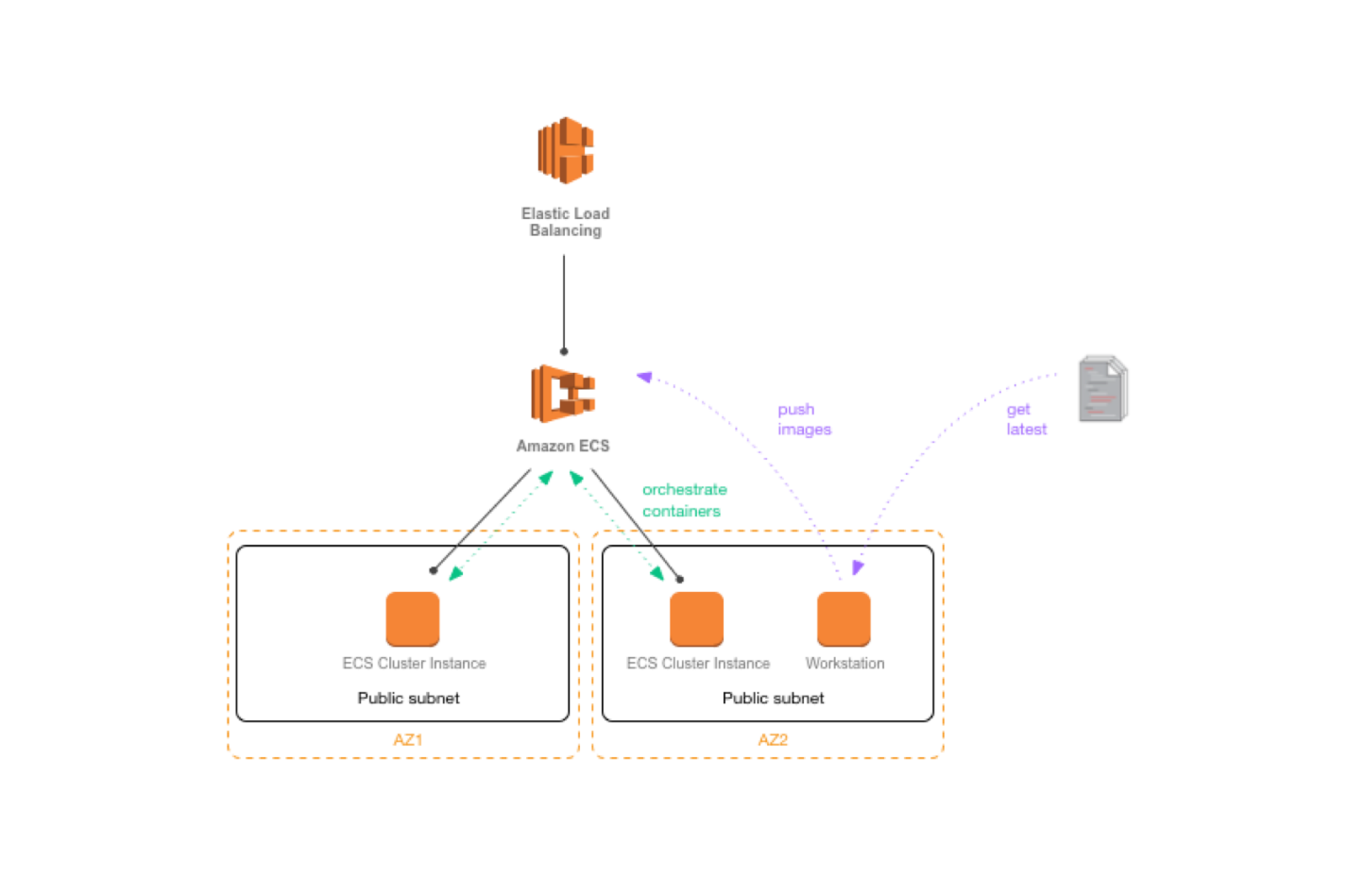
**AWS Container Immersion Day: Lab 1**

## Overview of lab

This lab introduces the basics of working with microservices and [ECS](https://aws.amazon.com/ecs/). This includes: preparing two microservice container images, setting up the initial ECS cluster, and deployment of the containers with traffic routed through an Application Load Balancer ([ALB](https://aws.amazon.com/elasticloadbalancing/applicationloadbalancer/)).

