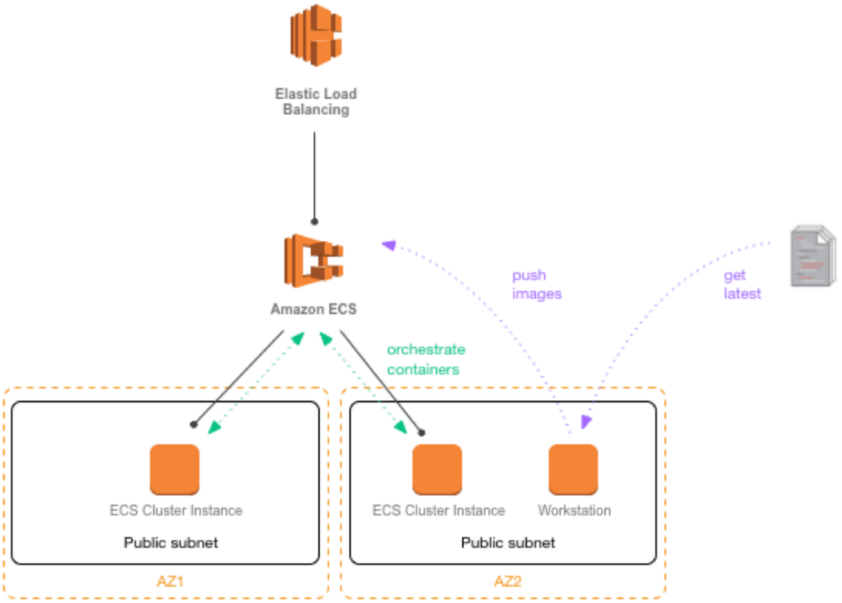
AWS Container Immersion Day: Lab 1

Overview of lab

This lab introduces the basics of working with microservices and ECS. This includes: preparing two microservice container images, setting up the initial ECS cluster, and deployment of the containers with traffic routed through an ALB.



You'll need to have a working AWS account 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.

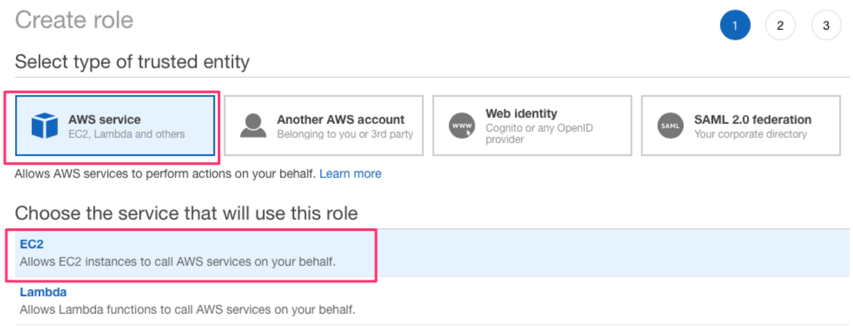
Note: If students in this lab are using a shared AWS account & VPC, skip this step of creating the VPC. When using a shared AWS account, to avoid confusion and conflicts, be sure to name/tag AWS resources (security groups, IAM roles, instances, clusters, repositories, Docker image tags, etc.), according 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

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, **Roles** > **Create New Role > AWS Service > EC2.** We will later assign this role to our workstation instance.



