Deploying to AWS Elastic Container Service

Deploying your application to the public cloud can be a burden. Also, the public clouds come with a steep learning curve. In this module we'll take a look deploying to AWS's Elastic Container Service (ECS) using the Docker CLI integration with ECS.

Step 1 - Create an AWS account

If you already have an AWS account, please feel free to skip this section.

Navigate to the AWS signup page and enter your details. Click "continue" and follow the steps presented to create an account.

Step 2 - Create a Docker Context

A Docker Context is a configuration that tells Docker where and how to run containers. A context contains all of the endpoint and security information required to manage a different environment that can run containers. The docker context command makes it easy to configure these contexts and switch between them.

Access key

We will need an access key to create our context for ecs. Login to the aws console and follow the steps below to create an access key.

1. Navigate to the IAM service.
2. Click on "Users' in the left panel.
3. Click the "Security credentials" tab.
4. Click the "Create access key" button.
5. Save the csv file to your computer.

Create a Docker Context

Now let's create our context and point it at ECS. Open your terminal and run the following command.

```
$ docker context ls
NAME                TYPE                DESCRIPTION
DOCKER ENDPOINT               KUBERNETES ENDPOINT
ORCHESTRATOR
```

```
default *           moby                Current DOCKER_HOST based
configuration   unix:///var/run/docker.sock
https://kubernetes.docker.internal:6443 (default)   swarm
```

The `docker context ls` command prints out a list of contexts currently configured on your local machine. You'll notice the `default` context has a `*` beside it. This denotes that it is the current context that all Docker commands will use and points to the local Docker engine.

Now lets create a context that we can use to target ECS. To do this, we'll use the `docker context create` command. Let's take a look at the help for this command.

```
$ docker context create --help

Create a new context

Create docker engine context:
$ docker context create CONTEXT [flags]


Create Azure Container Instances context:
$ docker context create aci CONTEXT [flags]
(see docker context create aci --help)


Create Amazon ECS context:
$ docker context create ecs CONTEXT [flags]
(see docker context create ecs --help)
...
```

I've truncated the output to the most relevant point. That is creating an Amazon ECS context. Let's take a look at the help for the `docker context create ecs` command.

```
$ docker context create ecs --help
Create a context for Amazon ECS

Usage:
  docker context create ecs CONTEXT [flags]

Flags:
      --description string   Description of the context
  -h, --help                 Help for ecs
      --local-simulation     Create context for ECS local simulation
endpoints
      --profile string       Profile
```

```
      --region string        Region

Global Flags:
      --config DIRECTORY   Location of the client config files
DIRECTORY (default "/Users/peter/.docker")
  -c, --context string     context
  -D, --debug              Enable debug output in the logs
  -H, --host string        Daemon socket(s) to connect to
```

As you can see, the `docker context create ecs` command takes a `CONTEXT` as parameter. The `CONTEXT` parameter will be used to name the context. Let's run the basic command without passing any flags.

```
$ docker context create ecs ecs
? Select AWS Profile  [Use arrows to move, type to filter]
> new profile
  default
  workshop
```

The command recognizes that we did not provide a profile and will ask us to select one to use or create a new one. Let's create a new one. Move the arrow (`>`) so that it is pointing to the `new profile` option and press enter.

We'll use the name "workshop". Enter that as your profile name.

```
$ docker context create ecs ecs
? Select AWS Profile new profile
? profile name workshop
```

Now we are asked if we want to provide AWS credentials. If you do not want to provide credentials now, you can add them to the `~/.aws/credentials` later. Let's enter them now. Open the csv file that you downloaded earlier and use the access key and secret below.

```
$ docker context create ecs ecs
? Select AWS Profile new profile
? profile name workshop
? Enter AWS credentials yes
? AWS Access Key ID XXXXXXXXXXXXXXXXXXX
? Enter AWS Secret Access Key
****************************************
```