You'll need to [have a working AWS account](https://aws.amazon.com/) to use this lab.

## 1. Setting up the VPC

We will create a new VPC for our entire infrastructure. We need 2 public subnets, for our developer workstation, ECS cluster and the ALB.

**Note**: If students in this lab are using a shared AWS account & VPC, skip this step of creating the VPC. When using a shared AWS account, to avoid confusion and conflicts, be sure to name/tag AWS resources (security groups, IAM roles, instances, clusters, repositories, Docker image tags, etc.), according to your organizations naming conventions or at the very least, choose descriptive names to distinguish your resources from the other students’ resources (i.e. prefix the resource names with your name).

Skip to step 2 if you’re using an existing VPC. Otherwise, configure a VPC with the following requirements:

| **field** | **value** |
| --- | --- |
| Name tag | ECS Lab VPC |
| IPv4 CIDR | 10.0.0.0/16 |
| **Subnet a** |  |
| Name tag | Public subnet a |
| CIDR | 10.0.0.0/24 |
| **Subnet b** |  |
| Name tag | Public subnet b |
| CIDR | 10.0.1.0/24 |

Go to the [VPC Console](https://console.aws.amazon.com/vpc/home):

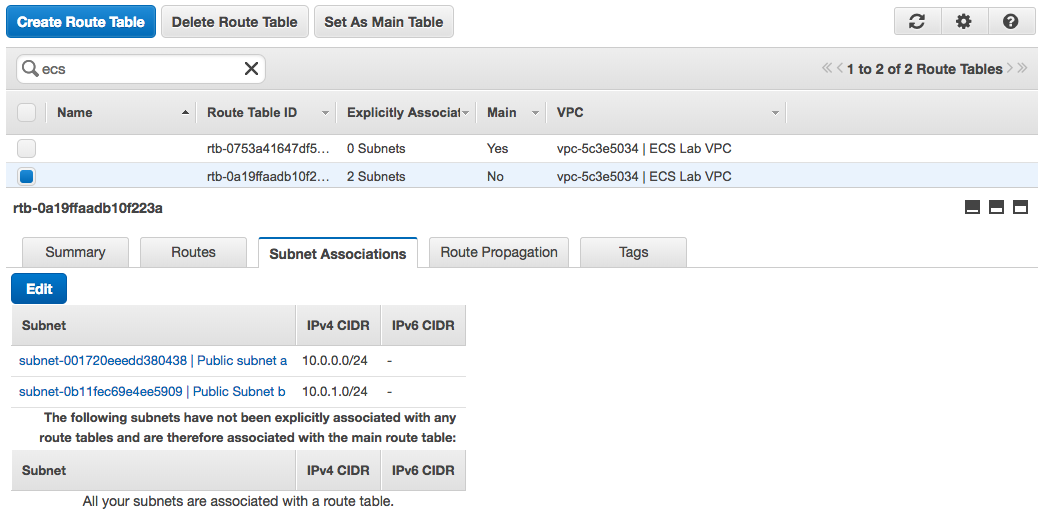
1. click **Start VPC Wizard**
2. Click **Select** on the **VPC with a Single Public Subnet** tab
3. Enter the **IPv4 CIDR block** and **VPC name** (see table above)
4. Enter the **Public subnet’s IPv4 CIDR**
5. Enter the **Subnet name**
6. Choose an availability zone (make note of which you selected)
7. Click **Create VPC**

Create another public subnet:

1. Select **Subnets**
2. Click **Create Subnet**
3. Enter the subnet name (in **Name tag**)
4. Select the **ECS Lab VPC**
5. Select one of the **Availability Zones** that you did **not** use for **Public Subnet a**
6. Enter the **IPv4 CIDR block** for **Public subnet b**
7. Click **Yes, Create**

Configure routing tables

1. Click **Route Tables**
2. Select the route table that is explicitly associated with **Public subnet a**
3. Edit the **Subnet Associations** and ensure that **Public subnet b** is also associated with this route table



