website, and you don't yet know if the application is able to connect to the database.

19. Adjust the security group of the AWS Cloud9 EC2 instance to allow network traffic on port 3000 from your computer.

**Note:** Because you are using the AWS Cloud9 EC2 instance to run the container, you must open TCP port 3000 for inbound traffic.

- Return to the AWS Management Console browser tab, and navigate to the EC2 console.
- Locate and select the **aws-cloud9-Cloud9-Instance** instance.
- Choose the **Security** tab, and choose the **aws-cloud9-Cloud9-Instance** security group hyperlink.
- Choose the **Inbound rules** tab, and choose **Edit inbound rules**.
- Choose **Add rule** and configure the following:
  - Type: **Custom TCP**
  - Port range: **3000**
  - Source: **My IP**
- Choose **Save rules**.

20. Access the web interface of the application, which is now running in a container.

- In the EC2 console, choose **Instances** and choose the **aws-cloud9-Cloud9-Instance** instance.
- On the **Details** tab, copy the **Public IPv4 address** value.
- Open a new browser tab. Paste the IP address into the address bar, and add **:3000** at the end of the address.

The web application loads in the browser. You have seen this page before; however, you previously accessed the application that was running directly on the AppServerNode EC2 instance guest OS. This time you accessed the application running in a container on the AWS Cloud9 EC2 instance's Docker hypervisor.

## Summary of how you have used Docker so far

You just completed a series of steps with Docker. The following diagram summarizes what you have accomplished with Docker so far.

 Docker process

- You copied the code base into a directory, which acted as your build area [a]. You also created a Dockerfile that provided instructions for how to create a Docker image. That Dockerfile specified a **FROM** instruction that identified a starter image to use.
- You then ran the **docker build** command [b]. Docker read the Dockerfile and requested the starter image from an image repository [c]. The image repository returned the starter image file [d].

- The `docker build` command then finished building the image according to the instructions in the Dockerfile, which resulted in the Docker image [e].
- Finally, you ran the `docker run` command [f] to run a Docker container [g].

## 21. Analyze the database connection issue.

- In the coffee suppliers application, choose **List of suppliers**.

You see an error stating that there was a problem retrieving the list of suppliers.

**Analysis:** This is because the `node_app_1` container is having trouble reaching the MySQL database, which is running on the EC2 instance named **MysqlServerNode**.

- Return to the AWS Cloud9 IDE browser tab.
- Open the `config.js` file in the `containers/node_app/codebase_partner/app/config/` directory.

The top of the file contains the following code:

```
xxxxxxxxxx
let config = {
  APP_DB_HOST: "3.82.161.206",
  APP_DB_USER: "nodeapp",
  APP_DB_PASSWORD: "coffee",
  APP_DB_NAME: "COFFEE"
}
```

**Analysis:** The application code checks for an environmental variable to learn how to connect to the MySQL database. As you see in the code, the settings include a hardcoded IP address for the location of the MySQL database host.

- However, the application code also contains the following logic:

```
xxxxxxxxxx
Object.keys(config).forEach(key => {
  if(process.env[key] === undefined){
    console.log(`[NOTICE] Value for key '${key}' not found in ENV, using default value. See app/config/config.js`)
  } else {
    config[key] = process.env[key]
  }
});
```

This checks for the existence of an environmental variable. If one is found, that value overrides the placeholder (hardcoded) `APP_DB_HOST` address.

When you visited the web application earlier—the version that was directly installed on the EC2 instance guest OS—the node instance passed an environment variable (the IPv4 address of the **MysqlServerNode** EC2 instance) to the application.

However, you did not launch your `node_app_1` container with that environment variable. Therefore, the node application defaulted to the hardcoded 3.82.161.206 IP address, which does not match the IP address of the MySQL instance.

- Establish a terminal connection to the container to observe the settings.
  - To find the container ID, run the following command:

```
xxxxxxxxxx
docker ps
```

- To connect your terminal to the container, run the following commands, one at a time. Replace `<container-id>` with the actual container ID value that you just retrieved:

```
xxxxxxxxxx
docker exec -ti <container-id> sh
whoami
```

Your terminal is now connected to the container as the root user.

- To observe the environment variables that are present in the node user's environment, run the following commands:

```
xxxxxxxxxx
su node
```



```
env
```

Notice that the **APP\_DB\_HOST** variable is not present.

- To disconnect from the container, run the following commands:

```
xxxxxxxxxx
exit
exit
```

The first `exit` command makes you the root user again. The second command disconnects you from the container.

