YEVGENIY BRIKMAN

Running Docker on AWS from the ground up

Nov 11, 2015 DevOps, Software Engineering 36 Comments 40 min read

Docker is an awesome tool. In a previous post, I showed how you can use it to package your code so that it runs exactly the same way in development and in production. But how, exactly, do you run Docker in production? Most of the articles I found online assume you're already an expert in both Docker deployment and cloud providers. They don't take the time to explain their ideas from first principles and instead dive straight into jargon like clusters, auto scaling groups, scheduling, nodes, orchestration, PaaS, IaaS, and so on.

In this post, I'm going to introduce Docker deployment from the ground up, using Amazon Web Services (AWS) for hosting. I picked AWS because it's incredibly popular, offers a free tier you can use to try this tutorial at no cost, and provides first-class Docker support via the EC2 Container

Service (ECS). It took me longer than I want to admit to get Docker working on AWS, in no small part because the AWS docs use a lot of jargon (although Amazon Web Services in Plain English does help), so my goal is to make this tutorial accessible to both AWS deployment newbies and Docker deployment newbies (note: if you're a Docker *development* newbie, you should first read A productive development environment with Docker on OS X). Once you're done with this post, check out Infrastructure as code: running microservices on AWS using Docker, Terraform, and ECS for a discussion of how to automate this deployment process.

I'll start the tutorial by showing the most basic way of manually deploying a Docker container on a single server in AWS, then talk about how to manage multiple servers and containers using ECS, and finally, discuss the advantages and disadvantages of ECS, as well as possible alternatives. It's a fairly long post, so here is the table of contents so you can jump to the section you're interested in:

Deploying Docker containers manually

Let's start by manually firing up a server in AWS, manually installing Docker on it, and manually running a Docker container on it. For this

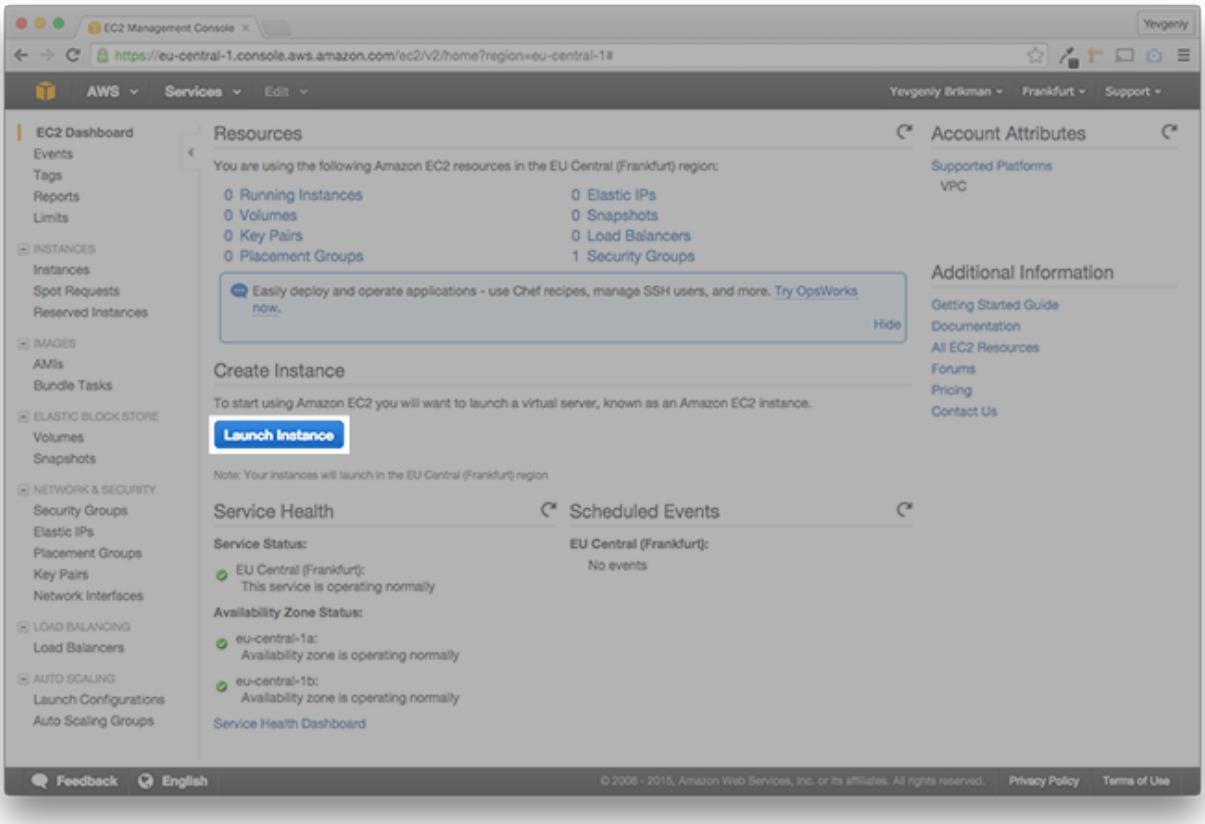
tutorial, the Docker image I'm going to use is training/webapp, which you can use to fire up a simple web server that listens on port 5000 and responds with `Hello, World`:

```
> docker run -d -p 5000:5000 training/webapp:latest python app.py
> curl http://dockerhost:5000
Hello world!
```

How do you run this Docker image on AWS? Well, first you need a server. To do that, you can use the AWS Elastic Compute Cloud (EC2). EC2 makes it easy to boot a virtualized server—called an *EC2 Instance*—with just a few clicks.