## 2. Setting up the IAM user and roles

In order to work with ECS from our workstation, we will need the appropriate permissions for our developer workstation instance. Go to the [IAM Console](https://console.aws.amazon.com/iam/home), **Roles** > **Create Role.** We will later assign this role to our workstation instance:

| **field** | **value** |
| --- | --- |
| Choose Service: | EC2 > EC2 |
| Use Case: | EC2 |
| Attach Policy: | AmazonEC2ContainerRegistryFullAccess |
| Role Name: | EcsLabWorkstationProfile |

Create another new role so that EC2 instances in the ECS cluster have appropriate permissions to access the container registry, auto-scale, etc. We will later assign this role to the EC2 instances in our ECS cluster:

| **field** | **value** |
| --- | --- |
| Choose Service: | EC2 > EC2 |
| Use Case: | EC2 |
| Attach Policy: | AmazonEC2ContainerServiceforEC2Role, AmazonEC2ContainerServiceAutoscaleRole |
| Role Name: | EcsLabInstanceProfile |

**Note**: By default, the ECS first run wizard creates ecsInstanceRole for you to use. However, it's a best practice to create a specific role for your use so that we can add more policies in the future when we need to.

## 3. Launching the Cluster

Next, let’s launch the ECS cluster which will host our container instances. We're going to put these instances in the public subnets since they're going to be hosting public microservices.

Create a new security group by navigating to the [EC2 console](https://console.aws.amazon.com/ec2/) **> Security Groups** and create EcsLabPublicClusterSg. Keep the defaults. Make sure the correct VPC is selected when creating the security group.

Navigate to the [EC2 Container Service console](https://console.aws.amazon.com/ecs/) **> Create Cluster**

| **Field Name** | **Value** |
| --- | --- |
| Cluster Name | EcsLabPublicCluster |
| EC2 instance type | t2.micro |
| Number of instances | 2 |
| EBS storage | 22 |
| Keypair | none |
| VPC | ECS Lab VPC [or name of shared VPC] |
| Subnets | pick the 2 public subnets |
| Security Group | EcsLabPublicClusterSg |
| IAM Role | EcsLabInstanceProfile |

Click Create.

## 4. Launching the Workstation

Next, let’s launch our developer workstation. Think of this as the developer's machine which runs Docker and has access to our Git repository.

Navigate to the [EC2 Console](https://console.aws.amazon.com/ec2) **> Launch Instance**

| **Field Name** | **Value** |
| --- | --- |
| AMI: | Amazon Linux AMI 2017.09.1 [or the latest Amazon Linux AMI] |
| Instance type: | t2.micro |
| Network: | ECS Lab VPC |
| subnet: | one of the public subnets |
| Auto-assign Public IP: | enable |
| Role: | EcsLabWorkstationProfile |
| Next, Storage | (leave default) |
| Next, Tags |  |
| Name: | EcsLabWorkstation |
| Next, Security Group | create a new security group |
| Name: | EcsLabWorkstationSg |
| Inbound rules: | SSH TCP 22 Source: My IP |
| Review and launch | Choose an existing keypair or generate a new one |

Once the instance is running, SSH into it via its public DNS:

$ ssh -i id\_rsa ec2-user@[public DNS]

Update to the latest AWS CLI:

$ sudo yum update -y

Install docker:

$ sudo yum install -y docker

$ sudo service docker start

Add ec2-user to the docker group so you can execute Docker commands without using sudo:

$ sudo usermod -a -G docker ec2-user

Exit and SSH in again to pick up the new permissions.

Verify docker is configured correctly:

**$ docker info**

Containers: 0

Running: 0

Paused: 0

Stopped: 0

Images: 0

Server Version: 17.03.1-ce

Storage Driver: overlay2

Backing Filesystem: extfs

Supports d\_type: true

Native Overlay Diff: true

Logging Driver: json-file

Cgroup Driver: cgroupfs

Plugins:

Volume: local

Network: bridge host macvlan null overlay

Swarm: inactive

Runtimes: runc

Default Runtime: runc

Init Binary: docker-init

containerd version: (expected: 4ab9917febca54791c5f071a9d1f404867857fcc)

runc version: N/A (expected: 54296cf40ad8143b62dbcaa1d90e520a2136ddfe)

init version: N/A (expected: 949e6facb77383876aeff8a6944dde66b3089574)

Security Options:

seccomp

Profile: default

Kernel Version: 4.9.32-15.41.amzn1.x86\_64

Operating System: Amazon Linux AMI 2017.03

…

We now have a working developer workstation.