Create role

Select type of trusted entity

Allows AWS services to perform actions on your behalf. Learn more

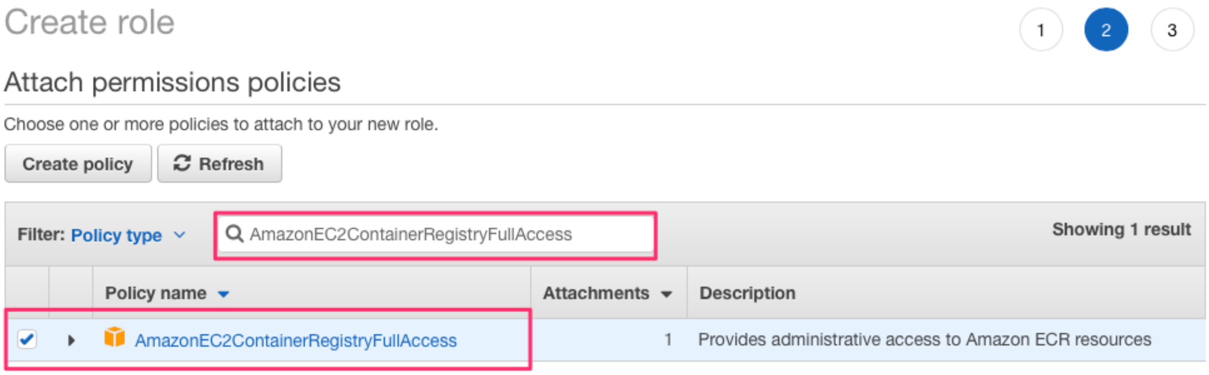
Choose the service that will use this role

EC2
Allows EC2 instances to call AWS services on your behalf.

Lambda
Allows Lambda functions to call AWS services on your behalf.

Click **Next.**

Enter **AmazonEC2ContainerRegistryFullAccess** in the Filter text field.



Create role

Attach permissions policies

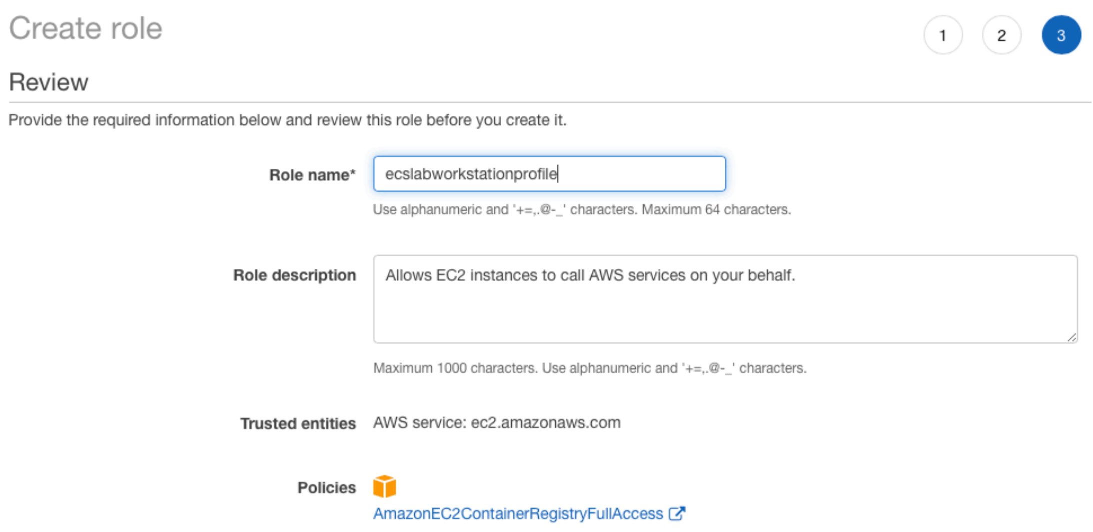
Choose one or more policies to attach to your new role.

Filter: Policy type AmazonEC2ContainerRegistryFullAccess Showing 1 result

		Policy name	Attachments	Description
☑	▸	AmazonEC2ContainerRegistryFullAccess	1	Provides administrative access to Amazon ECR resources

Click **Next.**

Enter **ecslabworkstationprofile** for the Role name and click **Create Role**.



Create role

Review

Provide the required information below and review this role before you create it.

Role name* ecslabworkstationprofile

Use alphanumeric and '+=,.@-_' characters. Maximum 64 characters.

Role description Allows EC2 instances to call AWS services on your behalf.