## 22. Stop and remove the container that has the database connectivity issue.

- To get the ID of the running container, run the following command:

```
xxxxxxxxxx
docker ps
```

Notice the name of the application that is returned in the **NAMES** column.

- To stop and remove the container, run the following command:

```
xxxxxxxxxx
docker stop node_app_1 && docker rm node_app_1
```

- To verify that the application is no longer running, run the following `curl` command:

```
xxxxxxxxxx
curl http://localhost:3000
```

The output indicates a failure to connect to the application (*Connection refused*).


**Tip:** If you refresh the web application in the browser (the version that is running as a container on the AWS Cloud9 instance), you will find that the application no longer loads.

## 23. Launch a new container. This time, you will pass an environment variable to tell the node application the correct location of the database.

- Return to the EC2 console, and copy the **Public IPv4 address** value of the **MysqlServerNode** EC2 instance.
- Return to the AWS Cloud9 terminal.
- To run the application in a container and pass an environment variable to specify the database location, run the following command. Replace `<ip-address>` with the actual public IPv4 address of the **MysqlServerNode** EC2 instance:

```
xxxxxxxxxx
docker run -d --name node_app_1 -p 3000:3000 -e APP_DB_HOST="<ip-address>" node_app
```

When you pass in the network location of the database as an environment variable, you give the node application the information that it needs to establish network connectivity to the database, as illustrated in the following diagram:

 Node application network connectivity to database

**Optional:** To check the environment variables of the new container, run the `docker exec` command, which you used previously. The container now has an **APP\_DB\_HOST** variable.

## 24. Verify that the database connection is now working.


- Try to access the web application again.
  - If you still have the page open, refresh the browser tab. Otherwise, to navigate to the application in a new browser tab, go to `http://<cloud9-public-ip>:3000` (replace `<cloud9-public-ip>` with the actual public IPv4 address of your AWS Cloud9 instance).
- The application is working. Choose **List of suppliers** to go to the `http://<cloud9-public-ip>:3000/suppliers` page.
- The page displays the supplier entry that you created earlier. This indicates that your container is connecting to the **MysqlServerNode** EC2 instance where that data is stored.

Congratulations! You have successfully migrated the node application to a container. Also, the application container (**node-app\_1**) is now able to successfully establish a network connection with the MySQL database, which is still running on an EC2 instance.

## Task 4: Migrating the MySQL database to a Docker container

In this task, you will work to migrate the MySQL database to a container as well. To accomplish this task, you will dump the latest data that is stored in the database and use that to seed a new MySQL database running in a new Docker container.

The following diagram shows the migration that you will accomplish in this task:

 Database migrated to a container

25. Create a mysqldump file from the data that is currently in the MySQL database.

- Return to the AWS Cloud9 IDE, and close any file tabs that are open in the text editor.
- Choose **File > New File** and then paste the following code into the new file:

```
xxxxxxxxxx
mysqldump -P 3306 -h <mysql-host-ip-address> -u nodeapp -p --databases COFFEE > ../../my_sql.sql
```

- Next, go to the EC2 console and copy the **Public IPv4 address** value of the **MysqlServerNode** instance.
- Return to the text file in AWS Cloud9 and replace <mysql-host-ip-address> in the code with the IP address that you copied.
- In the terminal, to ensure that you are in the correct directory, run the following command:

```
xxxxxxxxxx
cd /home/ec2-user/environment/containers/node_app/codebase_partner
```


- Finally, copy the command that you created in the text editor into the terminal and run the command.

Your command will look similar to the following example, but your IP address will be different:

```
xxxxxxxxxx
mysqldump -P 3306 -h 100.27.45.2 -u nodeapp -p --databases COFFEE > ../../my_sql.sql
```

- The mysqldump utility prompts you for a password. Enter the following password:

```
xxxxxxxxxx
coffee
```

If successful, the terminal does not show any output. However, in the left navigation panel, notice that the mysql file icon in AWS Cloud9 file now appears in the **containers** directory.

**Tip:** To see the new file, you might need to choose the settings icon in the upper-right corner of the file tree panel and then choose **Refresh File Tree**.

26. Open the mysqldump file and observe the contents.

- Open the **my\_sql.sql** file in the AWS Cloud9 editor.
- Scroll through the contents of the file.
  - Notice that it will create a database named **COFFEE** and a table named **suppliers**.
  - Also, because you added a record using the application web interface earlier in this lab, the script inserts that record into the **suppliers** table.
- Make a small change to one of the values in the file.
  - Locate the line that starts with **INSERT INTO**. It will appear around line 51.
  - Modify the address that you entered. For example, if the address has a street named **Main** change it to **Container**. **Note:** This change will help you later in the lab when you want to confirm that you are connected to the new database running on a container, and not the old database.
  - Choose **File > Save** to the change.