## 5. Prepping the Docker images

At this point, we're going to pretend that we're the developers of both the web and api microservices, and we will get the latest from our source repo. In this case we will just be using plain old curl, but pretend you're using git:

$ curl -O <https://s3-us-west-2.amazonaws.com/apn-bootcamps/microservice-ecs-2017/ecs-lab-code-20170524.tar.gz>

$ tar -xvf ecs-lab-code-20170524.tar.gz

Our first step is to build and test our containers locally. If you've never worked with Docker before, there are a few basic commands that we'll use in this workshop, but you can find a more thorough list in the [Docker "Getting Started" documentation](https://docs.docker.com/get-started/).

To build your first container, go to the web directory. This folder contains our web Python Flask microservice:

$ cd <path/to/project>/ aws-microservices-ecs-bootcamp/web

To build the container:

$ docker build -t ecs-lab/web .

This should output steps that look something like this:

Sending build context to Docker daemon 4.096 kB

Sending build context to Docker daemon

Step 0 : FROM ubuntu:latest

---> 6aa0b6d7eb90

Step 1 : MAINTAINER widha@amazon.com

---> Using cache

---> 3f2b91d4e7a9

If the container builds successfully, the output should end with something like this:

Removing intermediate container d2cd523c946a

…

Successfully built ec59b8b825de

Successfully tagged ecs-lab/web:latest

To run your container:

$ docker run -d -p 3000:3000 ecs-lab/web

This command runs the image in daemon mode and maps the docker container port 3000 with the host (in this case our workstation) port 3000. We're doing this so that we can run both microservices on a single host without port conflicts.

To check if your container is running:

$ docker ps

This should return a list of all the currently running containers. In this example, it should just return a single container, the one that we just started:

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

7b0d04f4502c ecs-lab/web "python app.py" 9 seconds ago Up 9 seconds 0.0.0.0:3000->3000/tcp eloquent\_noether

To test the actual container output:

$ curl localhost:3000/web

This should return:

<html><head>...</head><body>hi! i'm served via Python + Flask. i'm a web endpoint. ...</body></html>

Repeat the same steps with the api microservice. Change directory to /api and repeat the same steps above:

$ cd ../api

$ docker build -t ecs-lab/api .

$ docker run -d -p 8000:8000 ecs-lab/api

$ curl localhost:8000/api

The API container should return:

{ "response" : "hi! i'm ALSO served via Python + Flask. i'm an API." }

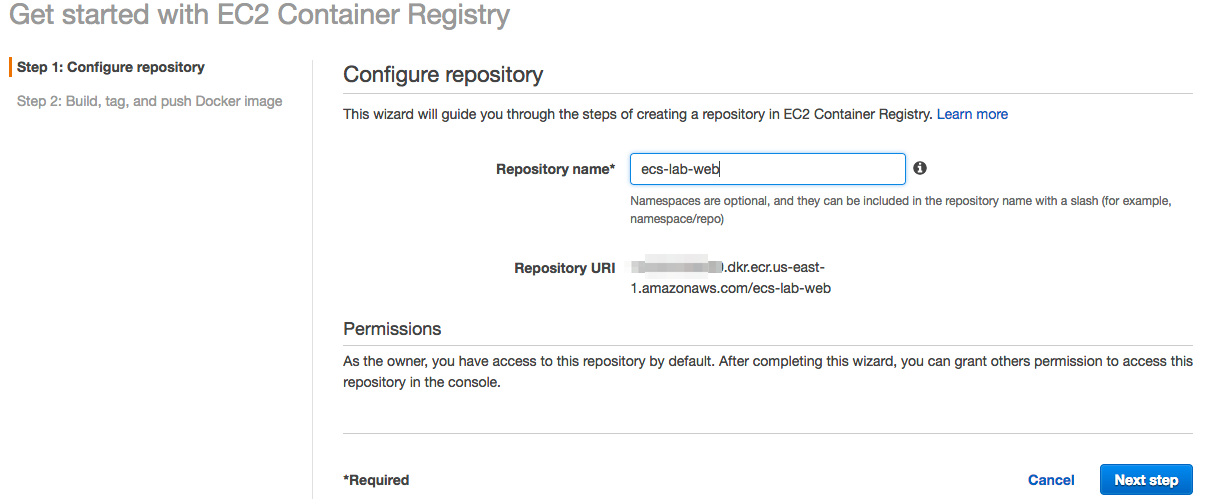
We now have two working microservice containers.