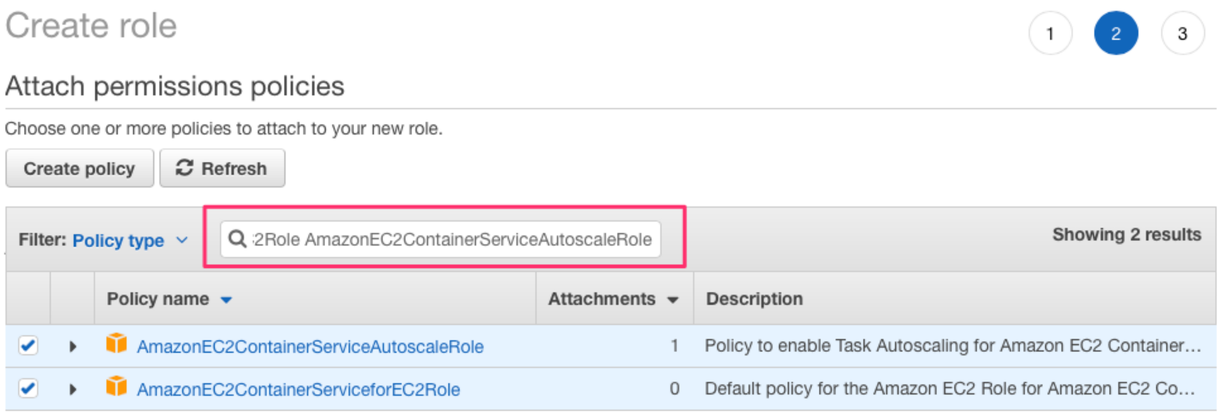
Maximum 1000 characters. Use alphanumeric and '+=,.@-_' characters.

Trusted entities AWS service: ec2.amazonaws.com

Policies AmazonEC2ContainerRegistryFullAccess

Use the same process to 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.

In the Create Role screen, enter AmazonEC2ContainerServiceforEC2Role AmazonEC2ContainerServiceAutoscaleRole in the text field (without a comma) and select the two policies.



Create role

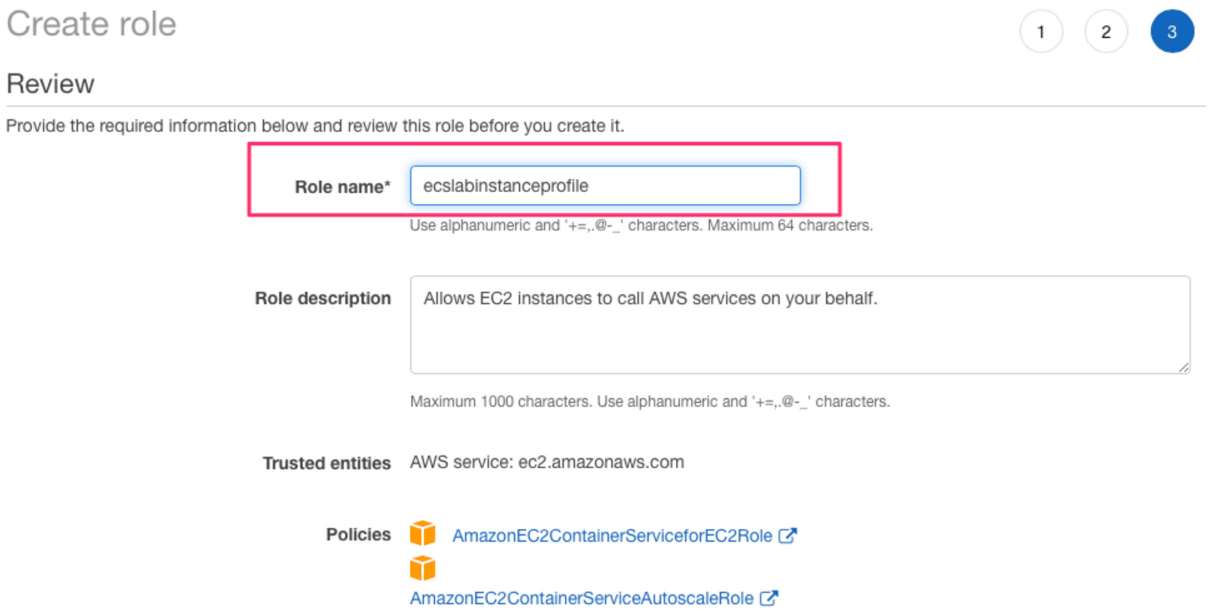
Attach permissions policies

Choose one or more policies to attach to your new role.

