Running Container-Enabled Microservices on AWS
Lab Guide
Version 2.0

AWS-300-RCM-20-EN

Corrections or feedback on the course, please email us at:
aws-course-feedback@amazon.com.
For all other questions, contact us at:
https://aws.amazon.com/contact-us/aws-training/.

Contents

Lab 0
Pre-Requisites & qwikLAB Setup

Overview

During this one-day boot camp, you will learn how to use Amazon Web Services to architect, deploy, run, and manage applications constructed from a collection of containerized micro-services. You will be leveraging Docker to build containers and EC2 Container Service (Amazon ECS) to deploy and run these containers. You will also learn how to integrate other AWS services with your microservices architecture such as Elastic Load Balancers, Cloudwatch monitoring, and a third-party solution, HashiCorps Consul, for Service Discovery.

The bootcamp includes several hands-on exercises:

- Pre-requisites & qwikLABS setup (the guide you are now reading)
- Building and running your first container
- Constructing your microservice-enabled application
- Running microservices on Amazon ECS using the Service scheduler
- Running batch workers on Amazon ECS
- Service Discovery and Telemetry on Amazon ECS

What is qwikLAB?

qwikLAB gives you access to AWS services for the bootcamp. qwikLAB will provide a username and password for an AWS Account that you will use during the bootcamp. All charges relating to this account are included in the cost of your bootcamp. **You will not be charged for the AWS services you use during the bootcamp.**

Prerequisites

This lab requires:

- Access to a notebook computer with Wi-Fi running Microsoft Windows, Mac OS X, or Linux (Ubuntu, SuSE, or Red Hat)
- The qwikLABS lab environment is not accessible using an iPad or tablet device, but you can use these devices to access the student guide.
- For Microsoft Windows users: Administrator access to the computer
- An Internet browser such as Chrome, Firefox, or IE9 (previous versions of Internet Explorer are not supported)

- An SSH client such as PuTTY

Duration

This lab should take about 10 minutes.

Task 1: Getting Started with qwikLAB

Overview

First, you will need to create a qwikLABS account. **Please register for qwikLABS using the same email address that was used to register for this bootcamp.** If you already have a qwikLABS account you can log in and skip ahead and login.

Task 1.1: Create a QwikLABS account

1.1.1 In your web browser, go to the qwikLABS URL provided by your instructor.

1.1.2 Complete the Create a New Account section.

1.1.3 qwikLABS will send you an email confirmation following a successful registration

Task 1.2: Start your lab

1.2.1 After you have registered and have logged into qwikLABS, you should see a link for your bootcamp.

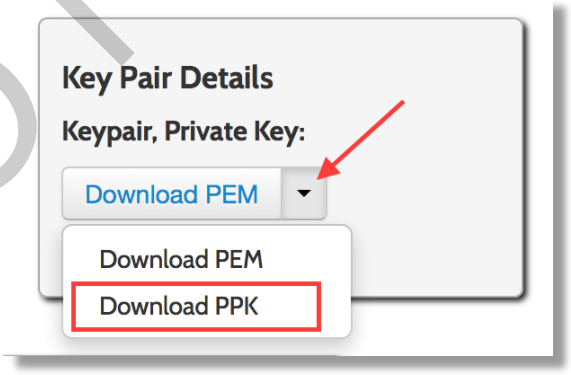
1.2.2 Click the “Start Lab” button.

1.2.3 If you are using Microsoft Windows, skip to the next step (1.2.4). Otherwise, download the PEM file from qwikLABS.

Note: You will likely have to change permissions on the downloaded PEM file for SSH to allow you to make a remote connect. The following command sets the correct permissions in OSX/Linux/Git Bash:

```
$ chmod 600 <YOUR PEM FILE>
```

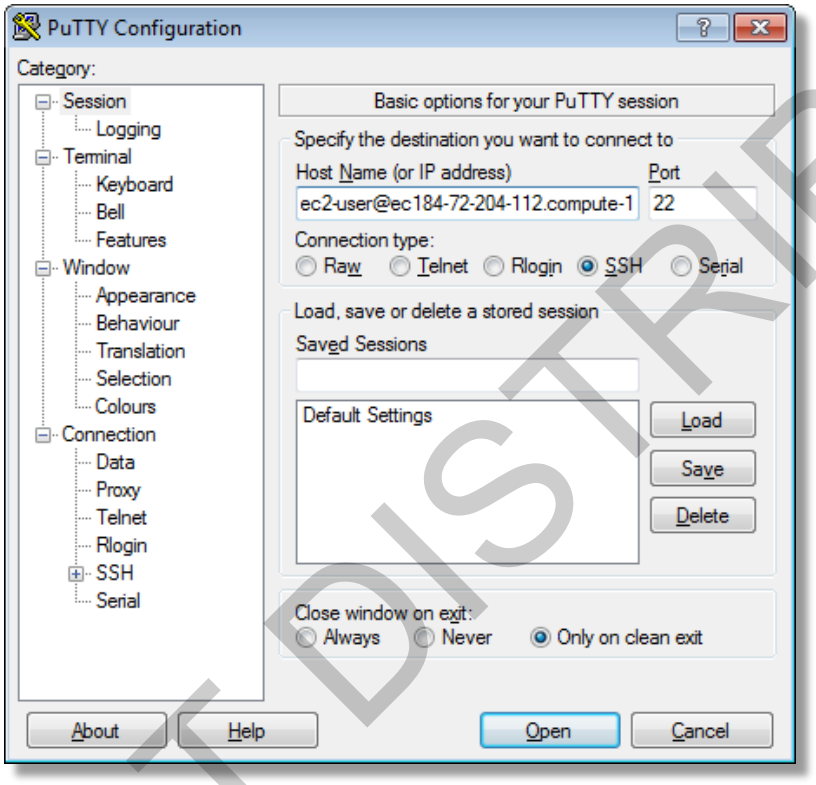
1.2.4 (Windows Only) Windows users will need to use the putty.exe application (or similar) to connect to their EC2 instance using SSH. PuTTY requires the key to be in PPK format.



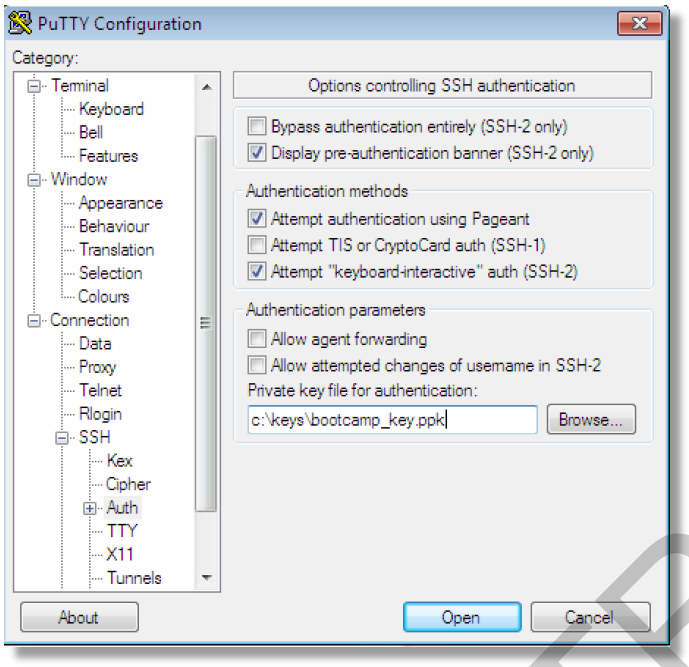
Windows users should download the PPK file

Here are the steps to configure and connect to your instance using PuTTY.

Start **PuTTY** (from the Windows Start menu, choose All Programs→PuTTY→PuTTY). A dialog box opens with a Category menu on the left side. On the right side, the basic options for your PuTTY session are displayed. In the Host Name field, you will enter the public DNS name or public IP address of your instance. You can optionally prefix the DNS name or public IP address with ec2-user@ to automatically log in with the ec2-user, which you will be using today for all instances you launch.

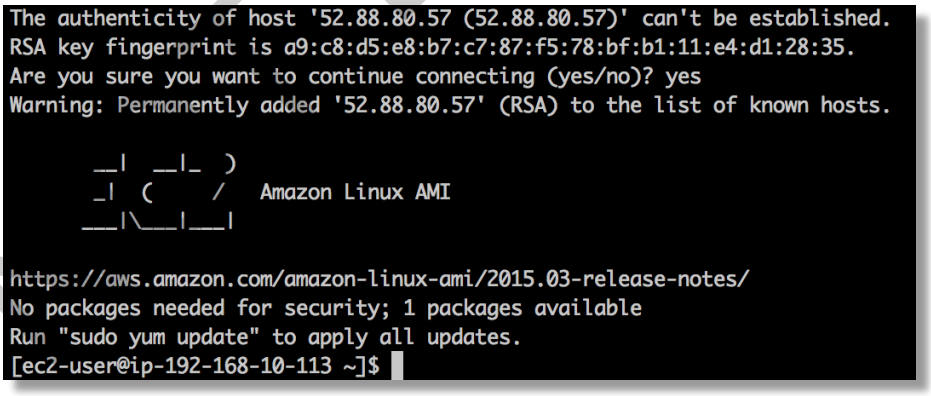


In the Category menu, under Connection, click **SSH** and then Auth. The options controlling SSH authentication are displayed. Click Browse and navigate to the PuTTY private key file you generated at the beginning of this workshop (i.e., “qwikLABS-XXXXX-XXXXX.ppk”).



Click Open. An SSH session window opens and PuTTY displays a security alert asking if you trust the host you're connecting to. Click Yes. In the SSH session window, log in as ec2-user if you didn't as part of starting the SSH session. Note: If you specified a passphrase when you converted your private key to PuTTY's format, you must provide that passphrase when you log in to the instance.

1.2.5 When connected, you should see something that looks like this (this screenshot is OSX Terminal - if you are using PuTTY your output should be fairly similar):



There may be messages about packages that can be applied. For the sake of the lab, you won't worry about applying those, but in a real-world scenario you'd want to make sure you were at least applying security updates and using the latest AMIs whenever possible.

Lab 1
Microservices

Overview

During this one-day boot camp, you will learn how to use Amazon Web Services to architect, deploy, run, and manage applications constructed from a collection of containerized micro-services. You will be leveraging Docker to build containers and EC2 Container Service (Amazon ECS) to deploy and run these containers. You will also learn how to integrate other AWS services with your microservices architecture such as Application Load Balancers, CloudWatch monitoring, and a third-party solution for secret sharing.

The bootcamp includes several hands-on exercises:

- Pre-requisites & QwikLABS setup
- Building and running your first container
- Constructing your microservice-enabled application
- Running microservices on Amazon ECS using the Service scheduler
- Running auto-scaling services on Amazon ECS
- Service Discovery, secret sharing, and Telemetry on Amazon ECS

Duration

This lab should take you about 45 minutes.

Command Reference File

command-reference.txt

Task 1: Connecting to Your EC2 Instance

Overview

Welcome to the first session of hands-on activity. Lectures are always fascinating and they keep you on the edge of your seat but when it comes to learning there is nothing quite like doing it yourself. So, let's start doing.

Scenario

Now is the time to refresh your Docker skills.

Task 1.1: Retrieve the CLI instance DNS name

Overview

The CLI instance you are going to connect to is part of a CloudFormation stack. Let's go find its name.

1.1.1 Point your browser to https://console.aws.amazon.com/cloudformation/

1.1.2 What you see in the table are "stacks", basically infrastructures that have been spun up according to a json-formattted template. You want to click on the line that has "CliInstanceStack" in its name.

1.1.3 In the lower pane, click on the Outputs tab.

1.1.4 Find the key named PublicDnsName and the value associated with it. It is the DNS you should use. In other words, the instance's DNS is the **value** for which the **key** is "PublicDnsName".

1.1.5 Copy this somewhere (your clipboard/buffer, for example).

You could also find the CLI instance DNS name from within Qwiklabs, in the blue tab called "Addl. Info" on the right side of the page.

Task 1.2: Connect to the instance

1.2.1 Change directory to where your .pem file is located.

1.2.2 SSH into the instance.

```
$ ssh –i <YOURKEY.PEM> ec2-user@<YOUR EC2 INSTANCE PUBLIC DNS NAME>
```


Building and Running the Monolithic Application in a Container

Overview

This task shows you how to create a container using a Dockerfile and how to run and stop a single container.

Scenario

You are connected to the CLI instance and will build the application as it is built since the startup you are part of was incepted, except you are going to wrap it in a container. So, it is going to be a containerized monolith. Then, you will play around with the app a little bit.

In a real-life scenario, you would probably have to fetch the source code from a repository but in the interest of time we have made the latest version available to you on the instance.

Task 1.3: Build the MustacheMe Docker image

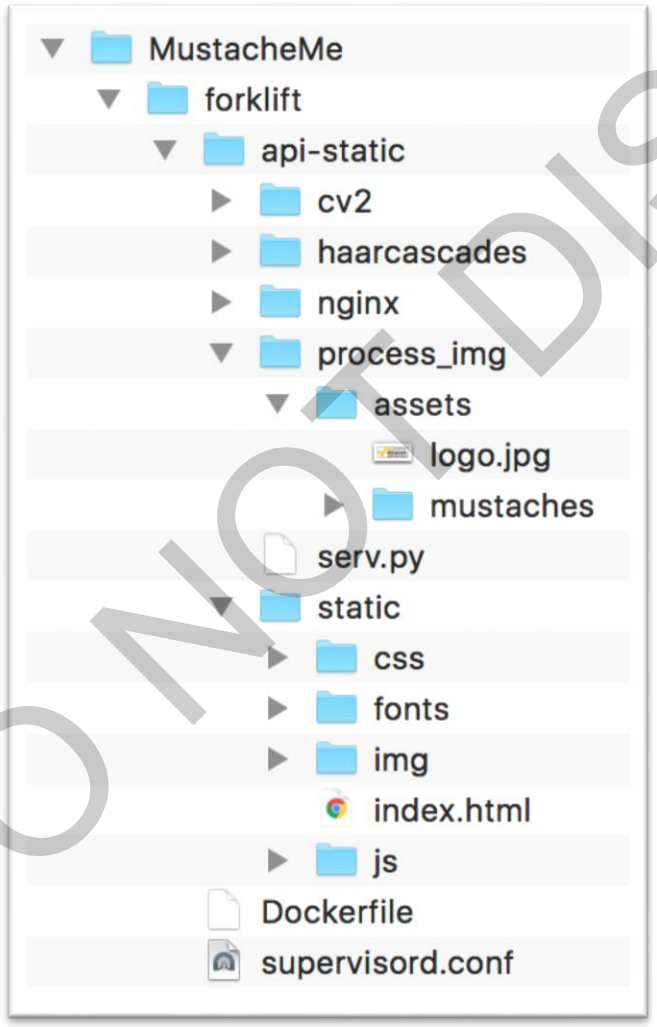
1.3.1 Navigate to the source directory.

```
$ cd lab-1-microservices/src/MustacheMe/forklift
```

1.3.2 Build the Docker image from the present Dockerfile (notice the dot at the end of the command line below):

```
$ docker build -t forklift .
```

1.3.3 While your first container image is getting assembled, you can quickly take the time to look at how this application was built.