27. In the terminal, to create a directory to store your mysql container code and navigate into the directory, run the following commands:

```
xxxxxxxxxx
cd /home/ec2-user/environment/containers
mkdir mysql
cd mysql
```

28. Create a Dockerfile.

- To create a new Dockerfile, run the following command:

```
xxxxxxxxxx
touch Dockerfile
```

- To move the sqldump file into the new **mysql** directory, run the following command:

```
xxxxxxxxxx
mv ../my_sql.sql .
```

- Open the empty Dockerfile (in **containers/mysql/**) and then copy and paste the following code into the file:

```
xxxxxxxxxx
FROM mysql:8.0.23
COPY ../my_sql.sql /
EXPOSE 3306
```

- Save the changes.

**Analysis:** This Dockerfile code specifies that a new Docker image should be created by starting with an existing mysql Docker image. Then, the sqldump file, which you created in a previous step, is copied to the container. The code also specifies that the container should allow network traffic on TCP port 3306, which is the standard MySQL port for network communication.

29. Attempt to free up some disk space on the AWS Cloud9 EC2 instance by removing unneeded files.

- Run the following command:

```
xxxxxxxxxx
docker rmi -f $(docker image ls -a -q)
```

**Note:** You can safely ignore any *Error response from daemon* messages that display.

- Finally, run the following command:

```
xxxxxxxxxx
sudo docker image prune -f && sudo docker container prune -f
```

**Note:** In some cases, the total reclaimed space might be zero; however, you might see that some unneeded containers were deleted.

30. To build an image from the Dockerfile, run the following command:

```
xxxxxxxxxx
docker build --tag mysql_server .
```

The output is similar to the following:

```
xxxxxxxxxx
Sending build context to Docker daemon  5.12kB
Step 1/3 : FROM mysql:8.0.23
8.0.23: Pulling from library/mysql
f7ec5a41d630: Pull complete
9444bb562699: Pull complete
6a4207b96940: Pull complete
181cefd361ce: Pull complete
8a2090759d8a: Pull complete
15f235e0d7ee: Pull complete
d870539cd9db: Pull complete
5726073179b6: Pull complete
eadfac8b2520: Pull complete
f5936a8c3f2b: Pull complete
cca8ee89e625: Pull complete
```

```

6c79df02586a: Pull complete
Digest: sha256:6e0014cdd88092545557dee5e9eb7e1a3c84c9a14ad2418d5f2231e930967a38
Status: Downloaded newer image for mysql:8.0.23
---> cbe8815cbea8
Step 2/3 : COPY ./my_sql.sql /
---> b71b3be6d378
Step 3/3 : EXPOSE 3306
---> Running in 975ff38ce91c
Removing intermediate container 975ff38ce91c
---> 13c22244afbb
Successfully built 13c22244afbb
Successfully tagged mysql_server:latest

```

### 31. Verify that the Docker image was created.

- To list the Docker images that your Docker client is aware of, run the following command:

```

xxxxxxxxxx
docker images

```

- The output is similar to the following:

```

xxxxxxxxxx
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
mysql_server        latest       13c22244afbb     About a minute ago 546MB
node_app            latest       6148dceae9d0     3 hours ago      82.7MB
mysql               8.0.23      cbe8815cbea8     3 weeks ago      546MB
node                11-alpine   f18da2f58c3d     23 months ago    75.5MB

```

- Notice the **mysql\_server** line item, which was created only a few minutes ago.

### 32. Create and run a Docker container based on the Docker image.

- To create and run a Docker container from the image, run the following command:

```

xxxxxxxxxx
docker run --name mysql_1 -p 3306:3306 -e MYSQL_ROOT_PASSWORD=rootpw -d mysql_server

```

**Analysis:** This command launches a container with the name **mysql\_1**, using the **mysql\_server** image that you created as the template. The **-e** parameter passes an environment variable.

- The terminal returns the container ID for the container.
- To view the Docker containers that are currently running on the host, run the following command:

```

xxxxxxxxxx
docker container ls

```

Two containers are now running. One hosts the node application, and the other hosts the MySQL database.

### 33. Import the data into the MySQL database and define a database user.

- Run the following command:

⚠ Note that space is *not* included between **-p** and **rootpw** in the command.

```

xxxxxxxxxx
docker exec -i mysql_1 mysql -u root -prootpw < my_sql.sql

```

Ignore the warning about using a password on the command line interface being insecure.