Filter: Policy type 2Role AmazonEC2ContainerServiceAutoscaleRole Showing 2 results

		Policy name	Attachments	Description
☑	▸	AmazonEC2ContainerServiceAutoscaleRole	1	Policy to enable Task Autoscaling for Amazon EC2 Container…
☑	▸	AmazonEC2ContainerServiceforEC2Role	0	Default policy for the Amazon EC2 Role for Amazon EC2 Co…

In the Review screen, enter **ecslabinstanceprofile** for the Role name and click **Create Role**.



Create role

1 2 3

Review

Provide the required information below and review this role before you create it.

Role name* ecslabinstanceprofile

Use alphanumeric and '+=,.@-_' characters. Maximum 64 characters.

Role description Allows EC2 instances to call AWS services on your behalf.

Maximum 1000 characters. Use alphanumeric and '+=,.@-_' characters.

Trusted entities AWS service: ec2.amazonaws.com

Policies AmazonEC2ContainerServiceforEC2Role

AmazonEC2ContainerServiceAutoscaleRole

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.

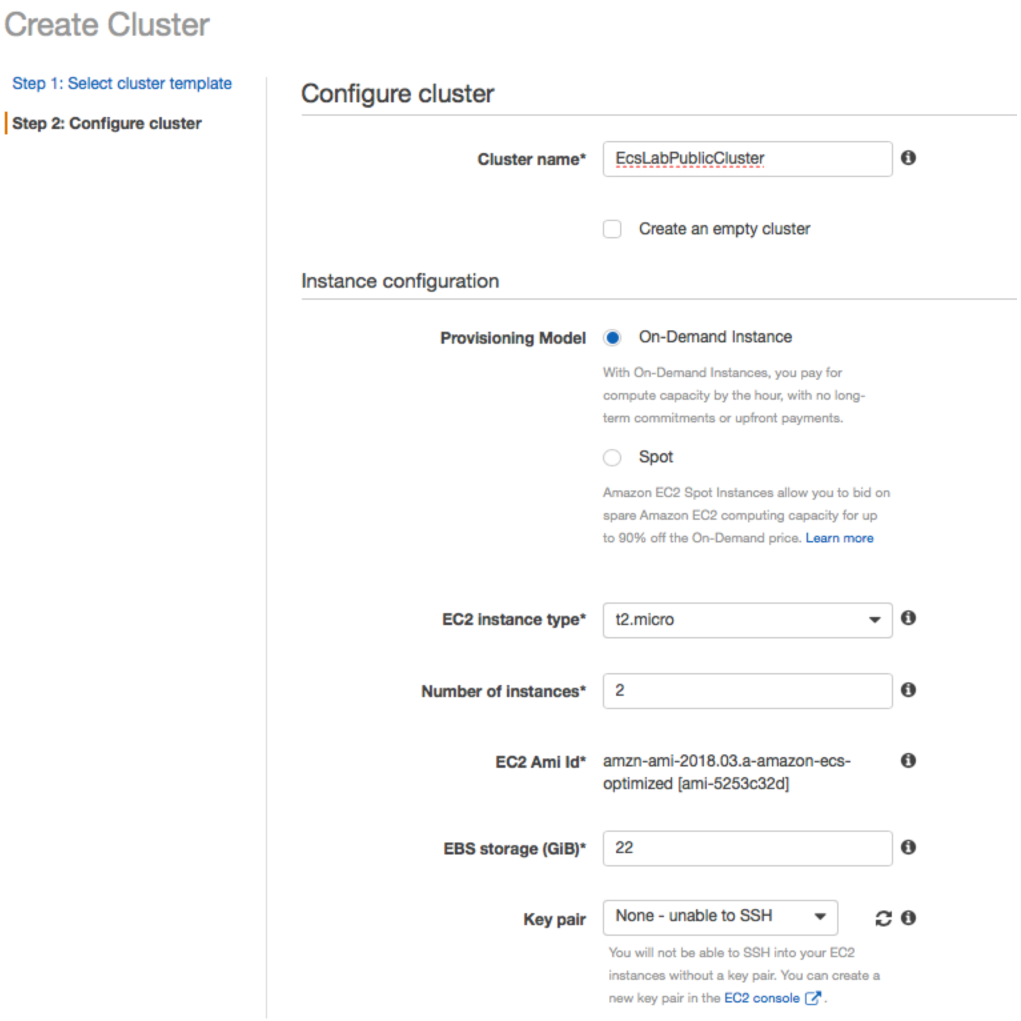
Create a new security group by navigating to the EC2 console > Security Group and create `sgecslabpubliccluster`. Keep the defaults. Make sure the correct VPC is selected when creating the security group.