Within the *api-static* folder are all the source files that you need. In *CV2* are the OpenCV (open source computer vision) libraries that are needed for placing the mustache, while the face detection part is kept in haarcascades. The *nginx* folder contains the web serving part of MustacheMe and you're left with two folders: *process_img* and *static*. In the process_img, you have the assets used for the image processing such as the moustaches and logos. After processing of images, the "moustached" images are stored there in a "processed" folder. And, static contains all the media and text assets we leverage in the website.

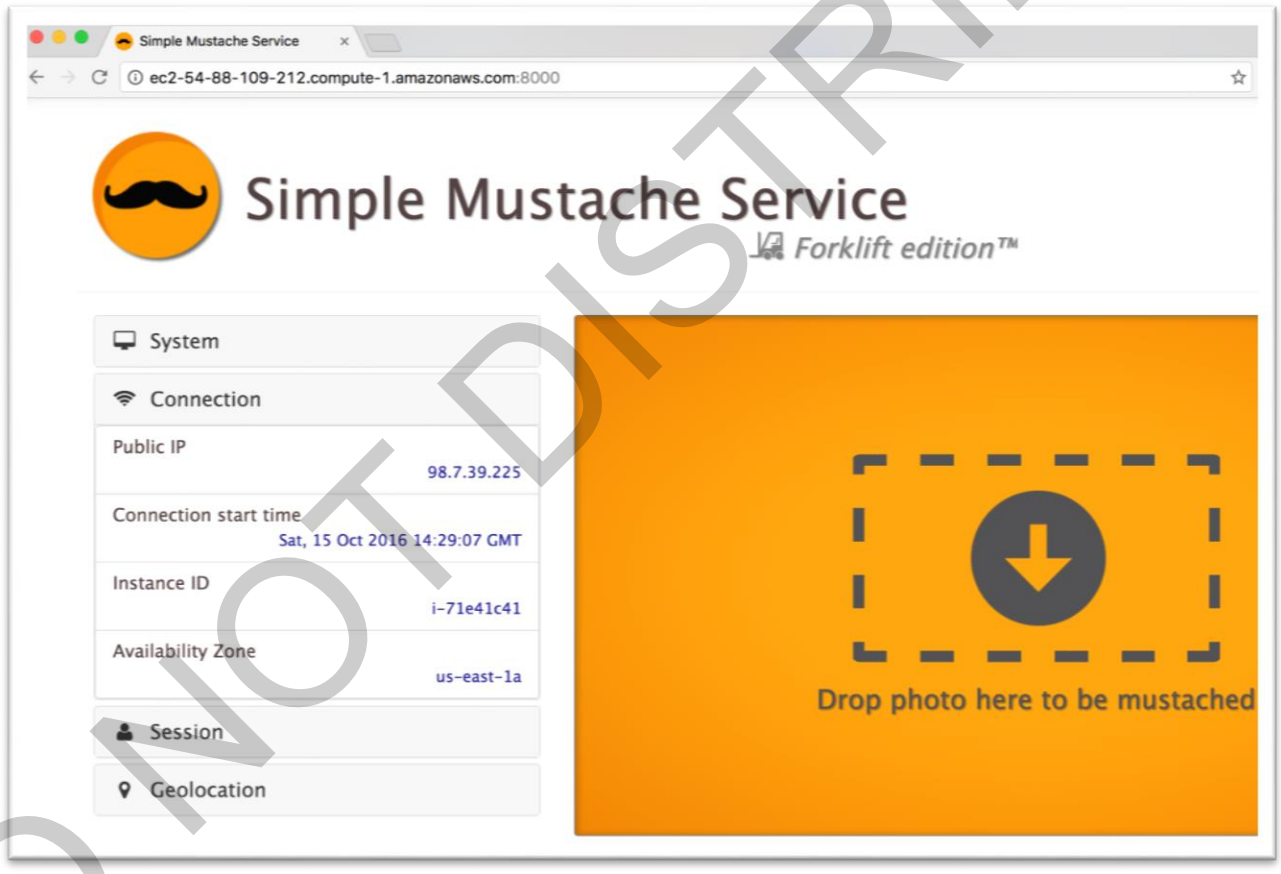
Task 1.4: Build and run the monolith container

1.4.1 Run the “forklift” MustacheMe container locally.

```
$ docker run –dp 8000:8000 forklift
```

You are binding port 8000 from the container to port 8000 on the host with –p. –d tells Docker to run in detached mode (equivalent to background processes in Unix).

1.4.2 Connect to the application from your browser. For this, you’ll simply need to copy the DNS name for the EC2 instance you were just connected to and paste it into your browser’s address bar. Then, add **:8000** at the end. You should see something like what is provided below:



In our experience a lot of corporate VPNs prevent the use of non-common TCP ports. Therefore, if you cannot reach the web front-end on port 8000, **make sure that you are not connecting via VPN.**

1.4.3 At this point, you should feel free to enjoy a bit of “mustaching”. Go ahead and drop a picture in the orange box and be amused as it facetiously places facial hair on the original picture.

Task 1.5: Stop this container

1.5.1 Let us shutdown this container.

```
$ docker stop $(docker ps -lq)
```

Good. Now, let’s get to breaking down this monolith.

Task 2: Building and Running Microservices, Locally

Overview

This task shows you how to assemble containers with Docker Compose so they can work together.

Scenario

The monolith runs inside a container. But, this container is rather unwieldy. What if your website is really successful and you need more front-end servers? Maybe there are more people willing to view their friends with a mustache than people willing to apply a mustache, and yet with the monolith you are left with having to run it all and duplicate it all. From a development perspective, every time you make a change within the application, at any place, you need to merge all your code, rebuild the whole application and ship it to your testing environment, and deploy it all. It would be a lot easier and agile if you had multiple small services that worked together but were decoupled and interchangeable.

Task 2.1: Build the MustacheMe base image

2.1.1 Navigate to the source directory.

```
$ cd ~/lab-1-microservices/src/MustacheMe/MustacheMeBase/
```

2.1.2 Build the Docker image from the present Dockerfile (notice the dot at the end of the command line below):

```
$ docker build –t mustacheme:base .
```

Task 2.2: Build the MustacheMe microservices Docker images and see them running with Docker Compose

2.2.1 Remain connected to the CLI instance and navigate to the source directory.

```
$ cd ~/lab-1-microservices/src/MustacheMe/
```

2.2.2 Build the Docker images for all microservices in one fell swoop with Docker Compose. The following is not only going to build all images required but also run the containers together. Check out docker-compose.yml for more information.

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

2.2.3 This brings us to another good opportunity to look at the innards of our application. What used to be a monolith is now made of 3 containers working with each other. There is a MustacheMeWebServer that handles the HTTP communication and displays the front-end. There is MustacheMeInfo container responsible for the info tidbits visible on the left side of the screen and finally there is the MustacheMeProcessor container that will take an image link as input and return a mustached picture as output.

2.2.4 Go ahead and see it run by typing what is given below:

```
$ docker-compose ps
```

2.2.5 Once again you should use your browser to navigate to the CLI instance's public DNS name, without forgetting to add :8000, and witness that the application is running, albeit being served by a combination of microservices.

2.2.6 Let us bring down the application.

```
$ docker-compose down
```

Time for us to run this whole setup on Amazon EC2 Container Service!

Task 3: Pushing the Microservice Images to Amazon EC2 Container Registry

Overview

This task shows you how to leverage Amazon ECR as the place of repository for your Docker images. Amazon EC2 Container Registry (ECR) is a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images.

Scenario

Now, you know the application can run as a set of microservices. Let us push these images to a safe place so they can be stored and shared.

Task 3.1: Create your first repository via the Console and push the first image to ECR

3.1.1 Point your browser to https://console.aws.amazon.com/ecs/ and click Repositories in the left column.

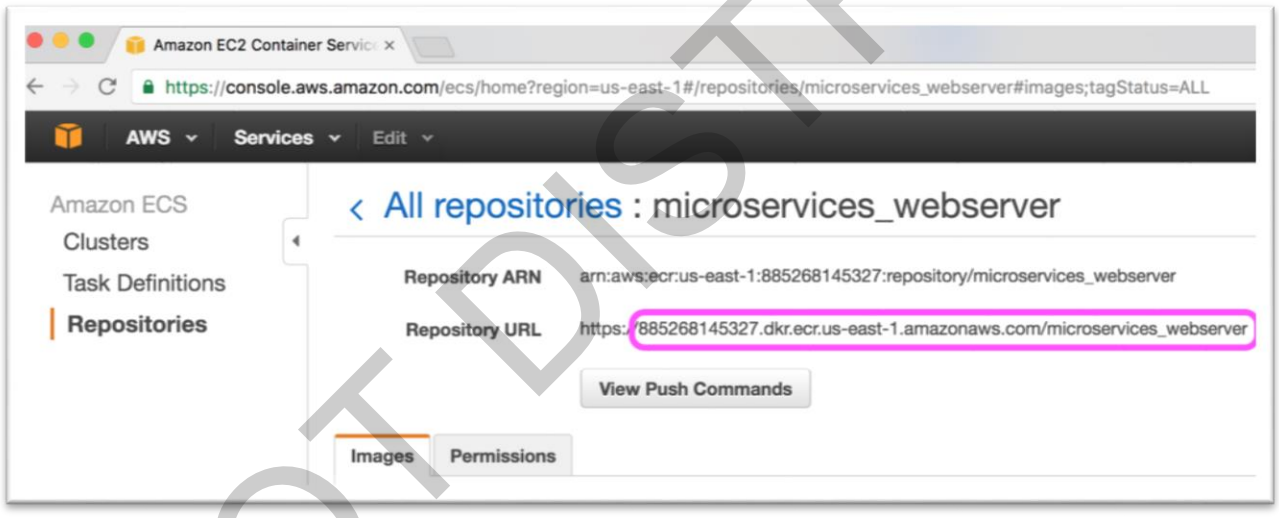
3.1.2 Click the Get Started button. If you do not see one then hit the Create repository button.

3.1.3 Enter “microservices_webserver” as the repository name and hit the Next step button.

The repository is created and you are shown a set of instructions. You can ignore these instructions today because you are leveraging the ECR Credential Helper. It is a native store for the Docker daemon that keeps Docker credentials safe and makes it easier to use Amazon EC2 Container Registry.

3.1.4 Click Done.

You need the repository URI. It is the Repository URL without the https:// prefix.



3.1.5 Let us tag the processor microservice.

```
$ docker tag mustacheme_webserver \

<ACCOUNTNUMBER>.dkr.ecr.<REGION>.amazonaws.com/microservices_webserver:la
test
```

That last bit with <ACCOUNTNUMBER> and <REGION> will be the URI you copied earlier. It is highlighted in the command above. You are using it again to push the image.

3.1.6 Push it.

```
$ docker push \

<ACCOUNTNUMBER>.dkr.ecr.<REGION>.amazonaws.com/microservices_webserver:la
test
```


Task 3.2: Create the other repositories and push the appropriate images

3.2.1 Now, onto our second image. Create the **microservices_processor** repo from the command line this time:

```
$ aws ecr create-repository --repository-name microservices_processor
```

This command will generate an output that contains “repositoryName”, “repositoryArn” and “repositoryUri”. You will want to copy URI.

3.2.2 Let us tag the processor microservice.

```
$ docker tag mustacheme_processor \

<ACCOUNTNUMBER>.dkr.ecr.<REGION>.amazonaws.com/microservices_processor:la
test
```

That last bit with <ACCOUNTNUMBER> and <REGION> will be the URI you copied earlier. It is highlighted in the command above. You are going to use it again to push the image.

3.2.3 And, push it.

```
$ docker push \

<ACCOUNTNUMBER>.dkr.ecr.<REGION>.amazonaws.com/microservices_processor:la
test
```

3.2.4 You’ve just created a repo for the webserver and the processor microservices, then tagged their images and pushed them to their respective repos. Now, you can do the last service by yourself. The Docker image is called microservices_info. All you have to do is follow the above 3 steps with a different service name. You’re on your own!

Task 4: Runing the MustacheMe Microservices on ECS with ecs-cli Compose

Overview

This task shows you how to bring up the MustacheMe service on your ECS cluster with the help of ecs-cli compose.

Heads-up: this task is **optional**.

Scenario

You have uploaded your Docker images to ECR. You can now easily deploy the MustacheMe microservices on your cluster. There are many ways to do this but because you used **docker compose** before, let's use the ECS equivalent.

Now our last engineer who tried to make ecs-cli compose work for us was not able to get it all up and running. He left the file as is, declaring "this isn't working, gotta take this call, I'll look into this in 10 minutes". This happened 2 months ago. Can you help us fix this YAML-formatted file? There are hints along with the instructions below.

Task 4.1: Configure ecs-cli compose

ecs-cli is pre-installed on the CLI instance you are connected to. But before you can use it, you need to configure it by telling it what region to use and optionally pointing it to an existing ECS cluster. You happen to have stood up a one-instance cluster so let us use it. You are going to need the ECS cluster name.

4.1.1 Find the ECS Cluster Name in the Additional Lab Information (just like you found the CLI instance information).

4.1.2 Now, go back to your CLI instance terminal and type the following:

```
$ ecs-cli configure --region $(aws configure get region) \
--cluster <ECS CLUSTER NAME>
```

If you ever need to troubleshoot ecs-cli, look in ~/.ecs/config but issues will only arise if you did not enter the proper region or cluster ID.

Task 4.2: Use a Compose file

4.2.1 You will find the **ecs-cli**-friendly compose file in the ecs-cli directory:

```
$ cd ~/lab-1-microservices/src/MustacheMe/ecs-cli
```

4.2.2 Here is how one would stand up microservices together with ECS CLI, simply by typing:

```
$ ecs-cli compose up
```

4.2.3 You can use the following command to find out what is running (or not running if something happened to fail in the previous step…hint hint):

```
$ ecs-cli ps
```

There are a few things wrong with the docker-compose.yml file but they fall into two sets. Maybe the ECS CLI documentation page will help you: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_CLI_tutorial.html#ECS_CLI_tutorial_compose_create

…

Hint 1: Maybe the image references are not 100% correct?

…

You do not need another hint, do you?

Hint 2: You need to link containers to each other. Do these links look right?