## 6. Creating container registries with ECR

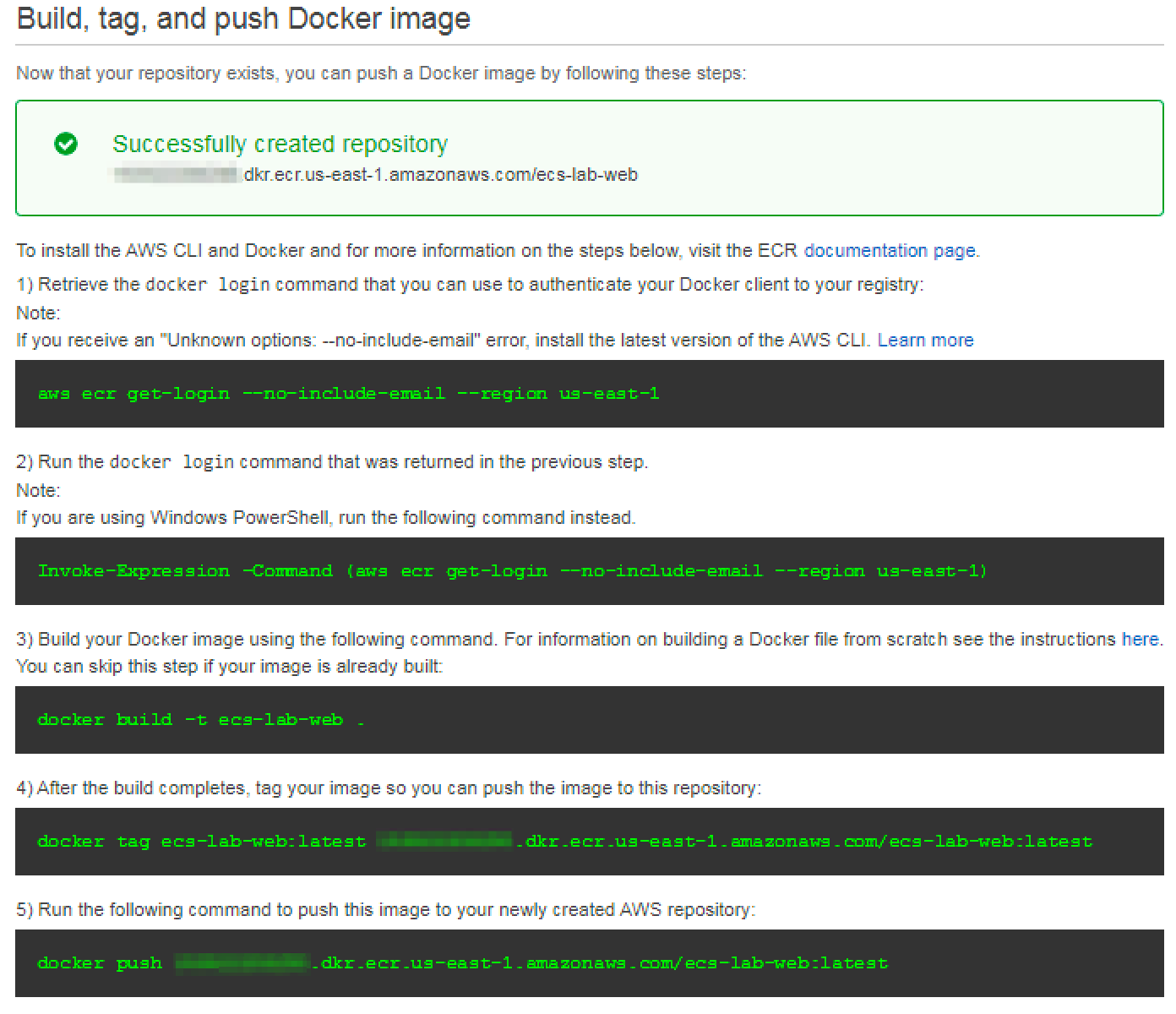
Once images are built, it’s useful to share them and this is done by pushing the images to a container registry. Let’s create two repositories in Amazon EC2 Container Registry ([ECR](https://aws.amazon.com/ecr/)).