Navigate to the ECS console and click Create Cluster. Choose the **EC2 Linux + Networking** cluster template. Click **Next Step**.

In the next screen, configure the cluster as follows:

Field Name	**Value**
Cluster Name	`EcsLabPublicCluster`
Provisioning Model	`On-Demand Instance`
EC2 instance type	`t2.micro`
Number of instances	`2`
EBS storage	`22`
Keypair	`none`
Networking Section	
VPC	`ECS Lab VPC [or name of shared VPC]`
Subnets	pick 2 public subnets
Security Group	`sgecslabpubliccluster`
IAM Role	`ecslabinstanceprofile`

Click **Create**. It will take a few minutes to create the cluster.



Create Cluster

Step 1: Select cluster template

Step 2: Configure cluster

Configure cluster

Cluster name* EcsLabPublicCluster

□ Create an empty cluster

Instance configuration

Provisioning Model

● On-Demand Instance

With On-Demand Instances, you pay for compute capacity by the hour, with no long-term commitments or upfront payments.

○ Spot

Amazon EC2 Spot Instances allow you to bid on spare Amazon EC2 computing capacity for up to 90% off the On-Demand price. Learn more

EC2 instance type* t2.micro

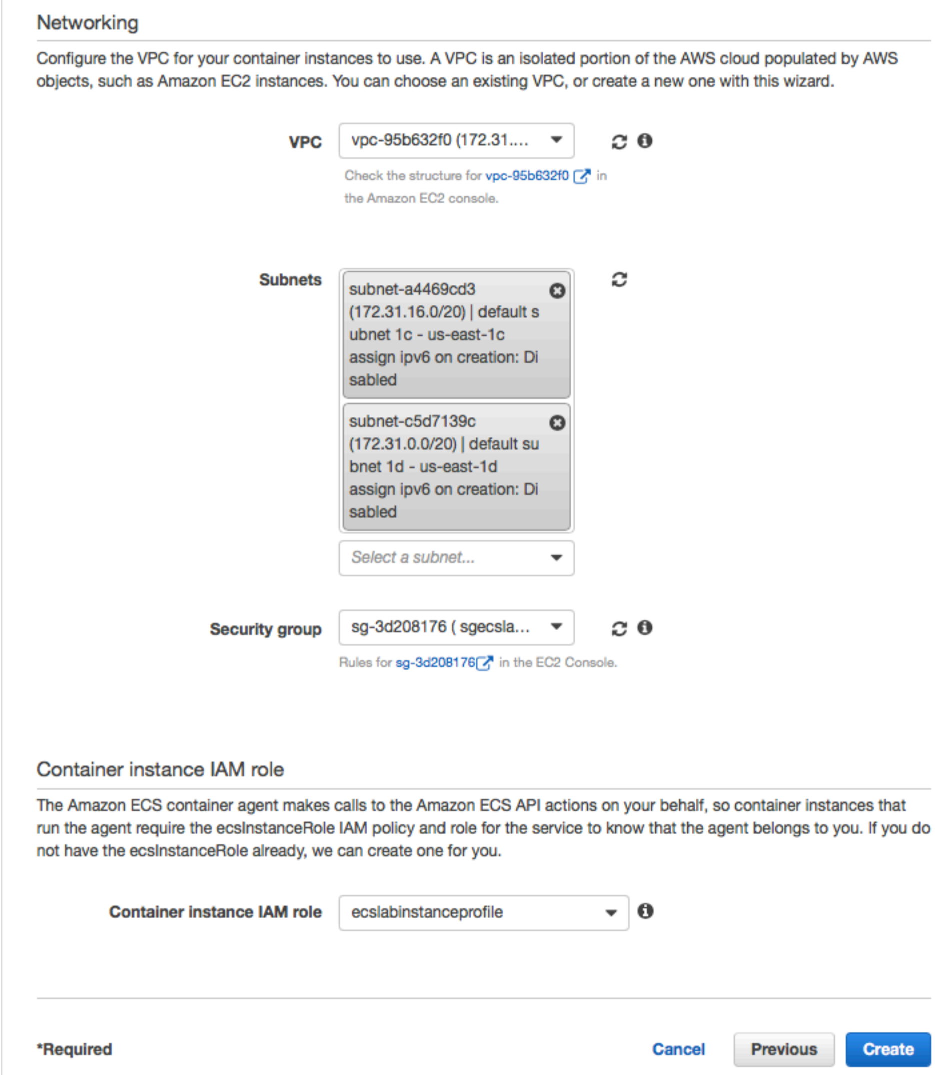
Number of instances* 2

EC2 Ami Id* amzn-ami-2018.03.a-amazon-ecs-optimized [ami-5253c32d]

EBS storage (GiB)* 22

Key pair None - unable to SSH

You will not be able to SSH into your EC2 instances without a key pair. You can create a new key pair in the EC2 console .



Networking

Configure the VPC for your container instances to use. A VPC is an isolated portion of the AWS cloud populated by AWS objects, such as Amazon EC2 instances. You can choose an existing VPC, or create a new one with this wizard.

VPC: vpc-95b632f0 (172.31....

Check the structure for vpc-95b632f0 in the Amazon EC2 console.

Subnets:

subnet-a4469cd3 (172.31.16.0/20) | default subnet 1c - us-east-1c assign ipv6 on creation: Disabled

subnet-c5d7139c (172.31.0.0/20) | default subnet 1d - us-east-1d assign ipv6 on creation: Disabled