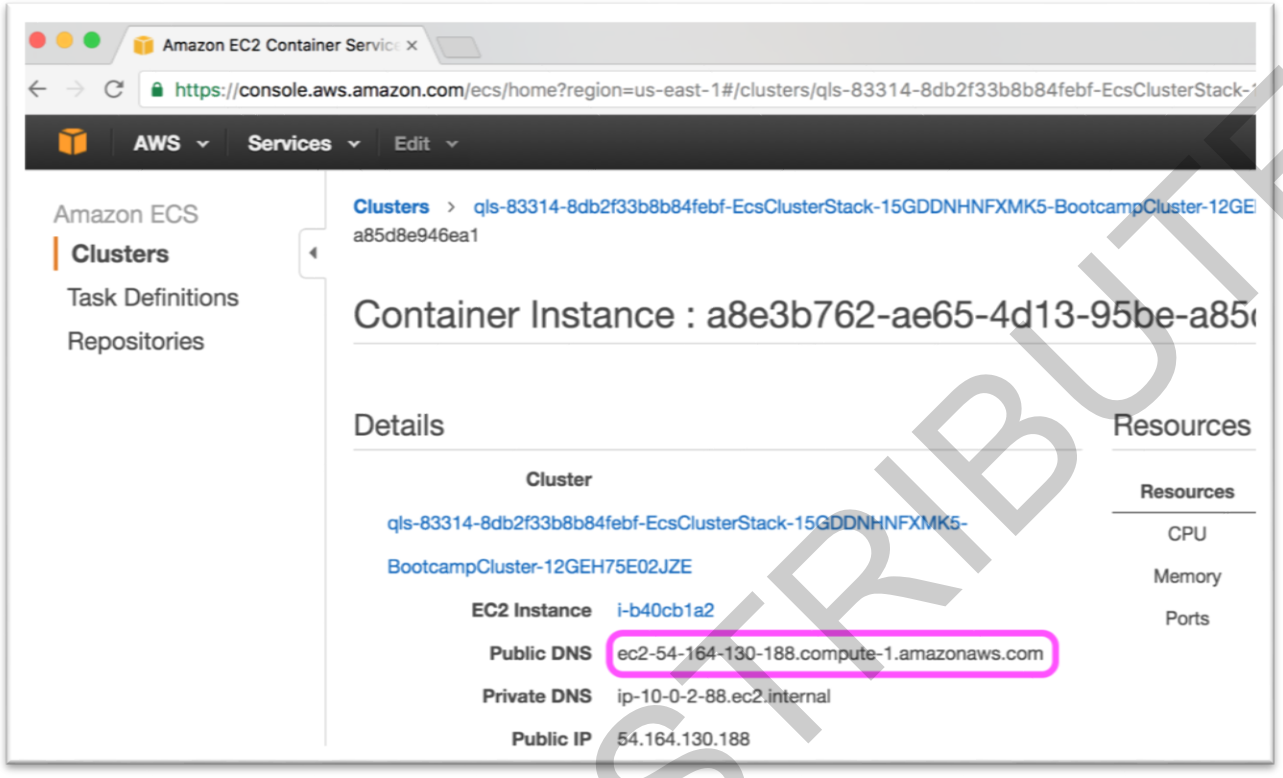
…

4.2.4 How do you make sure the system is working? First navigate to the ECS Console and click on your cluster.

4.2.5 Click on the ECS Instances tab.

4.2.6 Click on the Container Instance you see.

4.2.7 You will see a Public DNS record for this instance. Copy it.



4.2.8 When browsing to this destination, make sure to add :8000 to the URL.

4.2.9 When you are done, make sure to bring MustacheMe down.

```
$ ecs-cli compose down
```

4.2.10 Check that the services have changed to a stopped state.

```
$ ecs-cli ps
```

4.2.11 Lab 1 Microservices – Complete

Lab 2
Continuous Integration/Delivery Pipelines for Container based microservices

Overview

Lab 2 builds on the concepts and techniques you used during Lab 1. In Lab 2 you will take the 3 microservices from Lab 1 and build a continuous integration pipeline for each one automating the build, deployment and test of each microservice.

- **Front End:** The MustacheMe web application
- **Metadata:** The MustacheMe Info microservice
- **Image Processing:** The MustacheMe Processor service

The lab will include steps to automate the building and deployment of these individual microservices. You will start by building a Jenkins Docker image and turning it into an Amazon ECS service. All of the services will be built from sources that reside in Amazon CodeCommit using Amazon CodePipeline. The resulting images will be stored in Amazon ECR, then deployed onto Amazon ECS. All network traffic will flow through an Amazon ALB and will use different ports on the host ECS instance for each microservice. Finally, you will add tests using a 3rd party tool (Postman) that will allow you to validate each of the microservices you deploy.

In summary:

1. Create a Jenkins Docker image using the supplied Dockerfile
2. Create an Amazon ECR repository to store the Docker images
3. Create a Jenkins ECS service
4. Write a script to build and deploy three microservices.
5. Add a test suite to Code Pipeline to validate the microservice builds

Duration

This lab should take you between 45 minutes and an hour.

Connect to Your EC2 CLI Instance

Overview

Similar to Lab 1, you will need to connect to an EC2 instance to run the commands. Lab 2 creates its own CLI instance as part of the initial setup and deployment for Lab 2. You will not use the same CLI instance as Lab 1.

Task 1.1: Retrieve the CLI instance DNS name

Overview

The CLI instance you are going to connect to is part of a CloudFormation stack. Let's find its name.

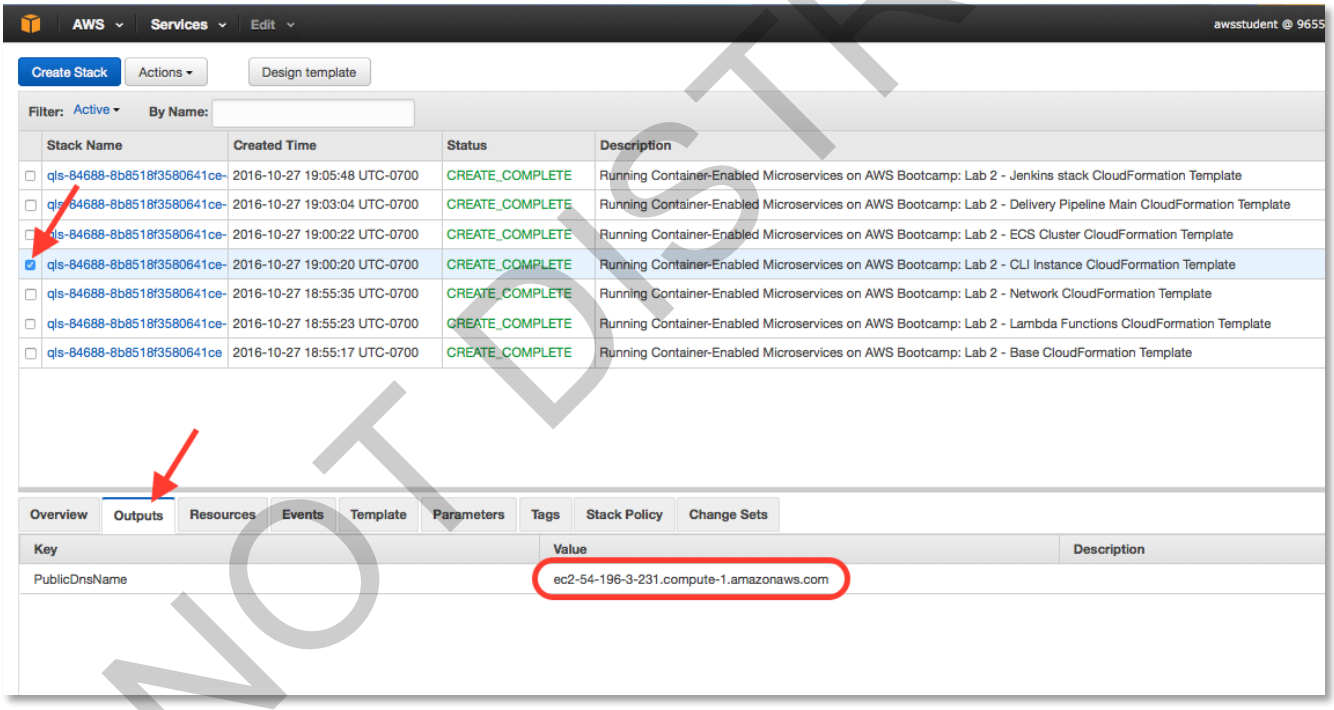
1.1.1 Point your browser to https://console.aws.amazon.com/cloudformation/

1.1.2 In the table you see a list of Stacks. These detail the infrastructure defined by YAML-formatted CloudFormation templates. Click on the line that has "CliInstanceStack" in its name.

1.1.3 In the lower pane, click on the Outputs tab.

1.1.4 The instance's DNS is the **value** for which **key** is "PublicDnsName"

1.1.5 Copy this somewhere (your clipboard/buffer for example)



Task 1.2: Connect to the instance

1.2.1 Change directory to where your .pem file is located.

Note: This is a separate .pem file from Lab1. Each lab will have its own credentials.

1.2.2 SSH into the instance.

```
$ ssh –i <YOURKEY.PEM> ec2-user@<YOUR EC2 INSTANCE PUBLIC DNS NAME>
```

Building and Running Jenkins as an AWS ECS Service

Overview

In this exercise, you will be using scripts to build a Jenkins Docker container and push it into ECR. Jenkins will then be provisioned from ECR to ECS as a service. Let's start by taking a look at the scripts to do this.

Command Reference File

command-reference.txt

Scenario

You can add flexibility to your architecture by breaking your monolith into separate microservices. However, this flexibility comes at the cost of increasing complexity. In order to keep this complexity manageable, you will want to automate as much of the process as possible. This automation also accelerates the time to deployment making your business more agile.

To build a full-fledged CI/CD process you will first need a build server. You will use Jenkins for this and while you could run it on an Amazon EC2 instance, this is a bootcamp about containers and microservices so let's run it as a service in Amazon ECR!

The next thing you will need to do is to integrate the Jenkins service into automation. You will use CloudFormation to create a set of Amazon CodePipelines to automate each microservice build and their subsequent deployment. You will script this to make it easier to repeat for each microservice.

Task 1.3: Build the Jenkins Docker image and push it to Amazon ECR

1.3.1 Navigate to the source directory on the Lab2 CLI instance.

```
$ cd /home/ec2-user/lab-2-pipeline/src/jenkins
```

In this directory there is a Dockerfile to build the Jenkins container.

This file uses the main bootcamp Jenkins Docker image as its base and then simply adds in some plugins and scripts. Take a quick look at the Dockerfile to see if you understand what it is doing.

```
$ cat Dockerfile
```

1.3.2 Before you can run the Docker commands you will need to create an EC2 Container Repository (ECR). The following AWS CLI command will create an ECR repository named "jenkins"

```
$ aws ecr create-repository --repository-name jenkins
```

1.3.3 You are almost ready to build a Jenkins Docker image. Before doing this you need to get the URL for the repository you just created. You will need this value as input parameter for the Docker build and push commands. An easy way to get the URL is to simply query ECR using the AWS CLI. Let's write the value into an environment variable on the instance so you can continue to use it in subsequent commands:

```
$ export JENKINS_REPO_URI=$(aws ecr describe-repositories \
--repository-names jenkins --query \
'repositories[].repositoryUri' --output text)
```

Next verify that these environment variables are set correctly by running the command:

```
$ env | grep JENKINS
```

You should see an output similar to this, where JENKINS_REPO_URI is set:


```
[ec2-user@ip-10-0-1-95 jenkins]$
[ec2-user@ip-10-0-1-95 jenkins]$ env | grep JENKINS
JENKINS_REPO_URI=965597281383.dkr.ecr.us-east-1.amazonaws.com/jenkins
[ec2-user@ip-10-0-1-95 jenkins]$
```

Note: if your ssh session becomes disconnected or you open another connection, you will need to re-export this environment variable.

1.3.4 Now that your environment is setup you can run the docker build command. The "-t" adds a tag of the ECR repository.

```
$ docker build -t ${JENKINS_REPO_URI}:lab-2-pipeline \
/home/ec2-user/lab-2-pipeline/src/jenkins
```

1.3.5 Run docker images to list the images and show the repository they are tagged with:

```
$ docker images
```

You should see your jenkins image and your ECR repository.

1.3.6 Next you can use docker push to copy the jenkins docker image from your local instance into the AWS ECR repository:

```
$ docker push ${JENKINS_REPO_URI}:lab-2-pipeline
```

1.3.7 When that completes you can view the image in ECR via the AWS Management Console or by running given command:

```
$ aws ecr list-images --repository-name jenkins
```

Now that you have built and deployed the Jenkins image to EC2 Container Registry you can move onto the next section, where you will turn Jenkins into an ECS service and use automation to build and deploy the other components of your MustacheMe application as a set of individual containerized microservices.

Task 1.4: Turn your Jenkins Docker image into an AWS ECS Service

1.4.1 Running Jenkins as a service inside of Amazon ECS means that ECS will automatically restart the Jenkins Docker container should it become unresponsive. You could deploy this image as a service using the AWS Management console, but in the spirit of automation you have created a YAML file that you can use with AWS Cloudformation. This provides a programmatic way to deploy and update your infrastructure. Many people find the YAML format easier to read and write than JSON. Let's review the YAML file:

```
$ cat /home/ec2-user/lab-2-pipeline/scripts/jenkins-ecs-service.yaml
```

Do you understand the contents of this file? You will have more complex YAML files later on in the lab. Please ask for assistance if you are unclear of the purpose of any of the lines in the YAML file.

1.4.2 Now let's run the command to deploy Jenkins as a service using the above YAML file as a template:

```
$ aws cloudformation create-stack --stack-name JenkinsService \
--template-body file:///home/ec2-user/lab-2-pipeline/scripts/jenkins-ecs-
service.yaml
```

This deployment will take a few minutes to complete or return a status. You can see the stack events in the AWS Management Console on the CloudFormation page.

There is a refresh button on the page to refresh the status.

Did the stack create successfully? What events happened? What order did they happen in? Do the events and status match what you saw in the YAML file?

Task 1.5: Use CloudFormation and CodePipeline to create a CI/CD process for your microservices

1.5.1 Here are the next steps you will need to do per microservice:

(1) Create a CodeCommit repo for your Microservices

(2) Create a Jenkins build job for the Microservice

(3) Create a CodePipeline for the Microservice

(4) Create an Application Load Balancer Listener and Target group for the Microservice

(5) Clone a Git repo and commit a version to Amazon CodeCommit rep

Steps 1-4 have been coded into a YAML CloudFormation template for you to deploy.

1.5.2 Let's first take a look at this YAML file.

```
$ less /home/ec2-user/lab-2-pipeline/scripts/microservice-pipeline.yaml
```

Note all of the input parameters required. This is because the template will be reused for each of your three microservices as well as the last part of the lab where you are writing tests.

Key parts of the CloudFormation include the following:

- Custom CloudFormation resource called "*JenkinsBuildJobResource*" used to create the Jenkins job which is implemented as a Lambda function. It takes a Jenkins job config XML template file and replaces the provided parameters in the file and uploads it to the Jenkins server to create a new build project.
- CodeCommit repository resource to create the CodeCommit respository with the same name as the name of the microservice
- Application Load Balancer listener resource to forward traffic sent to a specific port on the Application Load Balancer to the microservice.
- Application Load Balancer Target Group that will be used by ECS to attach the ECS services for this microservice.
- CodePipeline resource containing the CodeCommit repository as the source triggered on new git push commands, Jenkins as the build resource and an AWS Lambda function that will create the microserivce ECS Task Definition and deploy the ECS service for the microservice.

1.5.3 Now you will use CloudFormation to launch this stack using the AWS CLI. You are using the CLI because further on in the lab you will automate this step:

```
$ cd /home/ec2-user/lab-2-pipeline

$ aws cloudformation create-stack \
--stack-name MustacheMeWebServerPipeline \
--parameters \
ParameterKey=MicroserviceName,ParameterValue=MustacheMeWebServer \
ParameterKey=RepoName,ParameterValue=mustachemewebserver \
ParameterKey=PortNumber,ParameterValue=8000 \
--template-body file://scripts/microservice-pipeline.yaml
```

1.5.4 You can watch the status of the CloudFormation stack on the command line with the following command:

```
$ aws cloudformation wait stack-create-complete \
--stack-name MustacheMeWebServerPipeline
```

1.5.5 When the stack has completed successfully, you can use git to clone the MustacheMe web server code into your own CodeCommit repo (this is the 5th step in task 1.5.1). Before running any git command you need to set another environment variable for the region. Then you can use git to clone the repository.