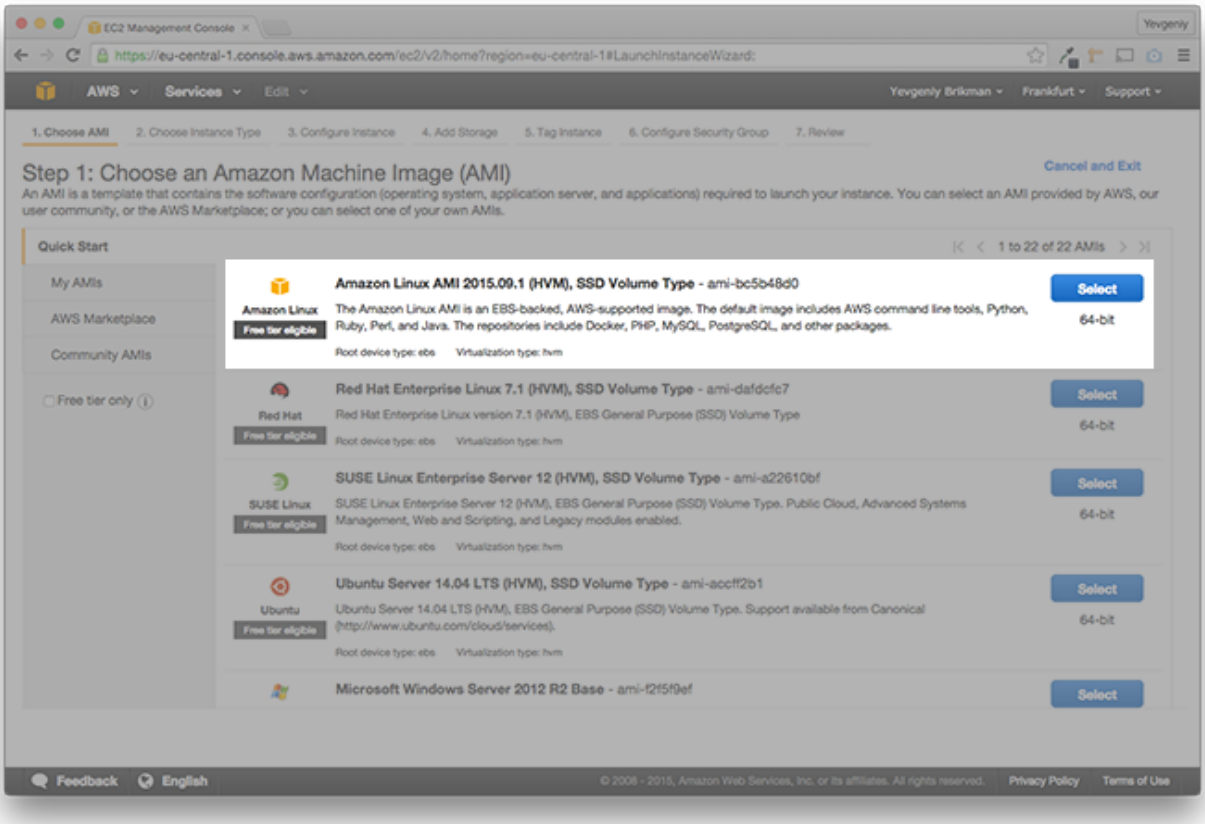
Launching an EC2 Instance

Log into your AWS Console, click the EC2 link to go to the EC2 Console, and click the blue "Launch Instance" button:



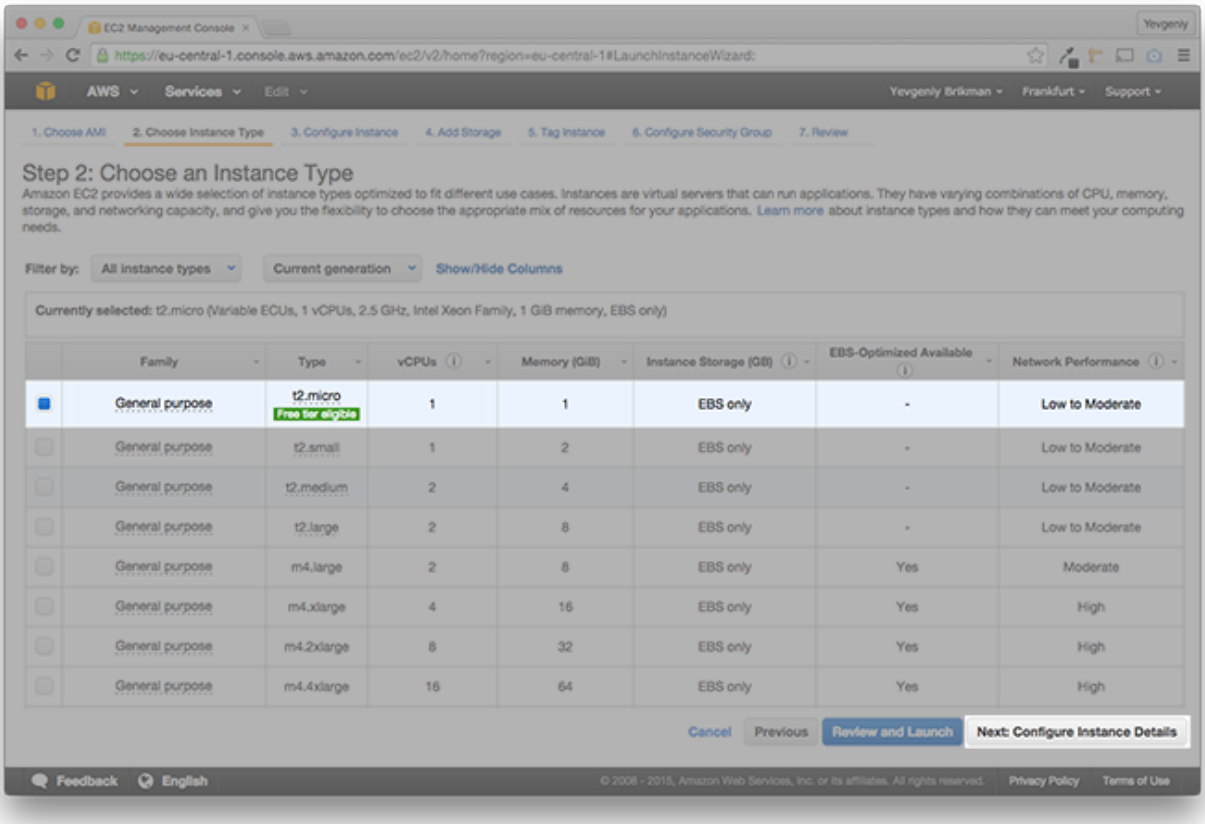
EC2 Dashboard

On the next page, you need to pick an Amazon Machine Image (AMI) to run on your EC2 Instance. The AMI contains the software configuration (operating system, application server, and applications) that will be launched on your server. AWS offers many free and paid options, such as AMIs with Ubuntu, Windows, or MySQL pre-installed. For this tutorial, just pick the top option, which is the `Amazon Linux AMI`:



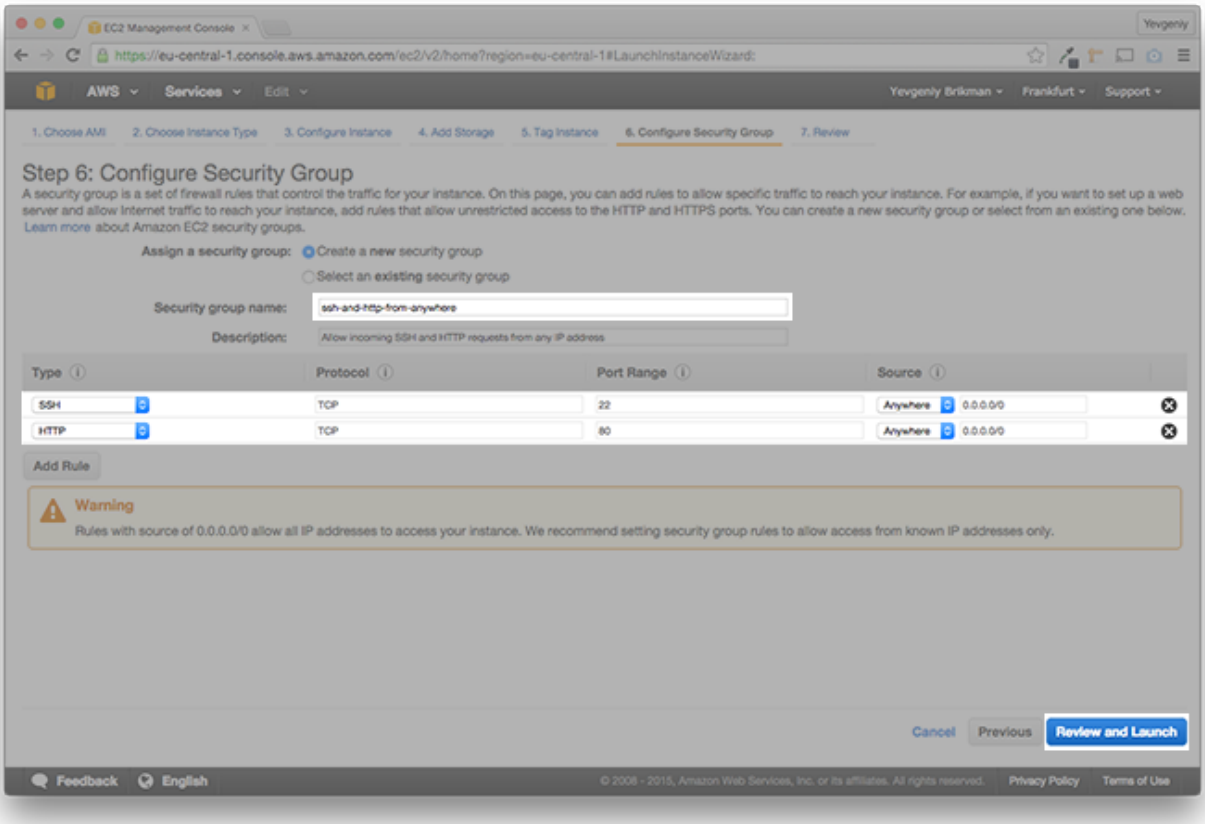
Pick an AMI

Next, you need to pick the Instance Type, which determines what kind of CPU, memory, storage, and network capacity your server will have. Stick with the default option, `t2.micro`, and click the gray "Next: Configure Instance Details" button:



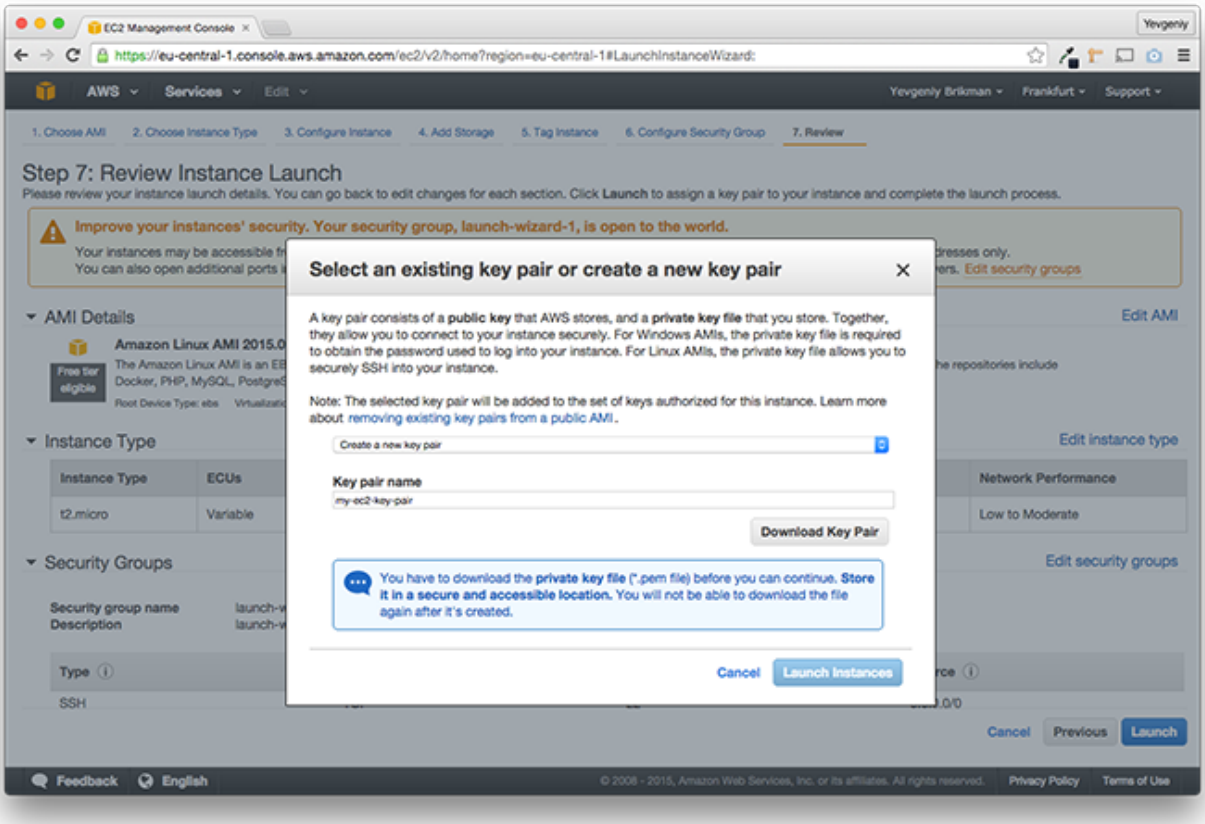
Pick an Instance Type

You can keep the default options for Instance Details, Storage, and Tags, so keep clicking the gray "Next" button until you get to the "Configure Security Group" page. A Security Group is a set of firewall rules that control network traffic for your instance. By default, all incoming ports are blocked, so use this page to add rules that allow incoming SSH (TCP, port 22) and HTTP (TCP, port 80) requests from any source (`0.0.0.0/0`). Give the Security Group a name such as `ssh-and-http-from-anywhere`, and click the blue "Review and Launch" button:



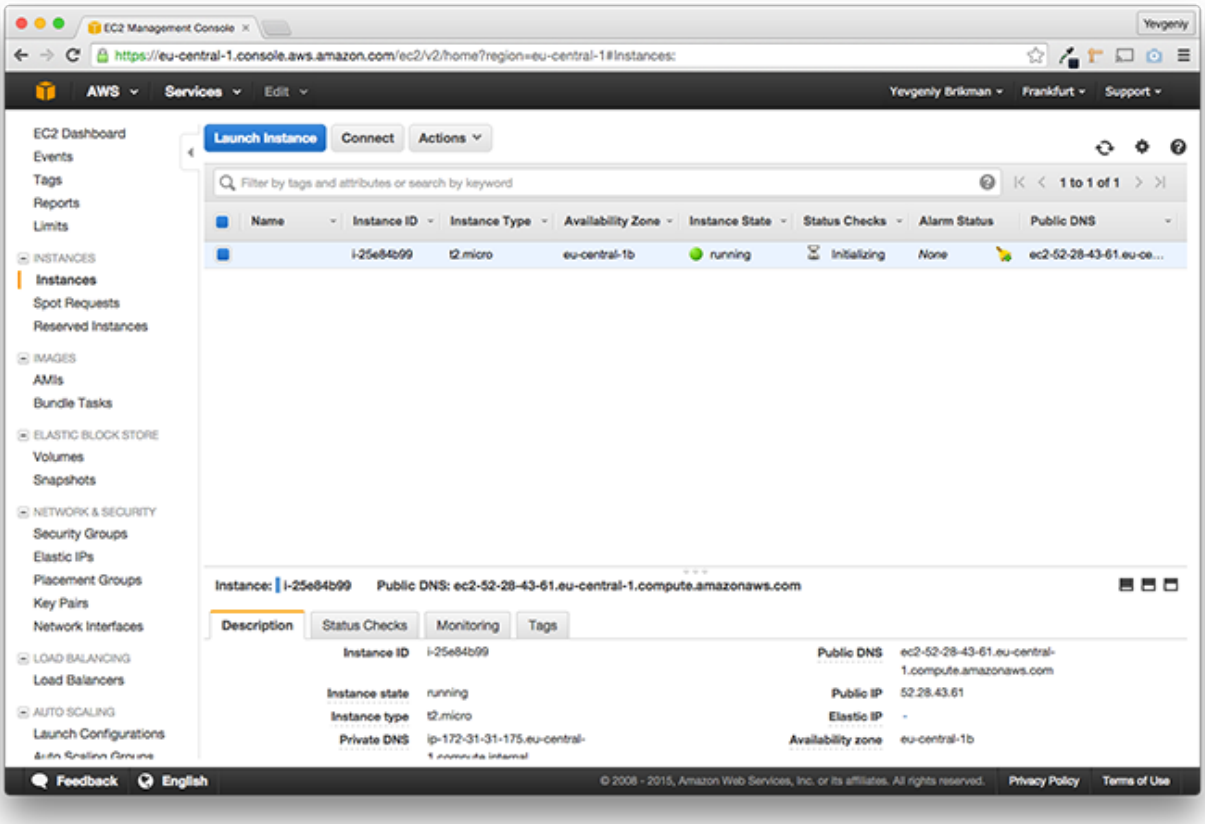
Configure security group

On the "Review Instance Launch" page, click the blue "Launch" button. This will pop up a modal that asks you to pick a Key Pair. A key pair consists of a public key and a private key file that you can use to connect to your EC2 Instance over SSH. Select "create a new key pair" from the drop-down, give the Key Pair a name like `my-ec2-key-pair`, and click "Download Key Pair":