Select a subnet...

Security group: sg-3d208176 (sgecsla...

Rules for sg-3d208176 in the EC2 Console.

Container instance IAM role

The Amazon ECS container agent makes calls to the Amazon ECS API actions on your behalf, so container instances that run the agent require the ecsInstanceRole IAM policy and role for the service to know that the agent belongs to you. If you do not have the ecsInstanceRole already, we can create one for you.

Container instance IAM role: ecslabinstanceprofile

*Required

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 **> Launch Instance**

Field Name	Value
Step 1: AMI:	Amazon Linux AMI 2018.03.0 (HVM) [or the latest Amazon Linux AMI]
Step 2: Instance type:	t2.micro
Step 3: Configure Instance Details	
Network:	`ECS Lab VPC or your shared VPC`
Subnet:	one of the public subnets
Auto-assign Public IP:	enable
IAM Role:	`ecslabworkstationprofile`
Next, Step 4: Storage	(leave default)
Next, Step 5:Tags	Add Tag
Name:	`ecs-lab-workstation`
Next, Step 6:Security Group	create a new security group
Name:	`sgecslabworkstation`
Inbound rules:	`SSH TCP 22 Source: My IP`
Step 7: 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 cert.pem 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 the plain old `curl`, but just 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.

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-v2/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
```

To view the image that was just built:

```
 $ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ecs-lab/web         latest              2b849343f6be        13 seconds ago      452MB
ubuntu              latest              113a43faa138        12 days ago         81.2MB
```

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 images
$ 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." }
```

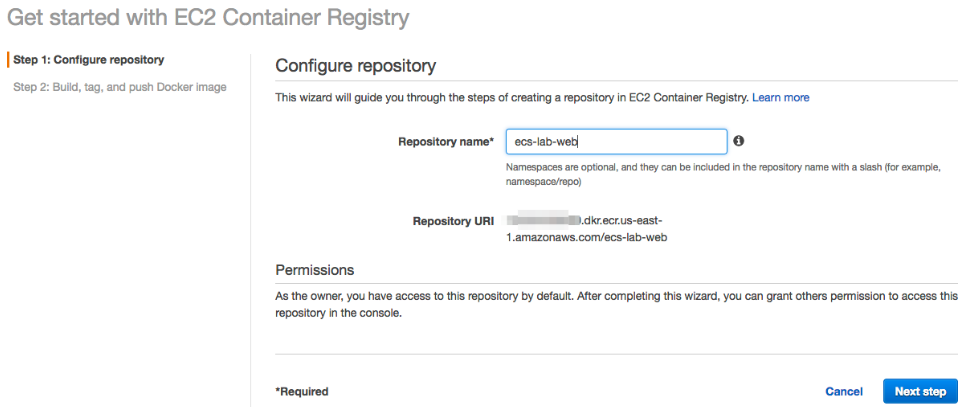
We now have two working microservice containers.