- To create a database user for the node application to use, run the following command:


```
xxxxxxxxxx
```

```
docker exec -i mysql_1 mysql -u root -prootpw -e "CREATE USER 'nodeapp' IDENTIFIED WITH mysql_native_password BY 'coffee'; GRANT all privilege
```

## Task 5: Testing the MySQL container with the node application

Recall that in a previous task you connected to the node application running in the container, but it was connected to the MySQL database that was running on the MysqlServerNode EC2 instance.

In this task, you will update the node application running in the container to point to the MySQL database running in the container. The following diagram shows the migration that you will accomplish in this task:

 MySQL container network connectivity

34. To stop and remove the node application server container, run the following command:

```
xxxxxxxxxx
```

```
docker stop node_app_1 && docker rm node_app_1
```

35. Discover the network connectivity information.

- To find the IPv4 address of the **mysql\_1** container on the network, run the following command:

```
xxxxxxxxxx
```

```
docker inspect network bridge
```

The following example output shows only a portion of the output:

```
xxxxxxxxxx
```

```
"Containers": {  
  "c349bf43b684ab1224d41eb898ce916c48572af11f1ee69063302c17d83b5e92": {  
    "Name": "mysql_1",  
    "EndpointID": "a026ac254a9dd6d89b38cc1590d99b878c9b7173a3c1adbd47302e3c3c26a115",  
    "MacAddress": "02:42:ac:11:00:02",  
    "IPv4Address": "172.17.0.2/16",  
    "IPv6Address": ""  
  }  
},
```

In the example output, the IPv4 address used by the **mysql\_1** container is 172.17.0.2.

⚠ The IP address used by *your* **mysql\_1** container is different.

36. Start a new node application Docker container.

- In this step, you run the Docker command to start a new container from the **node\_app** Docker image. However, this time you pass the **APP\_DB\_HOST** environment variable to the container environment.
- Run the following command. Replace `<ip-address>` with the actual IPv4 address value that you just discovered. You do *not* need to surround the IP address in quotes.

```
xxxxxxxxxx
```

```
docker run -d --name node_app_1 -p 3000:3000 -e APP_DB_HOST=<ip-address> node_app
```

The container starts as it did previously.

37. To verify that both containers are running again, run the following command:

```
xxxxxxxxxx
```

```
docker ps
```

38. Test the application.

- Open the web application that is running as a container on the AWS Cloud9 instance.

The address for the web application is: `http://<cloud9-public-ip-address>:3000`

- In the coffee suppliers application, choose **List of suppliers** to verify that the database is connected.

If you see the supplier entry that you created previously, and the entry contains the change that you made to the street name (for example, Container Street), then that is confirmation that you have successfully connected the `node_app` running in the container to the database running in the other container.

One of the important benefits of running an application on containers is the portability and scalability that doing so provides.

Sofia has now proven that she can launch functional containers from each of the two Docker images that she created, she is ready to move this solution into production.

**NOTE:** to get full credit when you submit your lab, leave the two Docker containers running.

## Task 6: Adding the Docker images to Amazon ECR

In this final task in the lab, you will add the Docker images that you created to an Amazon Elastic Container Registry (Amazon ECR) repository.

39. Authorize your Docker client to connect to the Amazon ECR service.

- Discover your AWS account ID.
  - In the AWS Management Console, in the upper-right corner, choose your user name. Your user name begins with **voclab/user**.
  - Copy the **My Account** value from the menu. This is your AWS account ID.
- Next, return to the AWS Cloud9 terminal.
- To authorize your AWS Cloud9 Docker client, run the following command. Replace `<account-id>` with the actual account ID that you just found:

```
xxxxxxxxxx
aws ecr get-login-password \
--region us-east-1 | docker login --username AWS \
--password-stdin <account-id>.dkr.ecr.us-east-1.amazonaws.com
```

A message indicates that the login succeeded.

40. To create the repository, run the following command:

```
xxxxxxxxxx
aws ecr create-repository --repository-name node-app
```

The response data is in JSON format and includes a **repositoryArn** value. This is the URI that you would use to reference your image for future deployments.

The response also includes a **registryId**, which you will use in a moment.

41. Tag the Docker image.

In this step, you will tag the image with your unique **registryId** value to make it easier to manage and keep track of this image.

- Run the following command. Replace `<registry-id>` with your actual registry ID number.

```
xxxxxxxxxx
docker tag node_app:latest <registry-id>.dkr.ecr.us-east-1.amazonaws.com/node-app:latest
```

The command does not provide a response.