Navigate to the [ECS console](https://console.aws.amazon.com/ecs/), and select **Repositories** and choose **Create repository**.

Name your first repository **ecs-lab-web**:



Once you've created the repository, it will display the push commands. Take note of these, as you'll need them in the next step. The push commands should like something like this:



Once you've created the ecs-lab-web repository, repeat the process for the **ecs-lab-api** repository. Take note of the push commands for this second repository. Push commands are unique per repository.

## ­­­­7. Configuring the AWS CLI

On our workstation, we will use the AWS CLI to push images to ECR. Let’s configure the CLI by running:

$ aws configure

This should drop you into a set of prompts. Since our workstation is an EC2 instance pre-configured in an IAM role, the only information required is your preferred region:

$ aws configure

AWS Access Key ID: <leave empty>

AWS Secret Access Key: <leave empty>

Default region name [us-east-2]: us-east-2

Default output format [json]: <leave empty>

You can confirm that your CLI is setup correctly by running the command to obtain an ECR authentication token.

$ aws ecr --no-include-email get-login

This should output something like:

docker login -u AWS -p  -e none https://<account\_id>.dkr.ecr.us-east-2.amazonaws.com

To register ECR as your Docker repository, copy and paste that output or run:

$ `aws ecr get-login –-no-include-email --region us-east-2`

Your shell will execute the output of that command and respond:

Login Succeeded

If you are unable to login to ECR, check your IAM permissions.

## 8. Pushing our tested images to ECR

Now that we've tested our images locally, we need to tag and push them to ECR. This will allow us to use them in Task Definitions that can be deployed to an ECS cluster.

You'll need your push commands that you saw during registry creation. You can find them again by going back to the repository (**ECS Console** > **Repositories** > Select the Repository you want to see the commands for > **View Push Commands)**.

To tag and push to the web repository (if you’re using a shared account, use your name in the tag: fred-ecs-lab:latest):

$ docker tag ecs-lab/web:latest <account\_id>.dkr.ecr.us-east-2.amazonaws.com/ecs-lab-web:latest

$ docker push <account\_id>.dkr.ecr.us-east-2.amazonaws.com/ecs-lab-web:latest

This should return something like this:

The push refers to a repository [<account\_id>.ecr.us-east-2.amazonaws.com/ecs-lab-web] (len: 1)

ec59b8b825de: Image already exists

5158f10ac216: Image successfully pushed

860a4e60cdf8: Image successfully pushed

6fb890c93921: Image successfully pushed

aa78cde6a49b: Image successfully pushed

Digest: sha256:fa0601417fff4c3f3e067daa7e533fbed479c95e40ee96a24b3d63b24938cba8

To tag and push to the api repository:

$ docker tag ecs-lab/api:latest <account\_id>.dkr.ecr.us-east-2.amazonaws.com/ecs-lab-api:latest

$ docker push <account\_id>.dkr.ecr.us-east-2.amazonaws.com/ecs-lab-api:latest

**Note**: why :latest? This is the actual image tag. In most production environments, you'd tag images for different schemes, for example, you might tag the most up-to-date image with :latest, and all other versions of the same container with a commit SHA from a CI job. If you push an image without a specific tag, it will default to :latest, and untag the previous image with that tag. For more information on Docker tags, see the Docker [documentation](https://docs.docker.com/engine/reference/commandline/tag/).

You can see your pushed images by viewing the repository in the [ECS Console](https://console.aws.amazon.com/ecs). Alternatively, you can use the CLI:

$ aws ecr list-images --repository-name=ecs-lab-api

{

"imageIds": [ {

"imageTag": "latest",

"imageDigest": "sha256:f0819d27f73c7fa6329644efe8110644e23c248f2f3a9445cbbb6c84a01e108f"

}

]

}

**Keep all the infrastructure you have built running. You have successfully completed Lab 1. You will be building on this in Lab 2.**