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).

Navigate to the ECS console, and select **Repositories** and choose **Create repository**.

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



Get started with EC2 Container Registry

Step 1: Configure repository

Step 2: Build, tag, and push Docker image

Configure repository

This wizard will guide you through the steps of creating a repository in EC2 Container Registry. Learn more

Repository name* ecs-lab-web

Namespaces are optional, and they can be included in the repository name with a slash (for example, namespace/repo)

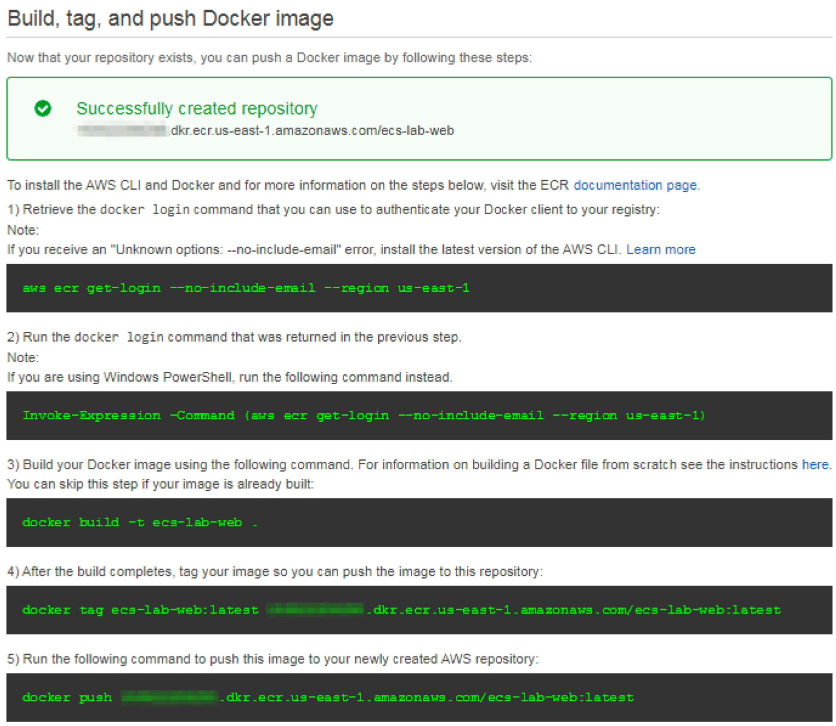
Repository URI [redacted].dkr.ecr.us-east-1.amazonaws.com/ecs-lab-web

Permissions

As the owner, you have access to this repository by default. After completing this wizard, you can grant others permission to access this repository in the console.

*Required

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:



Build, tag, and push Docker image

Now that your repository exists, you can push a Docker image by following these steps:

Successfully created repository
[redacted].dkr.ecr.us-east-1.amazonaws.com/ecs-lab-web

To install the AWS CLI and Docker and for more information on the steps below, visit the ECR documentation page.

1) Retrieve the `docker login` command that you can use to authenticate your Docker client to your registry:
Note:
If you receive an "Unknown options: --no-include-email" error, install the latest version of the AWS CLI. Learn more

```
aws ecr get-login --no-include-email --region us-east-1
```

2) Run the `docker login` command that was returned in the previous step.
Note:
If you are using Windows PowerShell, run the following command instead.

```
Invoke-Expression -Command (aws ecr get-login --no-include-email --region us-east-1)
```

3) Build your Docker image using the following command. For information on building a Docker file from scratch see the instructions here.
You can skip this step if your image is already built:

```
docker build -t ecs-lab-web .
```

4) After the build completes, tag your image so you can push the image to this repository:

```
docker tag ecs-lab-web:latest [redacted].dkr.ecr.us-east-1.amazonaws.com/ecs-lab-web:latest
```