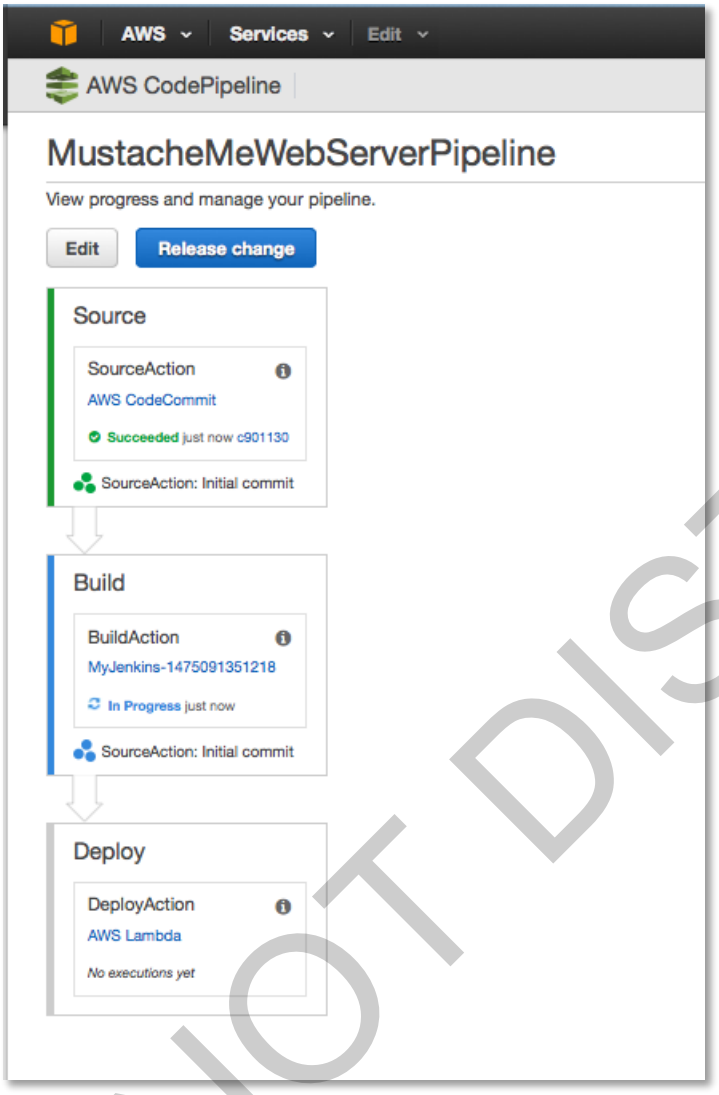
```
$ export AWS_REGION=$(aws configure get region)

$ git clone \
https://git-codecommit.${AWS_REGION}.amazonaws.com/v1/repos/MustacheMeWebserver \
/home/ec2-user/repos/MustacheMeWebserver
```

It is an empty repository that you are cloning, hence you can discard the relevant warning.

1.5.6 After cloning the repository, copy the code to your local repository directory. Then you use git to add all the files, add a commit message and perform the initial commit:

```
$ mv /home/ec2-user/lab-2-pipeline/src/MustacheMe/MustacheMeWebServer/* \
/home/ec2-user/repos/MustacheMeWebServer/

$ cd /home/ec2-user/repos/MustacheMeWebServer
$ git add -A
$ git commit -m "Initial commit"
$ git push -u origin master
```


1.5.7 After you push the changes to AWS CodeCommit, observe that they will be picked up by Code Pipeline and it will process them. This is because the CodePipeline is configured to watch for commits on the CodeCommit repository. You can see this visually using the AWS Management Console:



Congratulations! You have automated the deployment and deployed your first microservice! Just to recap, youhave done the following steps to deploy your microserivce:

- Run a CloudFormation script to create the following resources for your microservice:
  - CodeCommit repository for the microservice
  - Jenkins job via a Lambda function and custom CloudFormation resource that uploads a new Jenkins job using the Jenkins API.
  - CodePipeline for the microservice linking the CodeCommit repository where the source code resides to the Jenkins build job to build and deploy the Docker

image for the microservice to ECR and finally connecting to the AWS Lambda function that deploys the microservice via a CloudFormation template.

- Committed the source code of the microservice to the local git repository for the microservice and pushed the code to the CodeCommit repository triggering a deployment of the microservice via the CodePipeline service.

1.5.8 You will need to run the same five steps for the remaining Microservices:

(1) Create a CodeCommit repo for the Jenkins Microservice

(2) Create a Jenkins build job for the Microservice

(3) Create a CodePipeline for the Microservice

(4) Create an Application Load Balancer Listener and Target group for the Microservice

(5) Clone a Git repo and commit a version to Amazon CodeCommit rep

You could run them by hand, but let's use a script to run them. It's less error-prone and will be faster. The script is a grouping of the commands you have already used.

We have created a version of this script for you.

```
$ cat /home/ec2-user/lab-2-pipeline/scripts/deploy-microservice.sh
```

Take a look at it to see how it is running the commands you just ran by hand.

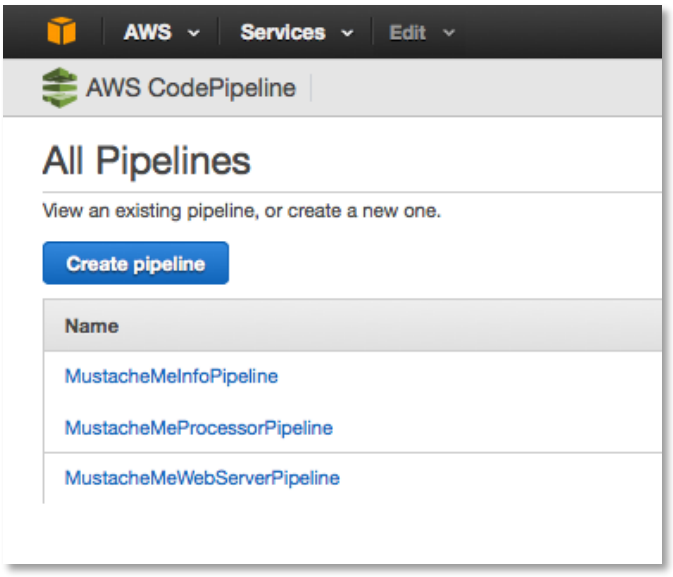
1.5.9 The script requires two inputs: the name of the service and the port it will run on:

```
deploy-microservice.sh <MICROSERVICE_NAME> <PORT>
```

1.5.10 Run the following commands to build the pipeline for the other remaining two microservices:

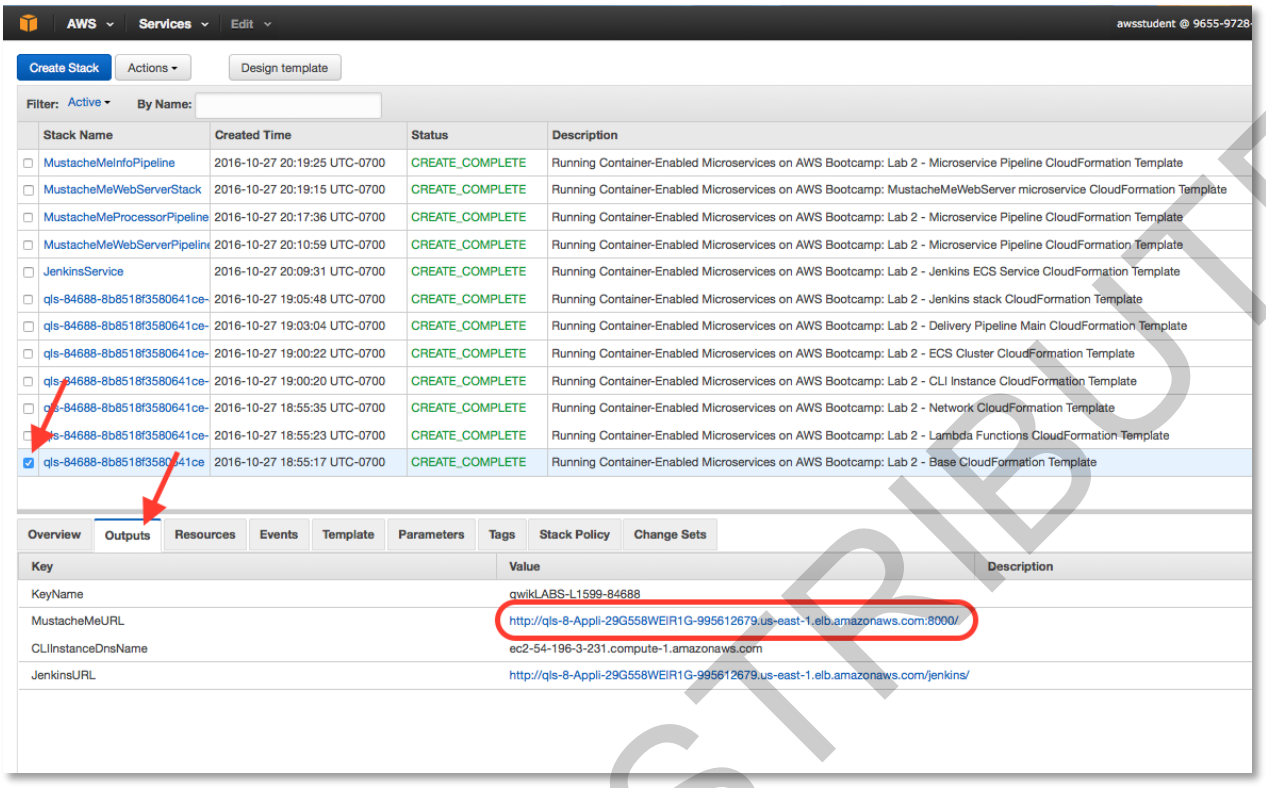
```
$ cd /home/ec2-user/lab-2-pipeline/scripts/

$ ./deploy-microservice.sh MustacheMeProcessor 8082

$ ./deploy-microservice.sh MustacheMeInfo 8092
```

1.5.11 Now you should see three separate Pipelines in the CodePipeline Management console display:



Note that each CodePipeline will be in a failed state as they will attempt to run when created however the CodeCommit repository per microservice will be empty. As soon as the code is committed and pushed to the CodeCommit repository a new release will be triggered which should complete successfully.

1.5.12 When all three microservices are deployed through each pipeline you should be able to view them by browsing to the URL of the ALB. Wait **until all three** of the microservice CloudFormation stacks names: "*MustacheMeInfoStack*", "*MustacheMeProcessorStack*" and "*MustachMeWebServerStack"* are deployed and in the state "***CREATE_COMPLETE***". You will find this in the output tab of the base CloudFormation stack:



Take a break and add a mustache to an image to see your microservices in action! Remember the MustacheMe web application is running on port 8000 so you will need the above URL to access it. If the image processing section is not working it will be because the MustacheMeProcessor microservice is not yet deployed and working, Wait until the CloudFormation stack name: *MustacheMeProcessorStack* is in the state: *CREATE_COMPLETE*. If the Session Info and Connection Info are not returning any data then the MustacheMeInfo microservice is not yet up and running so wait until the CloudFormation stack name *MustacheMeInfoStack* is in the state *CREATE_COMPLETE*.

Task 1.6: Add a Test phase to each microservice deployment pipeline (Optional)

1.6.1 Now that you have all 3 microservices working you can add a test action to each microservice to ensure each one is working as expected. You will be using the Postman testing framework to do this integrated with Jenkins. As part of your continuous integration process Jenkins will be invoked to run the postman test script and output the results. If any of the tests fail, the Jenkins build will fail.

View the postman test script for the MustacheMeWebServer microservice by running the following command:

```
$ cat /home/ec2-user/repos/MustacheMeWebServer/postman-collection.json
```

You will notice that the contents of the file run a couple of tests against the microservices endpoint:

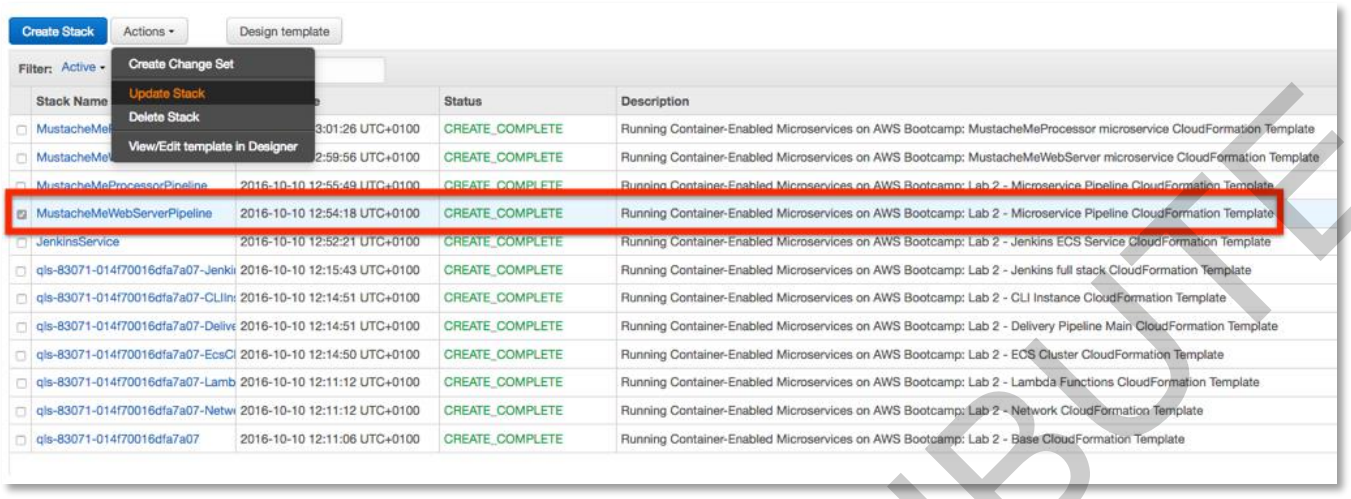
- We have configured a Postman “collection” for the bootcamp but customized to the specific URL of your MustacheMe microservice
- Test if the endpoint returns an HTTP response code of 200 (OK)
- Test if the returned html contains the text “Simple Mustache Service”

1.6.2 You can use the update command in CloudFormation to update the microservice pipeline CloudFormation template. This will add an extra step after the deploy stage to the build pipeline to test the endpoint for your microservice. Change the CloudFormation parameter called “*ExtendedFlag*” so that you create a pipeline with the added “Test” action and also deploy a Test project for the microservice to Jenkins.