- To verify that the tag was applied, run the following command:

```
xxxxxxxxxx
docker images
```

- This time, notice that the **latest** tag was applied and the image name includes the remote repository name where you intend to store it. The following image provides an example of the output:

 docker images command output example

42. Push the Docker image to the Amazon ECR repository.

- To push your image to Amazon ECR, run the following command. Replace <registry-id> with your actual registry ID number:

```
xxxxxxxxxx

docker push <registry-id>.dkr.ecr.us-east-1.amazonaws.com/node-app:latest
```

- The output is similar to the following:

```
xxxxxxxxxx

The push refers to repository [642015801240.dkr.ecr.us-east-1.amazonaws.com/node-app]

006e0ec54dba: Pushed
59762f95cb06: Pushed
22736f780b31: Pushed
d81d715330b7: Pushed
1dc7f3bb09a4: Pushed
dcaceb729824: Pushed
f1b5933fe4b5: Pushed

latest: digest: sha256:f75b60adddb8d6343b9dff690533a1cd1fbb34ccce6f861e84c857ba7a27b77d size: 1783
```

43. To confirm that the **node-app** image is now stored in Amazon ECR, run the following `aws ecr list-images` command:

```
xxxxxxxxxx

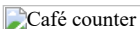
aws ecr list-images --repository-name node-app
```

The command returns information about the image that you just uploaded. The output is similar to the following:

```
xxxxxxxxxx

{
  "imageIds": [
    {
      "imageDigest": "sha256:f75b60adddb8d6343b9dff690533a1cd1fbb34ccce6f861e84c857ba7a27b77d",
      "imageTag": "latest"
    }
  ]
}
```

## Update from the café

 Café counter

Sofia is satisfied that she has made progress. She successfully containerized the web application from the coffee supplier company that the café owners purchased. The application had previously been installed directly on the guest OS of an EC2 instance. Now the application is containerized, and she has more flexibility to deploy the application and scale it cost effectively.

She also successfully containerized the backend database that the application uses. Finally, she was able to successfully register the Docker image in Amazon ECR to launch the coffee supplier application in the future.

In the next lab, Sofia will use the Docker image that she just stored in Amazon ECR to deploy the coffee supplier application using AWS Elastic Beanstalk. Although Sofia also containerized the MySQL database, she has decided not to push that image to Amazon ECR. She spoke with one of the AWS consultants who came in for coffee today, and the consultant convinced her that it makes more sense to use the Amazon Relational Database Service (Amazon RDS) service to host the coffee supplier database, instead of hosting it on a container. The next lab will discuss the reasons for this, but for now Sofia is quite content that she was able to containerize the coffee supplier application. She looks forward to deploying it in the next lab!

## Submitting your work

44. At the top of these instructions, choose Submit to record your progress and when prompted, choose **Yes**.

**Tip:** If you previously hid the terminal in the browser panel, expose it again by selecting the **Terminal** checkbox. This action will ensure that the lab instructions remain visible after you choose **Submit**.

45. If the results don't display after a couple of minutes, return to the top of these instructions and choose Grades

**Tip:** You can submit your work multiple times. After you change your work, choose **Submit** again. Your last submission is what will be recorded for this lab.

46. To find detailed feedback on your work, choose Details followed by **View Submission Report**.

## Lab complete

Congratulations! You have completed the lab.

47. Choose End Lab at the top of this page, and then select Yes to confirm that you want to end the lab.

A panel indicates that *DELETE has been initiated... You may close this message box now*.

48. Select the **X** in the top-right corner to close the panel.

© 2021 Amazon Web Services, Inc. and its affiliates. All rights reserved. This work may not be reproduced or redistributed, in whole or in part, without prior written permission from Amazon Web Services, Inc. Commercial copying, lending, or selling is prohibited.  
LAB 8.2

[version\_1.0.45]

# Lab 8.2: Running Containers on a Managed Service

## Lab overview and objectives

In a previous lab, you migrated an application that ran on Amazon Elastic Compute Cloud (Amazon EC2) instances to run on Docker containers. In this lab, you will deploy the application using two managed cloud services. You will deploy the database tier using Amazon Aurora Serverless and the web tier using AWS Elastic Beanstalk.

After completing this lab, you should be able to:

- Create a new Amazon Relational Database Service (Amazon RDS) instance using the AWS Management Console
- Launch a Docker container on AWS Cloud9 using an image pulled from Amazon Elastic Container Registry (Amazon ECR)
- Configure and test the containerized application connection to Aurora Serverless
- Use the Amazon RDS query editor to create database objects and load data
- Launch the default Elastic Beanstalk application
- Update the Elastic Beanstalk application to run your node application and communicate with Amazon RDS
- Configure an Amazon API Gateway endpoint to forward calls to the Elastic Beanstalk URL