5) Run the following command to push this image to your newly created AWS repository:

```
docker push [redacted].dkr.ecr.us-east-1.amazonaws.com/ecs-lab-web:latest
```

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-1]: us-east-1
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 get-login
```

This should output something like:

```
docker login -u AWS -p
AQECAHhwm0YaISJeRtJm5n1G6uqeekXuoXXPe5UFce9Rq8/14wAAAy0
wggMpBgkqhkiG9w0BBwagggMaMIIDFgIBADCCAw8GCSqGSIb3DQEHAT
AeBglghkgBZQMEAS4wEQQM+76slnFaYrrZwLJyAgEQgIIC4LJKIDmvE
DtJyr7jO661//6sX6cb2jeD/RP0IA03wh62YxFKqwRMk8gjOAc89ICx
lNxQ6+cvwjewi+8/W+9xbv5+PPWfwGSAXQJSHx3IWfrbca4WSLXQf2B
Dq0CTtDc0+payiDdsXdR8gzvyM7YWIcKzgcRVjOjjoLJpXemQ9liPWe
4HKp+D57zCcBvgUk131xCiwPzbmGTZ+xtE1GPK0tgNH3t9N5+XA2BYY
hXQzkTGISVGGL6Wo1tiERz+WA2aRKE+Sb+FQ7YDDRDtOGj4MwZ3/uMn
OZDcwu3uUfrURXdJVddTEdS3jfo3d7yVWhmXPet+3qwkISstIxG+V6I
IzQyhtq3BXW/I7pwZB9ln/mDNlJVRh9Ps2jqoXUXg/j/shZxBPm33LV
+MvUqiEBhkXa9cz3AaqIpc2gXyXYN3xgJUV7OupLVq2wrGQZWPVoBvH
Pwrt/DKsNs28oJ67L4kTiRoufye1KjZQAi3FIPtMLcUGjFf+ytxzEPu
TvUk4Xfoc4A29qp9v2j98090Qx0CHD4ZKyj7bIL53jSpeeFDh9EXube
qp6idIwG9SpIL9AJfKxY7essZdk/0i/e4C+481XIM/IjiVkh/ZsJzuA
PDIpa8fPRa5Gc8i9h0bioSHgYIpMlRkVmaAqH/Fmk+K00yG8USOAYtP
6BmsFUvkBqmRtCJ/Sj+MHs+BrSP7VqPbO1ppTWZ6avl43DM0blG6W9u
IxKC9SKBAqvPwr/CKz2LrOhyqn1WgtTXzaLFEd3ybilqhrcNtS16I5S
FVI2ihmNbP3RRjmBeA6/QbreQsewQOfSk1u35YmwFxloqH3w/lPQrY1
OD+kySrlGvXA3wupq6qlphGLEWeMC6CEQQKSiWbbQnLdFJazuwRUjSQ
lRvHDbe7XQTXdMzBZoBcC1Y99Kk4/nKprty2IeBvxPg+NRzg+1e0lkk
qUu31oZ/AgdUcD8Db3qFjhXz4QhIZMGFogiJcmo= -e none
https://<account_id>.dkr.ecr.us-east-1.amazonaws.com
```

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

```
$ `aws ecr get-login --region us-east-1`
```

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-1.amazonaws.com/ecs-lab-web:latest
$ docker push <account_id>.dkr.ecr.us-east-1.amazonaws.com/ecs-lab-web:latest
```

This should return something like this:

```
The push refers to a repository [<account_id>.ecr.us-
east-1.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:fa0601417fff4c3f3e067daa7e533fbed479c95e40ee96a2
4b3d63b24938cba8
```

To tag and push to the api repository:

```
$ docker tag ecs-lab/api:latest <account_id>.dkr.ecr.us-east-1.amazonaws.com/ecs-lab-api:latest
$ docker push <account_id>.dkr.ecr.us-east-1.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.

You can see your pushed images by viewing the repository in the ECS Console. Alternatively, you can use the CLI:

```
$ aws ecr list-images --repository-name=ecs-lab-api
{
    "imageIds": [        {
            "imageTag": "latest",
            "imageDigest": "sha256:f0819d27f73c7fa6329644efe8110644e23c248f2f3a9445cbbb6c84a01e108f"
        }
    ]
}
```

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