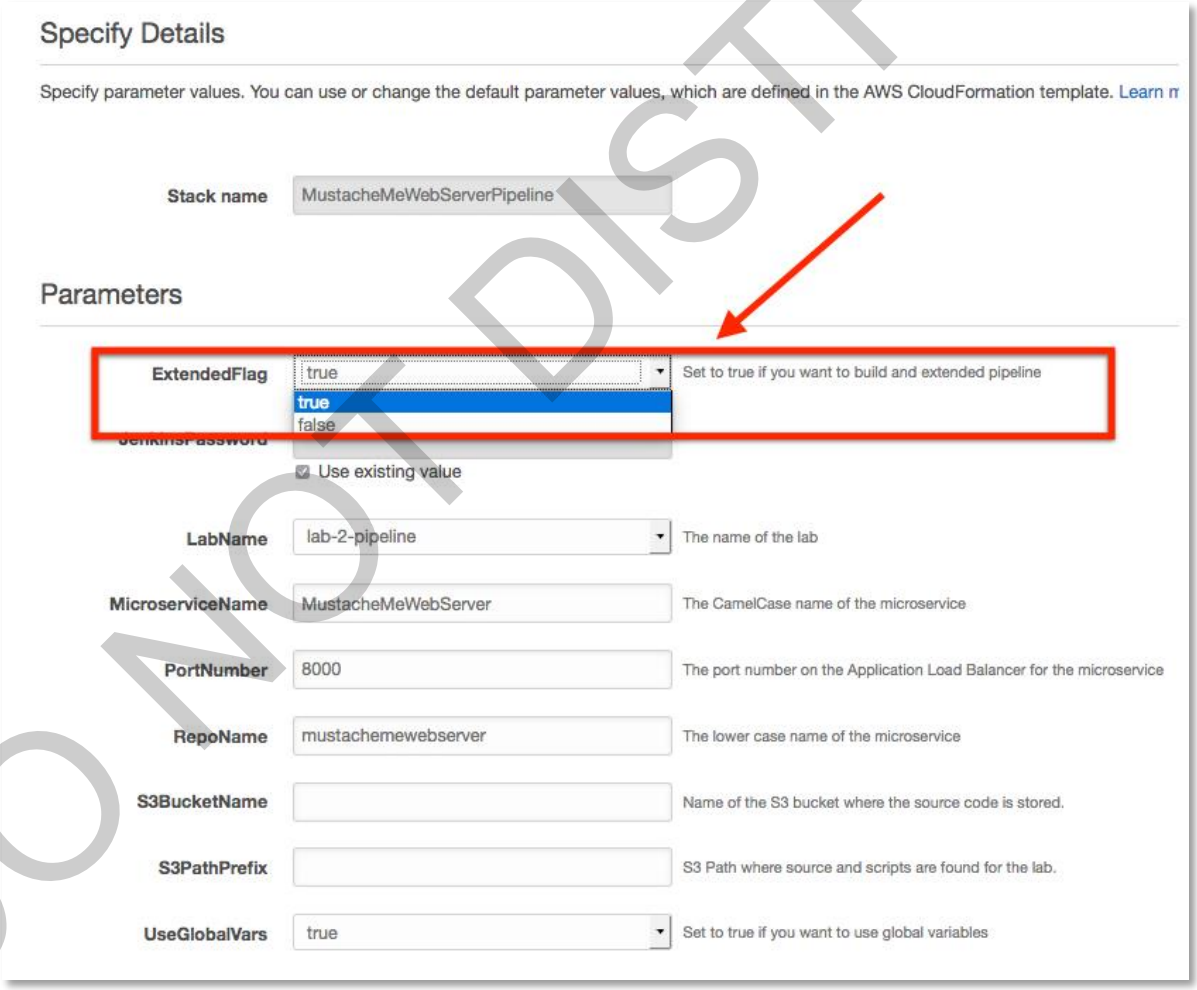
First, take a look at the Cloudformation template that will be updated and notice the resource named “ExtendedCodePipeline” will be created instead of the resource “SimpleCodePipeline” and it will also create the resource “JenkinsTestJobResource” which is a Jenkins job to test each microservice end point.

```
$ less /home/ec2-user/lab-2-pipeline/scripts/microservice-pipeline.yaml
```

1.6.3 To modify the CodePipeline open the CloudFormation service in the AWS Management Console. Select the CloudFormation stack name “MustacheMeWebServerPipeline” and click the “Update Stack” option like shown in the screenshot below.

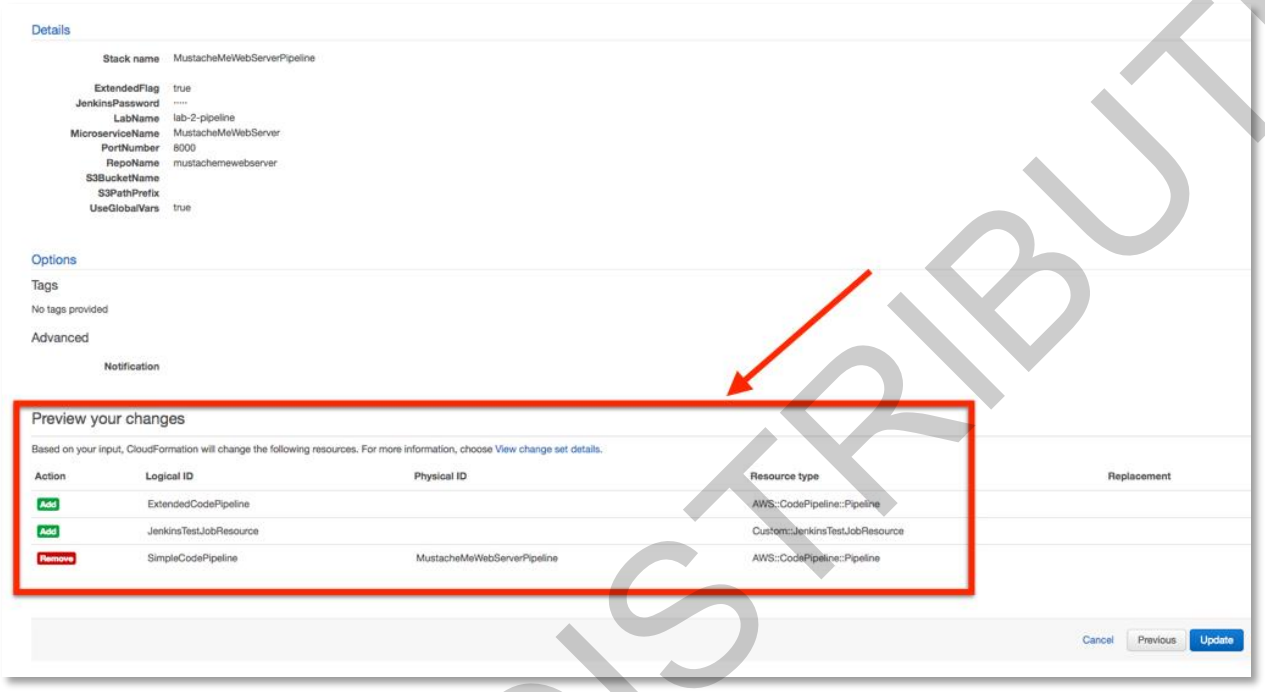


1.6.4 Select the “Use current template” option and click Next. Select the Parameter named “ExtendedFlag” and select the “true” option shown in the screenshot below and then click Next.



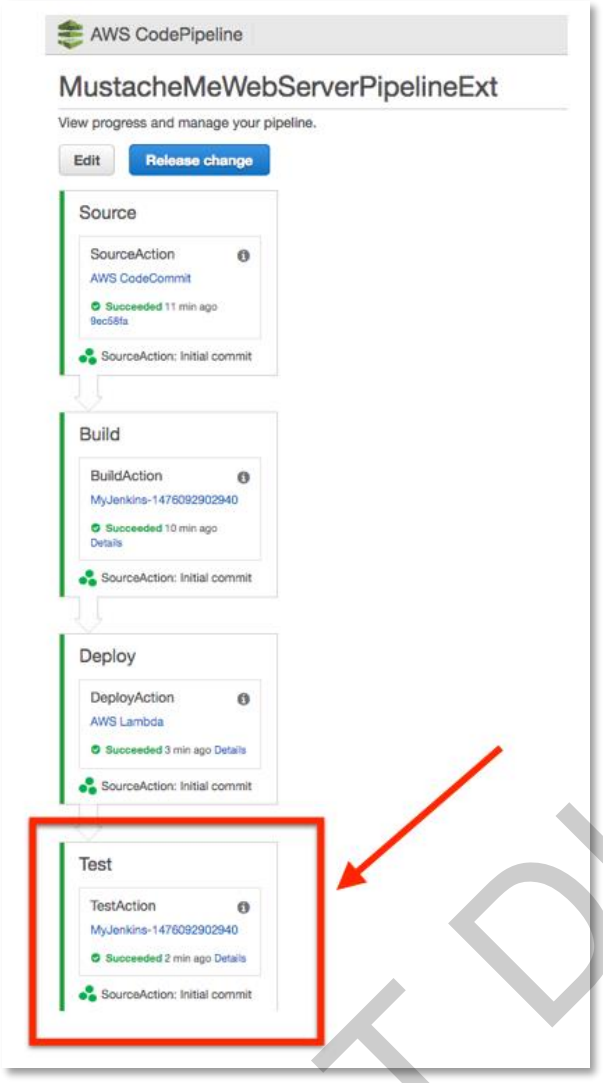
1.6.5 On the Options page, accept the defaults and click Next.

1.6.6 On the Review page you should now see the Change Sets that CloudFormation has calculated based on what is currently deployed and the changes that need to be made. You should be able to see that one CodePipeline will be removed and another added, and a Jenkins Test Resource added like shown in the screenshot below. Click the Update button to implement the changes.



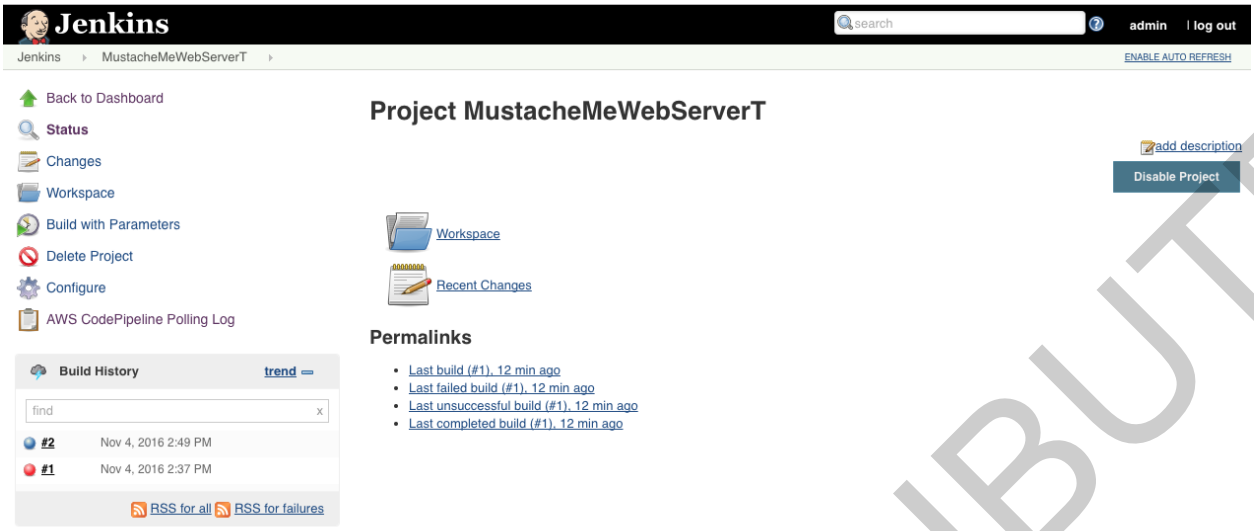
1.6.7 Click the CodePipeline service in the AWS Management Console. Select the Pipeline with the name “MustacheMeWebServerPipelineExt”. Notice how there is now an extra stage called “Test” with an action called “TestAction” like shown in the screenshot below.

Please note that it will take several minutes for the pipeline to complete to the “Succeeded” state as it needs to pass through all of the stages.

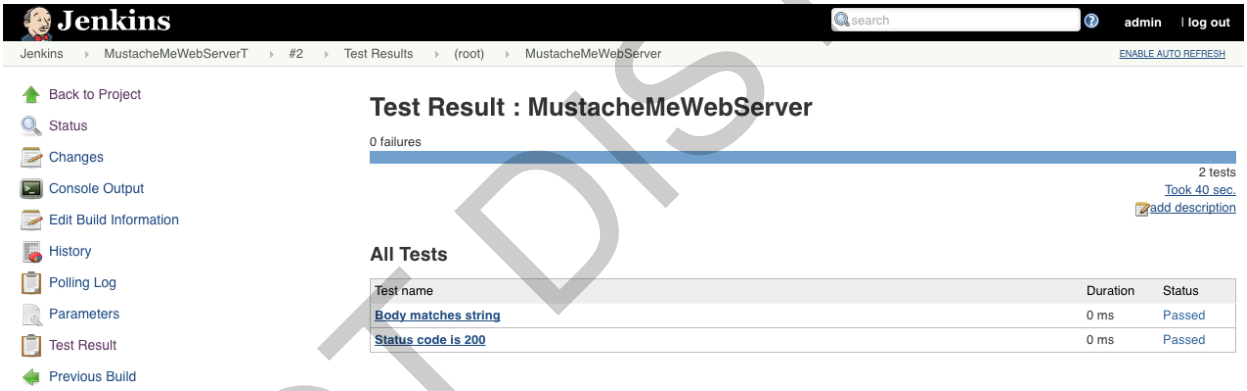


1.6.8 You can validate the test results by clicking on the Jenkins provider link in the test action. This will open a page similar to the following. The password for Jenkins' Admin user is the same as the password you used to login to the AWS Console. You can find it as part of the AWS Console Details in Qwiklab or as the **JenkinsPassword** output value of the **JenkinsStack** CloudFormation stack.

1.6.9 Click on the successful build number **#2.**



Click on the link "**(root)**" in the **All Tests** section of the page. Then click the link "**MustacheMeWebServer**" in the **All Tests** section of the page. This should bring up a page like the following showing the successful execution of 2 tests from the postman collection file.



1.6.10 Now repeat the step for the other two microservices (`MustacheMeProcessorPipeline` and `MustacheMeInfoPipeline`) by running the following CLI commands:

```
$ aws cloudformation update-stack --stack-name \
MustacheMeProcessorPipeline --use-previous-template --parameters \
ParameterKey=MicroserviceName,UsePreviousValue=true \
ParameterKey=RepoName,UsePreviousValue=true \
ParameterKey=PortNumber,UsePreviousValue=true \
ParameterKey=ExtendedFlag,ParameterValue=true

$ aws cloudformation update-stack --stack-name \
MustacheMeInfoPipeline --use-previous-template --parameters \
ParameterKey=MicroserviceName,UsePreviousValue=true \
ParameterKey=RepoName,UsePreviousValue=true \
ParameterKey=PortNumber,UsePreviousValue=true \
ParameterKey=ExtendedFlag,ParameterValue=true
```

1.6.11 You now have a fully automated CI/CD process for building, testing and updating your application.

That is the end of lab 2. We hope you enjoyed it.

Lab 3
Availability and Scale of Microservices

Overview

This part of the bootcamp is about making sure that your microservices are always available and can grow to meet demand. No one wants to be woken up at 3 a.m. because a component of the architecture is down. Amazon Web Services makes it easy to design for high availability without needing human intervention for recovery. You will learn how to apply self-healing to both your container instances powering the ECS cluster and your microservices.

In addition to high availability, AWS is also about scale. If a company or developers are not able to service demand when it starts increasing, then early adopters and eager customers are going to be frustrated. For a startup, this can be the crucial moment when everything should be running perfectly and customers are satisfied. Failing at this stage is doubly damaging because there is interest for the service but it cannot be met. Hence, you will focus on being able to scale services and the underlying infrastructure (i.e., container instances). Understanding that such a rush of customers can happen at any time, you will make sure to automate the scaling.

The lab includes several hands-on exercises:

- Reviewing the services deployed
- Ensuring services are highly available
- Ensuring high availability of the ECS cluster
- Setting up auto scaling for container instances
- Identifying metrics and auto scaling microservices

Prerequisites

This lab requires:

- Completion of Lab 1 and Lab 2 of “Running Container-Enabled Microservices on Amazon ECS”

Duration

This lab should take you about 90 minutes to complete.