Create and download a Key Pair

Save the Key Pair `.pem` file to a safe and accessible location on your computer (once you close this modal, you will never be able to download this `.pem` file again, so make sure to save it!). Now, click the blue "Launch Instances" button in the bottom right of the modal. This takes you to a "Launch Status" page. Click the blue "View Instances" button in the bottom right of this page, and you'll be taken to the EC2 Instances page:



EC2 Instances

This page shows all the EC2 Instances you have running. Click on your newly created EC2 Instance to see more information about it in the section at the bottom of the page, such as its state (running, pending, or terminated), launch time, and public IP address. Copy the public IP address, as you'll need it to SSH to the server and install Docker.

Installing Docker

The next step is to install Docker on your EC2 Instance. Open a terminal, `cd` over to the folder where you saved your Key Pair, and run the following commands:

```
> cd ~/my-aws-key-pairs
> chmod 400 my-ec2-key-pair.pem
> ssh -i my-ec2-key-pair.pem ec2-user@<EC2-INSTANCE-PUBLIC-IP-ADDRES
```

If you did everything correctly, you should see something like this:

```
       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2015.09-release-notes/
[ec2-user]$
```

You are now in control of a fully working Linux server running in the AWS cloud. Let's install Docker on it.

```
[ec2-user]$ sudo yum update -y
[ec2-user]$ sudo yum install -y docker
[ec2-user]$ sudo service docker start
```

Next, add the `ec2-user` to the `docker` group so you can execute Docker commands without using `sudo`. Note that you'll have to log out and log back in for the settings to take effect:

```
[ec2-user]$ sudo usermod -a -G docker ec2-user
[ec2-user]$ exit

> ssh -i my-ec2-key-pair.pem ec2-user@<EC2-INSTANCE-PUBLIC-IP-ADDRES

[ec2-user]$ docker info
```

If you did everything correctly, the last command, `docker info`, will return lots of information about your Docker install without any errors.

Finally, you can run the `training/webapp` image:

```
[ec2-user]$ docker run -d -p 80:5000 training/webapp:latest python a
```

The `-p 80:5000` flag in the command above tells Docker to link port 5000 on the Docker container to port 80 on the EC Instance. You can test that the Docker image is running as follows:

```
[ec2-user]$ curl http://localhost
Hello world!
```

Since the Security Group for this EC2 Instance exposed port 80 to the world, you can also connect to the public IP address of your EC2 Instance from any web browser:

Testing the EC2 Instance from the browser

If you see the `Hello world!` text, then the good news is that you are successfully running a Docker container in the AWS cloud. The bad news is

that launching Docker containers using this manual process has a number of drawbacks:

1. **Automation**. In the example above, you deployed just a single Docker container to a single EC2 Instance. What happens if you have many different Docker containers (e.g., one container for a front-end web app, another for a back-end, another for a database, and so on) and you have to deploy multiple copies of each of those containers across many EC2 Instances? You wouldn't want to repeat all of these manual steps over and over again, so you need a way to automate deployment.
2. **Integration**. Running a Docker container is only one piece of the puzzle. You also need to integrate it with all the other parts of your infrastructure, such as routing traffic to your containers (load balancing) and ensuring the your containers continue running (monitoring, alerting, crash recovery).

One way to solve both of these problems is to use the EC2 Container Service (ECS).

Deploying Docker containers on ECS

The idea behind ECS is that you create an *ECS Cluster*—which is a group of EC2 Instances managed by ECS—define what Docker containers you want to run, and ECS will take care of deploying those containers across the Cluster, rolling out new versions, and integrating with other AWS infrastructure. ECS can make it easier to manage multiple Docker containers running on multiple EC2 Instances—*if* you can figure out all the steps required to use it. These steps are:

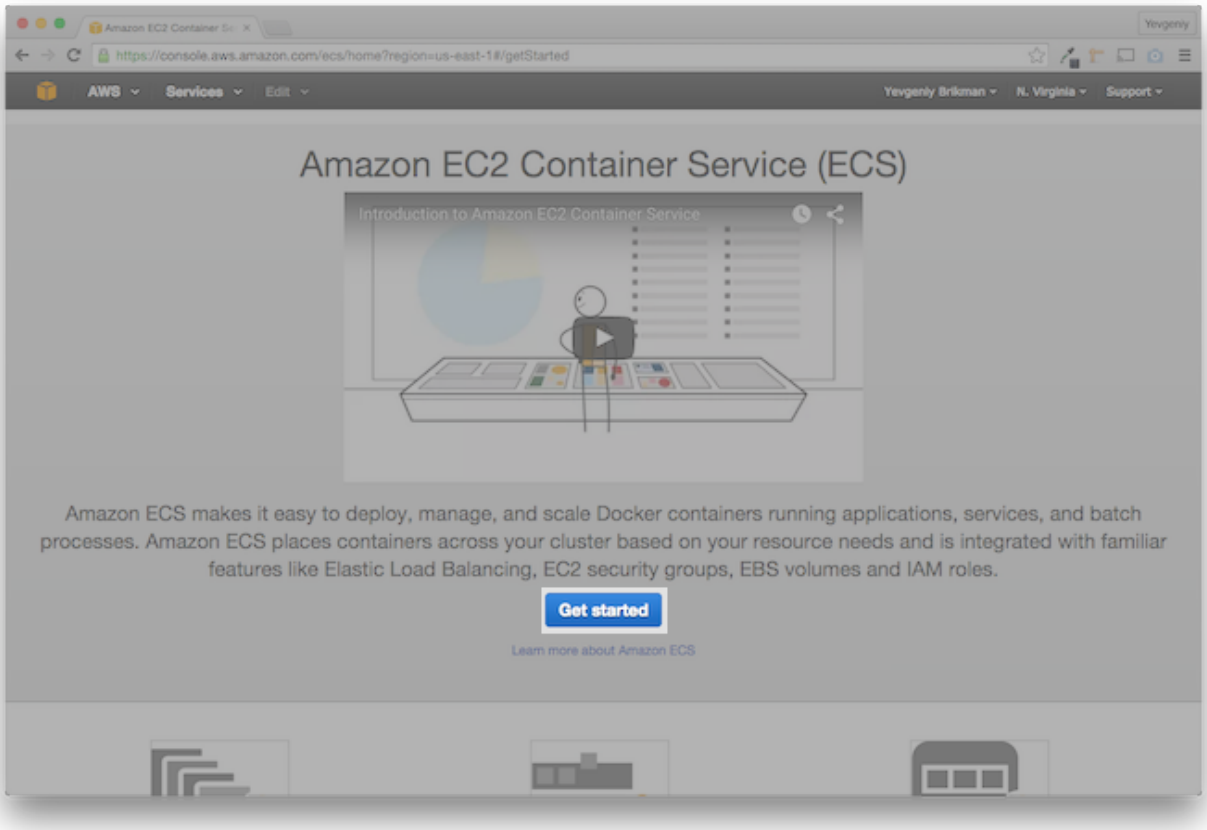
1. Create an ECS Cluster
2. Create an ELB
3. Create IAM Roles
4. Create an Auto Scaling Group

5. Run Docker containers in your ECS Cluster
6. Update Docker containers in your ECS Cluster

To understand what all of these steps mean and how to do them, let's walk through an example.

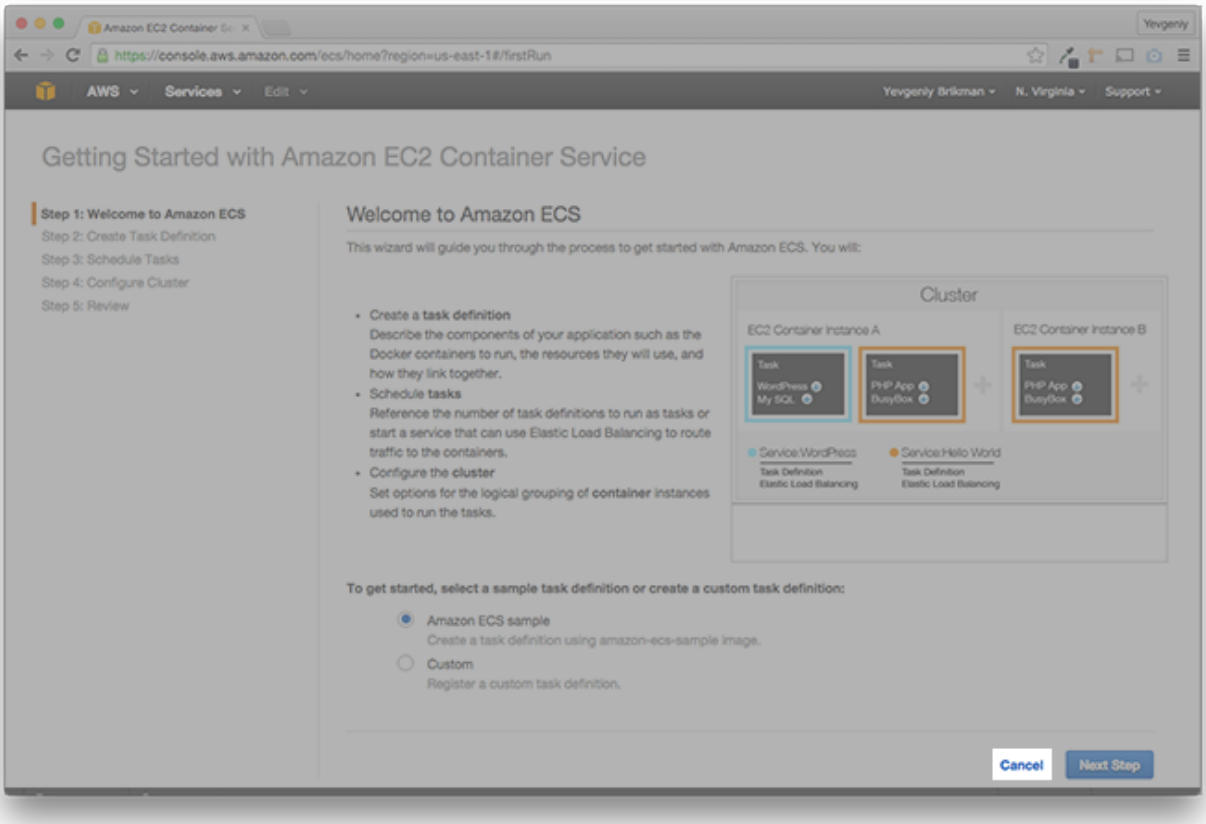
Creating a Cluster

Open up your AWS Console and click on the EC2 Container Service link to go to the ECS Console. If this is your first time using ECS, you will be taken to a getting started page. Click the blue "Get started" button:



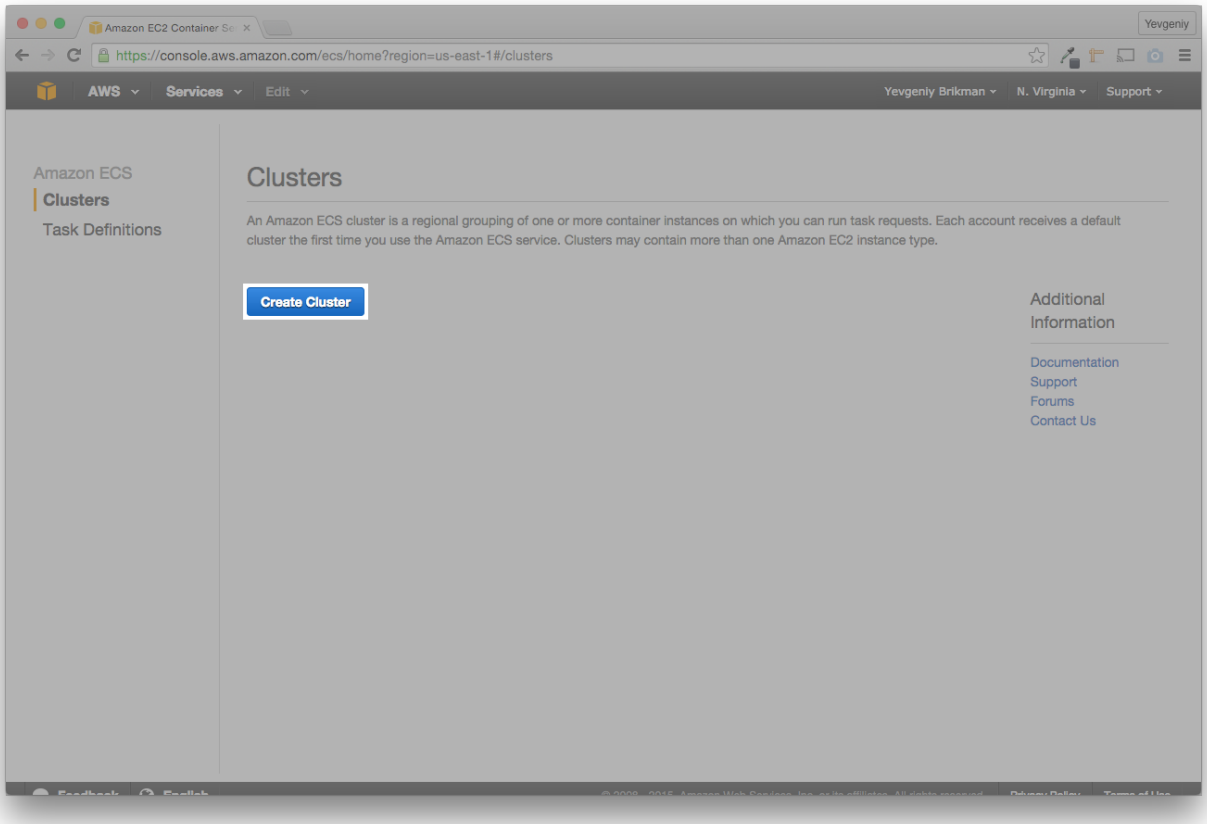
ECS Get Started

This takes you to a wizard that walks you through the process of using ECS, but I found the wizard confusing, and as you'll never be able to use the wizard again after this first time, it also doesn't teach you how to use the actual ECS UI. Therefore, I recommend clicking the cancel button in the bottom right corner:



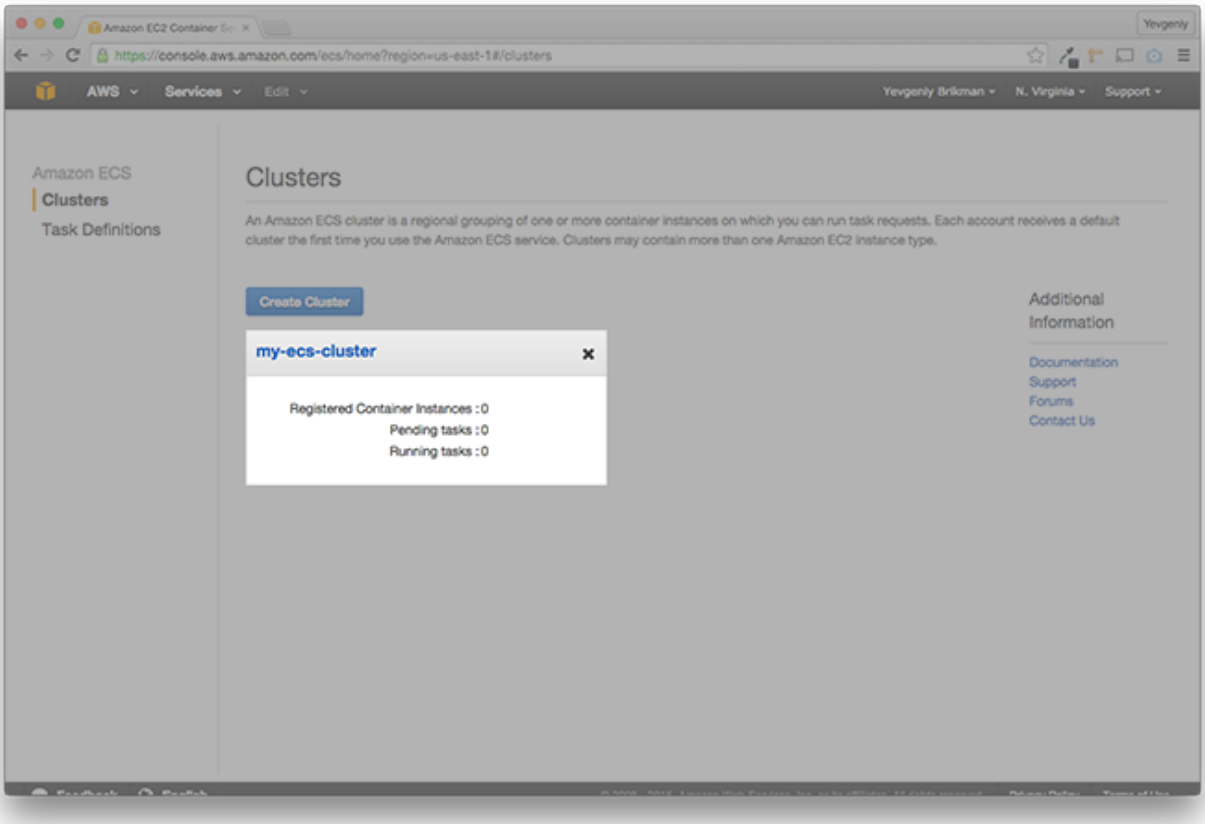
Click cancel to get out of the ECS wizard