Enter `us-east-1` for the region.

```
$ docker context create ecs ecs
? Select AWS Profile new profile
? profile name workshop
? Enter AWS credentials yes
? AWS Access Key ID XXXXXXXXXXXXXXXXXXX
? Enter AWS Secret Access Key
************************************
? Region us-east-1
Successfully created ecs context "ecs"
```

Let's print out the list of contexts.

```
$ docker context ls
NAME                TYPE                DESCRIPTION
DOCKER ENDPOINT               KUBERNETES ENDPOINT
ORCHESTRATOR
default *           moby                Current DOCKER_HOST based
configuration   unix:///var/run/docker.sock
https://kubernetes.docker.internal:6443 (default)   swarm
ecs                 ecs                 (us-east-1)
```

BOOOM! There we go. We now have a docker context pointing to the AWS ECS service in us-east-1.

Step 3 - Deploy to ECS

Now that we have a context pointing to ECS, we'll use it to deploy our pytorch application. To do that, we'll create a compose file to describe our service. A compose file allows us to describe our applications and how they should be started and run.

Create a text file named `docker-compose-ecs.yml` and add the following yaml to it. Make sure you replace the <dockerid> with your Docker ID.

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

```
version: "3.8"

services:
  dockerml:
    image: <dockerid>/docker-ml:full
```

As you can see at the top of the compose file, we are using version 3.8 of the compose spec. Next we create a section to configure our services. Then we created a service named

"dockerml" and then under the "dockerml" section we identify the image to use when starting the container.

Before we run our application in ECS, we need to push our image to Docker Hub and set the context to use the one we created above. Follow the steps below.

1. Login to docker hub from the terminal.
   `$ docker login --username`
2. Tag your image with your docker id
   `$ docker tag docker-ml:full pmckee/docker-ml:full`
3. Push your image to hub
   `$ docker push <dockerid>/docker-ml:full`
4. Use the “workshop” context
   `$ docker context use ecs`

To deploy your compose application, we'll use the `docker compose up` command. This will read the `docker-compose-ecs.yml` file and create and start our containers on ECS. Since we are not using the default file name (docker-compose.yml) for our compose file, we'll pass `docker-compose-ecs.yml` to the `--file` flag so Docker knows which compose file to use.

```
$ docker compose up --file docker-compose-ecs.yml
[+] Running 10/10
 ⠿ dockerml                        CREATE_COMPLETE
 ⠿ CloudMap                        CREATE_COMPLETE
 ⠿ DefaultNetwork                  CREATE_COMPLETE
 ⠿ Cluster                         CREATE_COMPLETE
 ⠿ DockermlTaskExecutionRole       CREATE_COMPLETE
 ⠿ LogGroup                        CREATE_COMPLETE
 ⠿ DefaultNetworkIngress           CREATE_COMPLETE
 ⠿ DockermlTaskDefinition          CREATE_COMPLETE
 ⠿ DockermlServiceDiscoveryEntry   CREATE_COMPLETE
 ⠿ DockermlService                 CREATE_COMPLETE
```

Docker will interact with ECS and update progress in the terminal.

Now that our compose application has been deployed, we can watch the output.

```
$ docker compose logs -f
```

Step 4 - Tear down the ECS cluster

The Docker ECS integration will handle tearing down all resources that were created when we deployed the application.

```
$ docker compose down
[+] Running 10/10
 ⠿ dockerml                          DELETE_COMPLETE
 ⠿ DefaultNetworkIngress             DELETE_COMPLETE
 ⠿ DockermlService                   DELETE_COMPLETE
 ⠿ Cluster                           DELETE_COMPLETE
 ⠿ DockermlServiceDiscoveryEntry     DELETE_COMPLETE
 ⠿ DefaultNetwork                    DELETE_COMPLETE
 ⠿ DockermlTaskDefinition            DELETE_COMPLETE
 ⠿ CloudMap                          DELETE_COMPLETE
 ⠿ LogGroup                          DELETE_COMPLETE
 ⠿ DockermlTaskExecutionRole         DELETE_COMPLETE
```

All the resource that docker created on ECS are now deleted.