Command Reference File

command-reference.txt

Task 1: Reviewing the Environment

Overview

The last lab had us deep into Continuous Integration/Continuous Deployment. You deployed a CI/CD pipeline for rapid development and deployment with consistent build steps. This part of Lab 3 will get you acquainted with the various parts as they have been laid out. Since you are picking up where Lab 2 left off, the MustacheMe application has been pre-deployed for you via the CI/CD pipeline.

Scenario

MustacheMe is almost in production. You have a working product and you know that you can iterate on the existing version at great speed with the existing CI/CD pipeline. In the seat of an Operations employee, it is important that you get familiar with all parts of the setup.

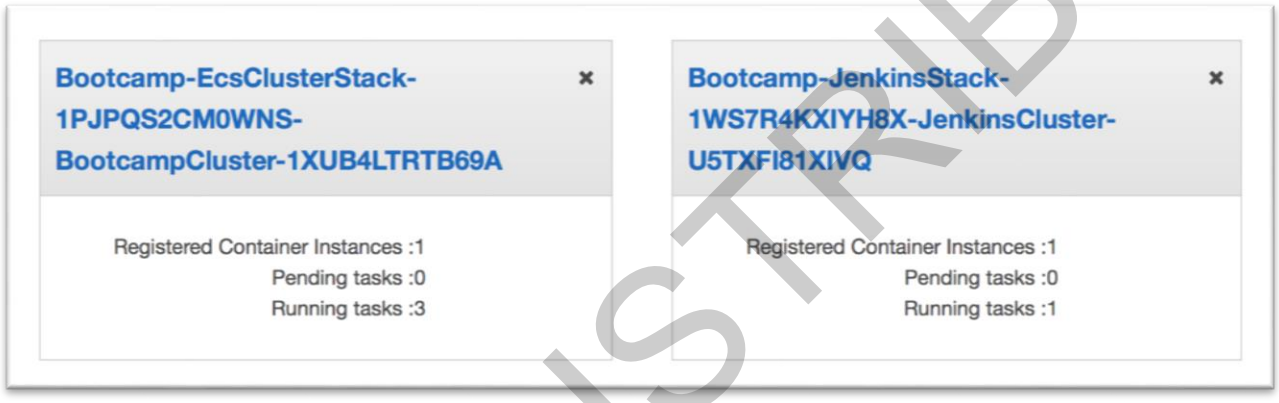
Task 1.1: EC2 Container Service: Clusters, Services

Overview

Two clusters were created for us. Let's explore:

1.1.1 Go to the ECS console by clicking on https://console.aws.amazon.com/ecs/. Make sure you are in the correct region.

1.1.2 You will see at least two clusters, like below. Full names will vary, as suffixes are randomly generated by CloudFormation.



1.1.3 Click on the **JenkinsStack** cluster.

1.1.4 You will see that one service is running. If you click on the **Tasks** tab you will be shown the one task that is part of the Jenkins service.

1.1.5 Clicking the **ECS Instances** shows one container instance as a member of this ECS cluster.

1.1.6 Now click on **Clusters** at the top of the page and click the **EcsClusterStack** on the next page.

1.1.7 You will be shown three services: **MustacheMeInfo**, **MustacheMeProcessor** and **MustacheMeWebServer**.

1.1.8 Click on the **MustacheMeWebServer** service.

1.1.9 On this page, you will find a few details such as the **Task Definition** used for the tasks that are part of this service, the **Desired count** of tasks, and how many tasks are **Pending** and **Running**.

1.1.10 In the pane at the bottom, you'll find the **Metrics** tab. Two charts are made available to you that will inform you about the CPU and Memory Utilization of the service. These are measured in minimum, average and maximum.

1.1.11 At the top right of the service page is the name of the target group that this service is part of. Go ahead and click on the **Target Group Name**.

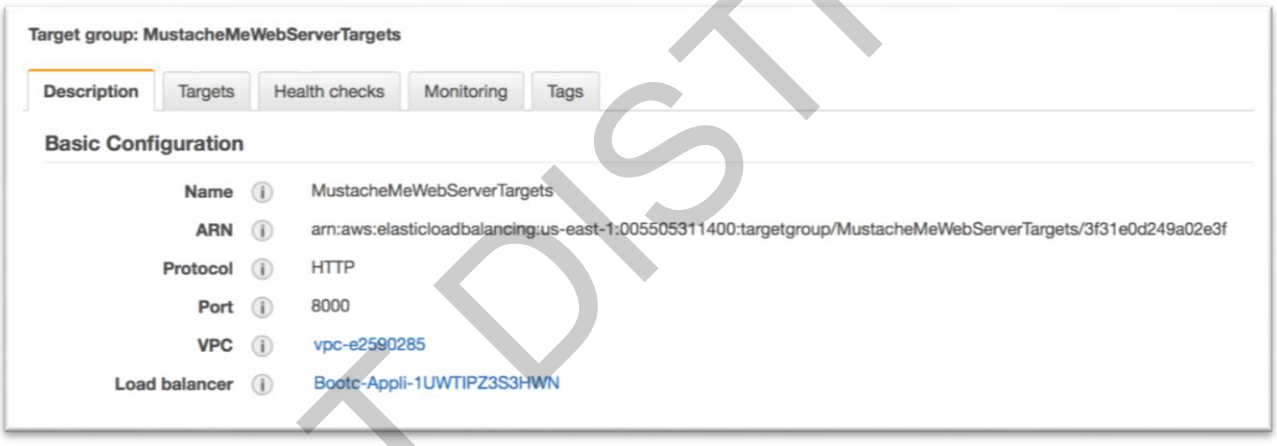
Task 1.2: Application Load Balancer

Overview

Application Load Balancer is a load balancing option for the Elastic Load Balancing service. It operates at the application layer and allows you to define routing rules based on content across multiple services or containers running on one or more Amazon Elastic Compute Cloud (Amazon EC2) instances.

1.2.1 You are looking at the MustacheMeWebServer target group. There are 3 other such groups in this lab. One for each microservice and one for Jenkins. At the top of the window is a filter text field. Click the "**X**" to remove the active filter and allow display of all target groups.

1.2.2 In the pane at the bottom, you can see the port on which this target group will be accessible, the VPC it lives in, the load balancer it is attached to, and some additional configuration and attributes.



1.2.3 Click on **Targets** tab and you will see the list of targets that are part of this group. There should be only one and it should show as healthy.

1.2.4 Speaking of health, go ahead and click on the **Health checks** tab. These are the parameters that are used to determine whether a target should be added or removed from a target group.

1.2.5 Let's go back to the **Description** tab and click on the **Load balancer**.

1.2.6 You are now looking at the Application Load Balancer. In the **Description** tab, you will be informed about the DNS name for this load balancer, whether it is Internet-facing or private, what availability zones it covers, the Security Groups that govern its access and more.

1.2.7 Click on the **Listeners** tab. Here is the part that indicates which traffic is going to go to which target group. You should expand some of these lines so you familiarize yourself with the idea.

1.2.8 Let us go back to the **Description** tab. In the Basic section, highlight and copy the **DNS Name** – discarding the "(A Record)" part.

1.2.9 Go to your browser of choice and paste the DNS name but add ":8000" at the end. You should see the MustacheMe website.

Task 1.3: Adding a Friendly URL

Overview

Port 8000 is not always default for web servers. Hence you should make sure that if your customers come to the website without specifying a specific port they will still be able to enjoy your glorious application.

1.3.1 Make sure you are on the EC2 Console page.

1.3.2 Click on **Load Balancers** under **Load Balancing**. Select your load balancer.

1.3.3 Click on the **Listeners** tab below.

1.3.4 There are four rules. Expand the listener for **Load Balancer Port** 80.

1.3.5 There are 2 rules. For the rule with an empty **Path Pattern** and a **Priority** of Default, click the **Edit** button under **Actions** (all the way to the right). Under the **Target group name** dropdown, change the **target group** to **MustacheMeWebServerTargets**. Make sure to click **Save** under **Actions**. This will create a default route on port 80 for the MustacheMeWebServer target group.

1.3.6 Navigate with your browser to the Load Balancer DNS A record. This time you do not need to add :8000. The update can take a few seconds to take effect.

Task 1.4: Accessing an API via the Load Balancer

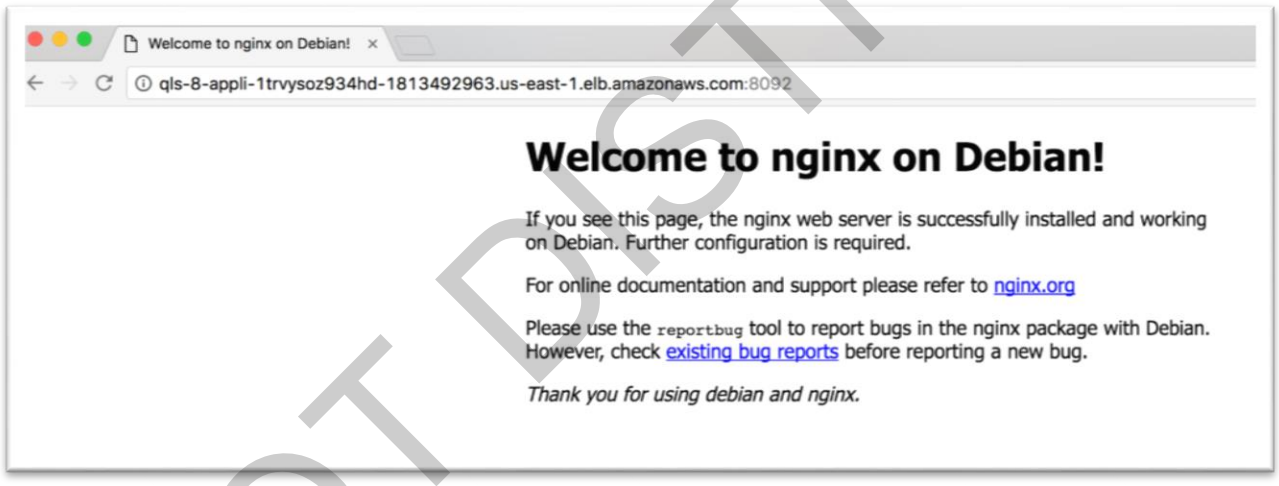
Overview

It is not only the web servers that are accessible via the Application Load Balancer. All your services are going to call each other via the ALB. Hence you can access the Info API microservice by specifying the correct port.

1.4.1 Use your browser to navigate back to the target groups. It is part of the EC2 Web console and can be accessed via **Load Balancing** \ **Target Groups**.

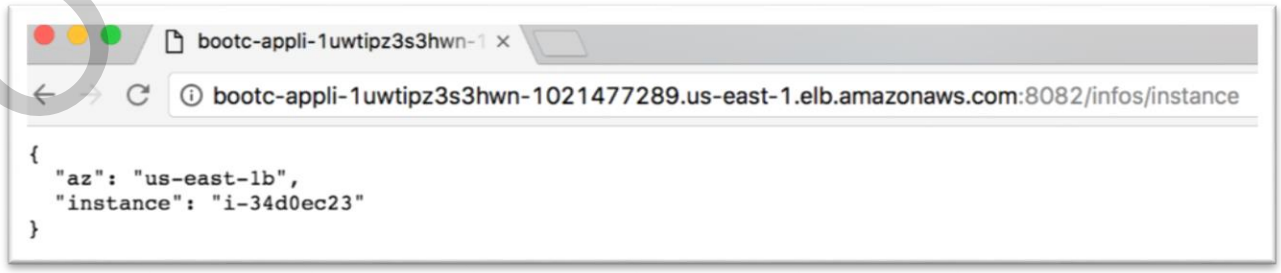
1.4.2 Select the **MustacheMeInfoTargets** target group. You can see that it uses a certain port. Note this port down.

1.4.3 You might still have a browser window opened on the MustacheMe website via the ALB DNS record. If you do not, you should now know how to find this DNS record. In the address bar, combine the ALB DNS name and the Info listening port and go.



1.4.4 You will be shown a default nginx page. Now modify the URL by adding "**/infos/instance**" and go.

You should see a JSON-formatted text displaying some information about the instance where the microservice is running. You have basically made a GET call on the Info API.



Task 2: Availability of Your Services

Overview

How reliable is your application? Can you open the gates to your customers with confidence? You should be in a position where if a container instance fails it can be recovered automatically at no impact to your users.

Scenario

One of the ways to see if a system is resilient is to break it and see if it comes back up. If it does not, what do you need to do to ensure that it does?

Task 2.1: Break a service

You are going to stop a task and see the impact.

2.1.1 Navigate to the ECS Console page.

2.1.2 Click on the **EcsClusterStack** cluster.

2.1.3 Click on the **MustacheMeWebServerStack** service.

2.1.4 You will see a single task running. Click on this Task.

2.1.5 A new page is displayed providing details about the task. There is a **Stop** button on the top right corner. Click on it.

2.1.6 Thankfully you are asked for confirmation. Confirm by clicking the red **Stop** button.

2.1.7 Now go back to the MustacheMe website via the ALB DNS Name and hit refresh on your browser. You might have to use the full page reload feature on your browser that bypasses any available cache.
In Chrome, use Ctrl + F5 or Shift + F5.
In Firefox, use Ctrl + F5.
In Safari, hold the Option key and press the Reload button in the toolbar.
In Internet Explorer, use Ctrl + F5.

2.1.8 You should see that the web server is down. You are shown a 503 error code.

Since this task was launched via a service, ECS will work as quickly as it can to bring up a replacement task. However, since there was only a single task running in this service, a failure of any task in your service means you will not be able to delight your customers with the joy of mustaching while that task is being replaced. You do not offer high availability. Let us remedy this.

Task 2.2: One more task, through the console

You are going to bring high availability to your service by having more than one task running.

2.2.1 Go back to the cluster page for the EcsClusterStack and click on the **MustacheMeWebServerStack** service.

2.2.2 On the top right corner, click the blue **Update** button.

2.2.3 Change the number of tasks to be 2.

2.2.4 Hit **Update Service**.

2.2.5 Hit **View Service**.

2.2.6 It will take a few seconds for the Running count to come to 2, matching the Desired count, in the **Deployments** tab. You can click the refresh button on the top right corner of the table at the bottom to get the last updates.

2.2.7 After the Running count is 2, **the web server is now more highly available**. You should rerun the experiment from the Task 2.1: exercise and see that the web server is always available even when you kill one task. You will see that after stopping a task, no matter how hard you refresh your browser on the MustacheMe site, you always get the front page back. This is a good practice and you should apply it to all your microservices. But before you do so, there is another concern you need to address.

Task 3: Availability of your cluster

Overview

Remember that everything fails all the time. Have you designed your underlying infrastructure for failure? Can you open the gates to your customers with confidence? You should be in a position where if an EC2 instance (container instance) in your ECS cluster fails it can be recovered automatically at no impact to your users.

Scenario

Once again, you are going to break things and see how they react.

Task 3.1: “We have lost compute”

What would happen if the container instance that your microservices are running on suddenly suffers malfunction? It is not too late to find out.

3.1.1 Navigate to the ECS Console page.

3.1.2 Click on the **EcsClusterStack** cluster.

3.1.3 Click on the **ECS Instances** tab.

3.1.4 Click on the one available **Container Instance**.

3.1.5 The page shown informs you about the status of this compute resource. You will see its public and private DNS records, as well as public and private IP addresses. You can also find out all the resource registered and available. Click on the **EC2 Instance** ID. This takes you to the EC2 Console.

3.1.6 Now click on **Actions** \ **Instance State** \ **Terminate**.

3.1.7 Confirm by clicking **Yes, Terminate**.

3.1.8 Go to the MustacheMe website via the ALB DNS Name and hit refresh on your browser. Once again you should use the full page reload feature on your browser that bypasses any available cache.
In Chrome, use Ctrl + F5 or Shift + F5.
In Firefox, use Ctrl + F5.
In Safari, hold the Option key and press the Reload button in the toolbar.
In Internet Explorer, use Ctrl + F5.

3.1.9 You will be shown a 503 error code.

Your website is down. Your customers will have a less than ideal experience. Many faces will go mustache deprived.

3.1.10 Now, still on the EC2 Console, click on **Auto Scaling Groups** in the left column. You might have to scroll down the left column.

If you are shown the Auto Scaling Group dashboard, above the Create Auto Scaling group button there is an **Auto Scaling Groups: <number>** link. Click on it.

3.1.11 One of the Auto Scaling Groups will have EcsClusterStack in the name. You might need to expand the first column to reveal the full name. Select the EcsClusterStack auto scaling group to display the bottom pane. Click on the **Instances** tab. If you have been quick enough you might see an unhealthy instance. Otherwise it is already gone and being replaced.
Our ECS cluster is populated by EC2 instances that are part of an auto scaling group. This means that AWS is going to create as many instances as are desired according to the Launch Configuration.

3.1.12 After a couple of minutes, you will see that a new instance has made it as part of this Auto Scaling Group. Now let us go back to the ECS Console.

3.1.13 Make sure you are looking at the correct cluster and focused in the **Services** tab. Little by little you will see the Running tasks number align with the Desired Tasks (you can hit the refresh button every couple of minutes). Your MustacheMe web app is coming back to life.

3.1.14 You have just simulated an outage. Even though the application recovered in minutes thanks to your Auto Scaling Group and the long-running feature of ECS Services, you should not let your customers suffer even a few minutes of down time.

Task 3.2: Multi-AZ

In accordance with the reliability pillar of the Well Architected Framework you will need to place your EC2 instances across availability zones. Thankfully Auto Scaling Groups are designed to do just this.

3.2.1 Navigate to the EC2 Console.

3.2.2 At the bottom of the left column, click on **Auto Scaling Groups**.

3.2.3 Select the ASG with EcsClusterStack in the name.

3.2.4 Click on the **Activity History** tab.
You will notice that a new EC2 instance was launched just a few seconds ago.

3.2.5 Click on the **Details** tab.
You will see that this ASG has only one instance. Let's fix this.

3.2.6 Click on **Edit**, all the way to the right.

3.2.7 Change **Desired** to 2 and **Max** to 2. Then click **Save**, all the way to the right.

3.2.8 Click on the **Activity History** tab and monitor the status of the ASG. You can hit the refresh button to watch for the status of the new instance launch to become successful.

3.2.9 Once you have made sure that your Auto Scaling group has two instances, return to the ECS Console and navigate to the EcsClusterStack cluster. The **Registered container instances** count should change to 2 once the new instance has registered itself with the cluster shortly after it boots. You may have to hit refresh a couple of times.

3.2.10 Once the **Registered container instances** count is 2, click on the **MustacheMeWebServerStack** service. You know that both tasks are running on the same instance and you need to fix this. You can verify this by clicking each of the **Task** IDs in the table and looking at the **EC2 instance id** in the **Details**.

3.2.11 Go ahead and stop one of the tasks (it doesn't matter which task you pick) while looking at the details by hitting the **Stop** button in the top right corner and confirm by hitting **Stop** again.

3.2.12 Now click on the **ECS Instances** tab from the **MustacheMeWebServerStack** service.

3.2.13 You should be able to infer from the Available CPU on both instances displayed that the service now has two tasks running on two instances. You can also confirm this by looking at each task in the service details and finding the instance ID for both tasks.

Task 4: Ensuring all services are highly available, the right way

Overview

You are now in a much better situation: You have a resilient cluster made of two instances and you know that you can use the Service feature to spread tasks on various hosts. You just need to apply this setup to all your services, and you are going to do this using your CI/CD pipeline.

Scenario

You are going to invoke the power of automation you garnered in the previous lab to apply best practices.

Task 4.1: Deploying a highly available Info service

Let us modify the parameters for one of the services. For this you will need to connect to the CLI instance.

4.1.1 Find the IP or DNS name for the CLI instance. You can find it in the EC2 Console.

4.1.2 SSH to the CLI instance.

```
$ ssh –i <YOURKEY.PEM> ec2-user@<YOUR EC2 INSTANCE PUBLIC DNS NAME>
```

4.1.3 Navigate to the appropriate local repo (git-repos/mustachemeinfo).

```
$ cd ~/git-repos/mustachemeinfo
```

4.1.4 Open microservice.template in your favorite editor.

```
$ nano microservice.template
```

4.1.5 Somewhere in this template is a variable you want to change from 1 to 2 to scale the MustacheMeInfo service from 1 to 2 tasks. Hint: it has “Desired” in the name of the variable.

4.1.6 After your edit is done, save the file and git add, commit and push the change.

```
$ git add * && git commit –m “scaling microservice from 1 to 2 tasks” &&
git push
```

4.1.7 You can visit AWS CodePipeline to see that the change in your code has triggered the desired source collection, Docker image build, and finally deployment.

4.1.8 Once the deployment is complete via the pipeline, you can visit the MustacheMeInfoStack service running on the EcsClusterStack cluster to see that 2 tasks are now powering this service (**Desired count** and **Running count** are both **2**).

Task 4.2: Making it All Highly Available

One resilient service is good. All of them is much better. You know how to change a template so that a service starts with two desired tasks. You should apply this to all your microservices.

4.2.1 Repeat the previous steps to bring all microservices to 2 tasks each.

4.2.2 Do not be alarmed if you see more running tasks than are desired, the deployment mechanism we have chosen has 100% minimum healthy percent and 200% maximum percent. It means that you can increase the total number of running tasks and always keep 100% of the desired tasks running during a deployment.

Task 5: Scaling at the service level

Overview

You are now in a much better situation: You have a resilient cluster populated by resilient services. Let's put it to the test. You made sure that your product would be highly available. But have you planned for growth? One of the key advantages of AWS is that it allows you to efficiently run at a scale that you establish based on the demand from your customers. In this case, you will have to simulate an onslaught of fans and early adopters trying to desperately get mustached after your website was shared on a succession of social websites.

Scenario

As much as you would like to, you are not going to run a Denial of Service on your own services here and see which microservice breaks first. Instead you are going to try to cope with a more mundane and maybe more plausible scenario: one of the team members has pushed flawed code to the repo and one of your microservices now takes a disproportionate amount of memory. The member responsible for this SNAFU has been identified and you will run a blameless post-mortem as soon as possible but for now you simply need to scale the service so that it can still serve your users. You will scale your service based on the memory used by a task.

Task 5.1: Automatically scale a service based on a CloudWatch alarm

The metric you are going to scale on is *TargetResponseTime*. You are going to use an unusually low threshold for scaling, to make sure you see actions based on triggers. Now let us define this auto scaling mechanism.

5.1.1 Got to the EC2 Console.

5.1.2 In the left column, click on **Target Groups** under **Load Balancing**.

5.1.3 Select the **MustacheMeProcessorTargets** Target group.

5.1.4 Click on the **Monitoring** tab in the information section below.

5.1.5 Click on **Create Alarm** on the top right side of the information section.

5.1.6 Uncheck the **Send a notification to** box.

5.1.7 Set the **Whenever** dropdowns to **Average** of **Average Latency**.

5.1.8 Set the **Is >=** to 0.05 seconds.

5.1.9 Leave the **For at least** checking period to be 1 **consecutive period** of 1 Minute and the **Name of the alarm** as the default value (but note the name of the alarm for use below).

5.1.10 Click on **Create Alarm**, then **Close** the dialog box.

5.1.11 Now go to the ECS console.

5.1.12 Click on the **EcsClusterStack** cluster.

5.1.13 Click on the **MustacheMeProcessorStack** service.

5.1.14 Click the **Update** button.

5.1.15 At the bottom left, click on **Configure Service Auto Scaling**.

5.1.16 Select **Configure Service Auto Scaling to adjust your service's desired count**.

5.1.17 Set **Minimum number of tasks** to 2 for high availability.

5.1.18 The **Desired number of tasks** should already be set to 2 from your work in a previous step.

5.1.19 Set the **Maximum number of tasks** to 10.

5.1.20 Select the Bootcamp-created IAM role that is available in the drop down for **IAM role for Service Auto Scaling**.

5.1.21 Click **Add scaling policy**.

5.1.22 Give it a **Policy name** of your choice (no spaces allowed).

5.1.23 For **Execute policy when**, select **Use an existing Alarm** and choose the alarm name created above in the dropdown.

5.1.24 In **Scaling action**, indicate that you want to **Add** 1 more task (when 0.05 should already be set).

5.1.25 Go ahead and reduce the **Cooldown period** to 30 seconds from 300.

5.1.26 Click **Save**.

5.1.27 Make sure that your new scaling policy is listed under the **Automatic task scaling policies**, and click **Save** again.

5.1.28 Review your new service settings and configurations, and click **Update Service**. You should see a list of successfully completed items.

5.1.29 Now go to the CloudWatch Console. Click on **Alarms** on the left. Quick lesson on CloudWatch alarms: your alarm may be sitting in a state of INSUFFICIENT_DATA. This is normal, as CloudWatch alarms will default to a state of INSUFFICIENT_DATA if there are no metrics to report (which there are not since there is no traffic coming to your site at the moment). Head back to your browser, visit the MustacheMe app, and throw some mustaches on a couple of faces. Check back to the CloudWatch Console, wait about a minute, and the alarm status should change to OK (you might need to refresh the alarm status a few times). If the status changes to ALARM, then you've triggered the alarm and have gotten a head start on the next step: simulating load on your microservice!

Task 5.2: Overload your microservice with a few clicks

For the alarm threshold you gave yourself (a response time equal to or greater than 0.05 seconds), you do not need a distributed denial of service to scale your service. All you will need is to use the service a few times. In fact, you may have already triggered the alarm in the steps above. If not, proceed with a bit more mustaching.

5.2.1 Navigate your browser to the ALB A record for MustacheMe.

5.2.2 Reload the page a few times, feed the application a bunch of pictures.

5.2.3 Visit CloudWatch to see if the alarm has been raised and action has been taken. Hint: check under the **History** tab of your alarm.

Task 6: Scaling at the cluster level

Overview

There is not a lot of room (i.e. resources) for your services. There are too many tasks and you need to make room for your services. In other words, you need the cluster to grow based on how many tasks are running. There are a good set of metrics for this. They are the per-cluster MemoryReservation and CPUReservation for example. In this section, you will use CPUReservation.

Scenario

Let's create an architecture such that your cluster always has space for tasks to land on. You are going to make sure to scale the cluster out by adding container instances based on a specific resource.

Task 6.1: Let us start by creating the relevant alarm

Here you will want to base your alarm on overall reserved CPU at the cluster level.

6.1.1 Navigate to the ECS console.

6.1.2 Click on the **EcsClusterStack**.

6.1.3 Click on the **Metrics** tab below.

6.1.4 Click on the **CPUReservation** graph name. This will open a browser link to this particular CloudWatch metric. See if you can create an alarm by yourself. Let us say that you want to trigger the alarm at 70% of your cluster's CPU being reserved for 1 consecutive period of length 1 Minute. Remember that you don't have your **Actions** set up just yet, so be sure to delete any associated actions for now.

Task 6.2: Create a scaling policy

It is time to act on your freshly created CPU reservation alarm.

6.2.1 Navigate to the EC2 console.

6.2.2 Click on **Auto Scaling Groups** at the bottom left.

6.2.3 Select the **EcsClusterStack** group.

6.2.4 Before anything, you need to make sure the Auto Scaling group can scale past 2 instances. Click on the **Details** tab.

6.2.5 Click **Edit**, change **Max** to 4 and hit the **Save** button.

6.2.6 Click on the **Scaling Policies** tab.

6.2.7 Click **Add policy**.

6.2.8 Give your policy a **Name**.

6.2.9 Set the **Execute policy when** dropdown to the alarm you created above.

6.2.10 **Take the action** should be set to **Add** 1 instance **when** 70 <= CPUReservation.

6.2.11 Set **Instance need** to 120 **seconds to warm up**.

6.2.12 Click **Create**.

6.2.13 Take a look at CloudWatch as the resource consumption goes up, you can also check the number of ECS Instances in your cluster (ECS Console) and the **Activity History** tab in the Auto Scaling Group part of the EC2 Console.

Congratulations for completing Lab 3!

Lab 4
Securing Your ECS Application

Overview

This part of the bootcamp is about making sure that the microservices are always secured, following the best practices defined by the security experts, and personalized to our company requirements. Security is AWS priority number, and it should be part of all the company's CI/CD process, to prevent the deployment of insecure applications or environments to production. You will learn how to integrate the security with all the automation presented in previous labs, to prevent risks before the environment is in production.

The lab includes several hands-on exercises:

- Review the Environment
- Secure git repository
- ECS services security best practices
- IAM usage in microservices
- Automated container security

Prerequisites

This lab requires:

- Completion of labs 1, 2, and 3 of Running container-enabled microservices on Amazon ECS.

Duration

This lab should take you about an hour and a half to complete

Task 1: Reviewing the Environment

Overview

The last lab had us deep into scaling microservices dynamically. This part of the 4th lab will get us acquainted with the various parts as they have been laid out.

Scenario

MustacheMe is almost in production. We have a working product and we know that we can iterate on the existing version at great speed with the existing CI/CD pipeline. We're ready for scaling up, and prepared for the failure, now let's assure the security of the application.

Task 1.1: Retrieve the CLI instance DNS name

Overview

The CLI instance we are going to connect to is part of a CloudFormation stack. Let's go find its name.

1.1.1 Point your browser to https://console.aws.amazon.com/cloudformation/home

1.1.2 In the table, you see are "stacks", basically infrastructures that have been spun up according to a json-formattted template. We want to click on the line that has "CliInstanceStack" in its name.

1.1.3 In the lower pane you, click on the Outputs tab.

1.1.4 The instance's DNS is the **value** for which the **key** is "PublicDnsName".

1.1.5 Copy this somewhere (your clipboard/buffer, for example).

Task 1.2: Connect to the instance

1.2.1 Change directory to where your .pem file is located.

1.2.2 SSH into the instance.

```
$ ssh –i <YOURKEY.PEM> ec2-user@<YOUR EC2 INSTANCE PUBLIC DNS NAME>
```


Task 2: Secure git repository

Overview

Developers store all the code in a common repository where all the teams have access to it. An agile environment normally will provide as much visibility as possible to allow your teams to share code and lessons learned during the development. But, if anybody published a secret to the repository, like a password or an access key, it opens a security breach.

We're going to use a tool to analyze the code looking for AWS Access and Secret keys to stop the push to the repository if a developer writes the credentials in the source code. To do it, the tool uses the git hooks, so it analyzes all the code before we push it to the repository.

You can find the tool in GitHub: https://github.com/awslabs/git-secrets

2.1.1 Clone the GitHub repository to the CLI instance:

```
$ git clone https://github.com/awslabs/git-secrets.git
```

2.1.2 Build and install the tool.

```
$ cd /home/ec2-user/git-secrets

$ sudo make install
```

2.1.3 Install the hook to one of the source code repositories, in this case we're going to use MustacheMeInfo, this microservice provides information about the IP and instance where the service is running.

```
$ cd /home/ec2-user/repos/MustacheMeInfo

$ git secrets --install

$ git secrets --register-aws
```

2.1.4 Add an access and secret key to the source code of one of the files, edit the file api-static/serv.py and add these lines in any place inside the code:

```
access_key = 'AKIAI4XJMHBUJWNP2GZQ'

secret_key = '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'
```

2.1.5 Try to push the code to the repository:


```
$ git add -A .

$ git commit -m “Secrets added to the source code”
```

You can see how the tool is analyzing the code before the commit and it alerts you of the secrets in the file, stopping the commit before it can occur:

```
[ec2-user@ip-10-0-1-241 MustacheMeInfo]$ git commit -m "Secrets added to the source code"
api-static/serv.py:27:access_key = 'AKIAI4XJMHBUJWNP2GZQ'
api-static/serv.py:28:secret_key = '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'

[ERROR] Matched one or more prohibited patterns

Possible mitigations:
- Mark false positives as allowed using: git config --add secrets.allowed ...
- Mark false positives as allowed by adding regular expressions to .gitallowed at repository's root directory
- List your configured patterns: git config --get-all secrets.patterns
- List your configured allowed patterns: git config --get-all secrets.allowed
- List your configured allowed patterns in .gitallowed at repository's root directory
- Use --no-verify if this is a one-time false positive
[ec2-user@ip-10-0-1-241 MustacheMeInfo]$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   api-static/serv.py
```


Task 3: ECS Services Security Best Practices

Overview

We're going to review some of the best practices in terms of security. When you design and develop your containers, you should define your own group of best practices and share them with your DevOps team, to ensure you're designing your applications with a standard level of security. We'll provide a tool to run all these best practices (and more), so you can automate the security analysis for your deployments.

The tool we're going to use in this lab is docker-bench-test (https://github.com/gaia-adm/docker-bench-test). This tool analyzes more than 60 security checks by default, and you can personalize it by creating your own checks. The tests are fully automated, are inspired by the CIS Docker 1.11 Benchmark, and based on Docker Bench for Security.

To run the tests, we're going to use a docker image with the tool installed, and launch it from the AWS console using the Run Command functionality. Run Command allows us to run any command in a Windows or Linux instance without any need for connecting to the instance. The output of the command is going to be stored in an S3 bucket for further review.

3.1.1 Open the EC2 web console: https://console.aws.amazon.com/ec2/v2/home

3.1.2 In the left menu, go to the "Command History" option.

3.1.3 As you can see in the console, we've been using this utility to deploy most of the elements of this lab.

3.1.4 Select the "Run a command" button.

3.1.5 As command document, select the "AWS-RunShellScript" option.

3.1.6 Select the two instances with the name "ECS Instance" as target. Launch the test in the two servers of the ECS cluster at the same time.

3.1.7 Write the following command:

```
docker run -e TERM=vt100 --net host --pid host --cap-add audit_control \
    -v /var/lib:/var/lib \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v /usr/lib/systemd:/usr/lib/systemd \
    -v /var/docker-bench-test:/var/docker-bench-test \
    -v /etc:/host/etc \
    --label docker_bench_test \
    awsbootcamp/docker-bench-test -r
cat /var/docker-bench-test/results/tests_latest.tap
```

3.1.8 Write the name of the bucket created in the base cloudformation template. You can retrieve this by going to the Qwiklabs console for the lab and selecting the “Additional Info” tab and copying the value of the parameter “SsmCmdOutputBucket”, or by selecting the Output parameter “SsmCmdOutputBucket” of the Base Cloudformation template.

3.1.9 Write “securitytest” as the S3 key prefix.

3.1.10 Run the command, you can see the progress in the console. When the command is completed, you can see part of the output from the console.

3.1.11 Go to the S3 console, and open the S3 bucket entered in step 3.1.8, then go to the folder “securitytest/<command id>/<instance id>/awsrunShellScript/0.aws:runShellScript” and review the file stdout with the full report for that ECS instance.

You will see a report in the Test Anything Protocol (TAP) format showing compliancy with checks passing starting with “**ok**” and checks failing with “**not ok**”.

Task 4: Using IAM in Microservices

Overview

The most secure credential is the credential that doesn't exist. IAM roles lets you access AWS resources without the need of any kind of credentials, and now, you can use them from a docker container in the ECS service. You can define the elements your application needs to access, like DynamoDB tables, SNS topics or Kinesis streams, and then your application can access those resources without any user or password.

In this lab, we're going to modify the Mustache application to store a log of the images processed in DynamoDB.

4.1.1 Connect to the CLI instance as described in the first task.

4.1.2 Create a new table in DynamoDB to store the log of the images processed, with one field for the name of the image file:

```
$ aws dynamodb create-table --table-name mustache --attribute-definitions
AttributeName=imagekey,AttributeType=S --key-schema
AttributeName=imagekey,KeyType=HASH --provisioned-throughput
ReadCapacityUnits=10,WriteCapacityUnits=10
```

4.1.3 Create the role that we're going to use to access the DynamoDB table:

```
$ aws iam create-role --role-name mustache --assume-role-policy-document
'{"Version": "2012-10-17","Statement": [{"Sid": "","Effect":
"Allow","Principal": {"Service": "ecs-tasks.amazonaws.com"},"Action":
"sts:AssumeRole"}]}'
```

4.1.4 Take note of the arn of the role (something like "arn:aws:iam::xxxxxxxxx:role/mustache")

4.1.5 Attach a policy to the role to allow access to the DynamoDB table created in the step 2.

```
$ aws iam put-role-policy --role-name mustache --policy-name Dynamo --
policy-document '{"Version": "2012-10-17","Statement": [{"Sid":
"","Effect": "Allow","Action": ["dynamodb:*"],"Resource":
"arn:aws:dynamodb:us-east-1:*:table/mustache"}]}'
```


4.1.6 Let's modify the source code of the MustacheMeProcessor microservice to use the role and connect to the DynamoDB table. In the CLI instance change to the directory "/home/ec2-user/repos/MustacheMeProcessor"

4.1.7 First, we need to install the boto3 library to be able to use the AWS SDK, edit the file "Dockerfile", and modify it.

```
# Install supervisord

RUN cd /usr/local/bin && \

        pip install supervisor
```

```
# Install supervisord

RUN cd /usr/local/bin && \

        pip install supervisor && \

        pip install boto3
```

4.1.8 Now, we're going to add the code to insert the element in the table, edit the file "api-static/serv.py", and add this line at the beginning of the file to import the boto 3 library.

```
import boto3
```

4.1.9 In the same file, look for the function "def image_handling", and add the lines of code to create the item in the table. We are not using any kind of authentication, because it's going to be based on the previous role:

```
def image_handling():

    if request.method == 'POST':

        file = request.files['file']

        client = boto3.client('dynamodb', region_name='us-east-1')

        client.put_item(TableName='mustache', Item={'imagekey': {'S':
str(uuid.uuid1())}, 'file': {'S': file.filename}})

        image, count = process_image(file)
```


4.1.10 Now, we assign the role defined in the step 3 and replace the role arn with the arn you took note in the step 4. Edit the file “microservice.yaml” and look for the resource “MustacheMeProcessorTaskDefinition”, and add the property defining the arn (remember to put your own arn):

```
  MustacheMeProcessorTaskDefinition:
    Type: "AWS::ECS::TaskDefinition"
    Properties:
      TaskRoleArn: "arn:aws:iam::<acount-id>:role/mustache"
      ContainerDefinitions:
```

4.1.11 Now, commit the changes and push them to the pipeline.

```
$ git add -A .
$ git commit -m “DynamoDB access”
$ git push origin master
```

4.1.12 Now, wait until CodePipeline build and deploy the microservice. You can try to upload an image to the service (you can find the URL of the application in the output of the CloudFormation template) and see the DynamoDB table content with:

```
$ aws dynamodb scan --table-name mustache
```


Task 5: Using Automated Container Security

Overview

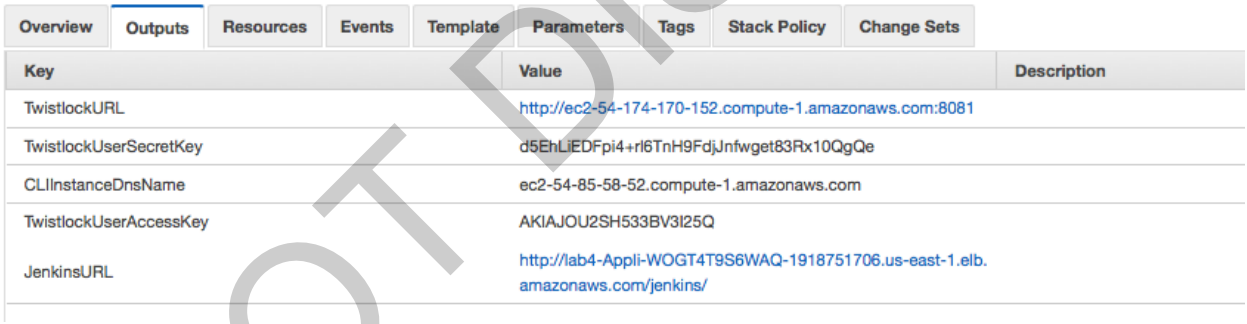
In this lab, we're going to use a tool to analyze the security of your docker images, and docker containers dynamically. You're going to be able to define the rules you want to check, and you can even enforce these rules, and avoid the deployment of containers with security issues.

The tool we're going to use is Twistlock (https://www.twistlock.com/), this tool is made from the ground up to manage the security of a docker environment. You can analyze your running containers to enforce your security policies, and you can also analyze your docker images in ECR to check them against your rules.

Twistlock scans images and registries to detect vulnerabilities in the code, and configuration errors. You can enforce standard configurations, container best practices, and recommended deployment templates.

We're going to analyze the security of our environment, and solve a problem in one of the images used during this workshop.

5.1.1 Connect to the CloudFormation console, and get the Twistlock web url from the outputs of the stack called "Running Container-Enabled Microservices on AWS Bootcamp: Lab 4 - Base CloudFormation Template".



5.1.2 Connect to the URL using your browser, and authenticate with the user "admin" and the password you used to connect to the AWS console.

5.1.3 You can see a dashboard, with a summary of the security information of your ECS environment. Here, you can see the hosts in your ECR cluster, and if you click on any of them, you'll see all the running containers in it.

5.1.4 On the left menu, you have three main options:

(1) Configure: Here, you can configure global system options, or add more hosts to the Twistlock configuration.

(2) Defend: In this menu, you can configure the rules to defend your cluster in real time. When Twistlock detects a vulnerability, it analyzes the rules defined, and decides to either report the issue or block the container. You have access rules (what a docker actions a host can do), runtime rules (File System, Network,

Process and System calls a container can do), and trust rules (Rules for any docker image).

(3) Monitor: In this option, you can see real-time information about the vulnerabilities detected in your environment.

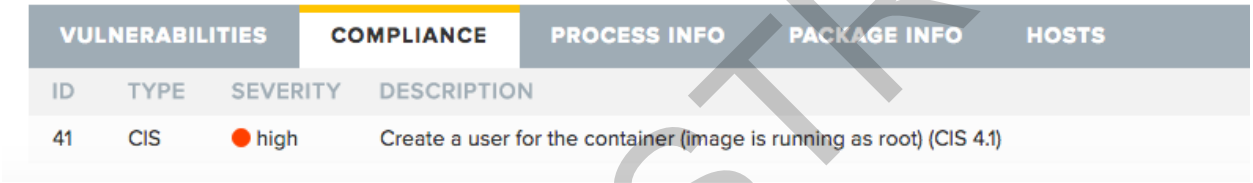
5.1.5 Select the option Monitor→Trust.

5.1.6 Select the tab option “Images”. You can see all the images with the number of vulnerabilities and non-compliance rules detected.

5.1.7 Write “mustachemeprocessor” into the search text box.

5.1.8 Select the version with the Running flag as true. You can see below each one of the vulnerabilities, and non-compliance rules for this image.

5.1.9 Select the Compliance tab, and you’ll see that this image is not compliant with one of the rules:



5.1.10 We’re running the container with the user root, and that’s a security issue, because as the root, you can access the host from the container. So, we’re going to solve this issue.

5.1.11 Connect to the CLI instance using your SSH client.

5.1.12 Change to the directory with the source code of the docker image:

```
$ cd /home/ec2-user/repos/MustacheMeProcessor/
```


5.1.13 Edit the file called: Dockerfile, and add these lines before the CMD final line of the file:

```
EXPOSE 8082

CMD ["/usr/local/bin/supervisord"]
```

```
EXPOSE 8082

RUN useradd -ms /bin/bash nginx && \
        mkdir /var/supervisord && \
        chown nginx /var/supervisord && \
        chown nginx /var/www/forklift/nginx && \
        chown nginx /var/log/nginx && \
        chown nginx /var/lib/nginx && \
        chown nginx /run
USER nginx
WORKDIR /var/supervisord
CMD ["/usr/local/bin/supervisord"]
```

5.1.14 With the “USER nginx” configuration option, instruct the docker to use this user as the user to run the docker process. But first, you need to create it and give the user the necessary permissions to run nginx.

5.1.15 After the problem is fixed, push the microservice to the repository and to production:

```
$ git add -A .

$ git commit -m “Security fix”

$ git push origin master
```

5.1.16 Now, wait until CodePipeline build and deploy the microservice, and you can review the status of the new docker image in Twistlock (click on Scan now to check again the running containers) to see how the compliance error disappears for the running version.

Task 5.2: Secure the docker container (Optional)

In this task, we're going to rewrite the container configuration to solve all the Twistlock detected vulnerabilities.

5.2.1 Use the Jenkins URL you can find in the output of the CloudFormation template to access the Jenkins UI. Gain access using the user "admin" and the same password you used to connect to the AWS console.

5.2.2 Select the MustacheMeProcessorB job, you'll see at least a correct build in the build history. Select the last one. The Twistlock plug-in tells you the number of vulnerabilities found in this container:

5.2.3 Connect to the CLI instance using your SSH client.

5.2.4 Change to the directory with the source code of the docker image:

```
$ cd /home/ec2-user/repos/MustacheMeProcessor/
```


5.2.5 Edit the file called: Dockerfile, and modify these lines at the beginning of the file:

```
FROM awsbootcamp/mustachemebase
```

```
FROM ubuntu:16.10

RUN apt-get -y update && \

    apt-get -y upgrade && \

    apt-get -y -q install gcc python2.7-dev nginx libjpeg-dev \

    zlibc zlib1g-dev zip virtualenv gettext-base python-pip

RUN rm /etc/nginx/sites-enabled/default

RUN mkdir -p /var/www/forklift

RUN cd /var/www/forklift && \

    virtualenv --python=python2.7 venv && \

    . venv/bin/activate && \

    pip install uwsgi numpy geopy flask pillow boto3
```

5.2.6 We're replacing an old image with a lot of vulnerabilities with a new Ubuntu system that has all the packages updated.

5.2.7 After the problem is fixed, push the microservice. During the build in, Jenkins Twistlock will analyze the vulnerabilities of the new image:

```
$ git add -A .

$ git commit -m "Optional Security fix"

$ git push origin master
```

5.2.8 Gain access to the new build in Jenkins and you'll see how the new image doesn't have any more vulnerabilities.