This takes you to the Clusters page in the normal ECS UI. To create a Cluster, click the blue “Create Cluster” button:



Create ECS Cluster

Give the cluster a name, such as `my-ecs-cluster`, click the blue "Create" button, and your new Cluster will show up on the Clusters page:



Your new ECS Cluster

Notice how your Cluster shows zero "Registered Container Instances". You need to create a bunch of new EC2 Instances and register them in the Cluster. Deploying, monitoring, and updating many EC2 Instances manually is tedious and error prone. A better solution is to use an *Auto Scaling Group* and an *Elastic Load Balancer*.

You can define an Auto Scaling Group to automatically launch multiple EC2 Instances based on rules you define. For example, you could define rules like "keep 5 EC2 Instances running at all times" or "always maintain a minimum of 3 EC2 Instances, but add one every time the CPU load is above 90% for more than 15 minutes, and remove one every time the CPU load drops below 30% for more than 15 minutes" (you can feed the Auto Scaling Group information about the CPU load and other metrics from Amazon's CloudWatch service).

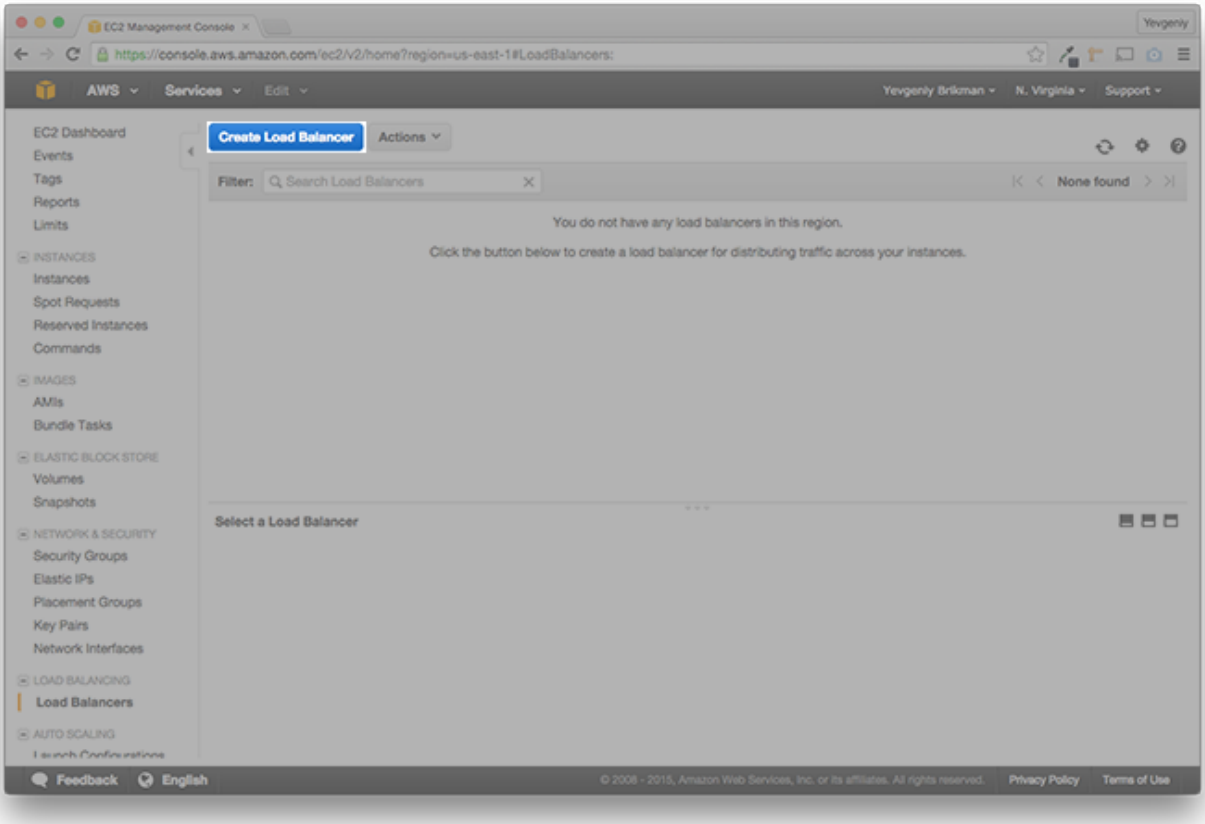
You can use an Elastic Load Balancer (ELB) when you are running multiple EC2 Instances and you want to distribute load between them. The ELB

monitors the health of your EC2 Instances, so if one goes down (due to a crash or an Auto Scaling Group reducing the number of instances) it can take it out of rotation or if a new one comes up (due to a new deployment or an Auto Scaling Group increasing the number of instances) it can add it to the rotation. Your users always send their requests to the ELB, so they are shielded from any changes within your data center.

Let's create the ELB first and then move on to the Auto Scaling Group.

Creating an ELB

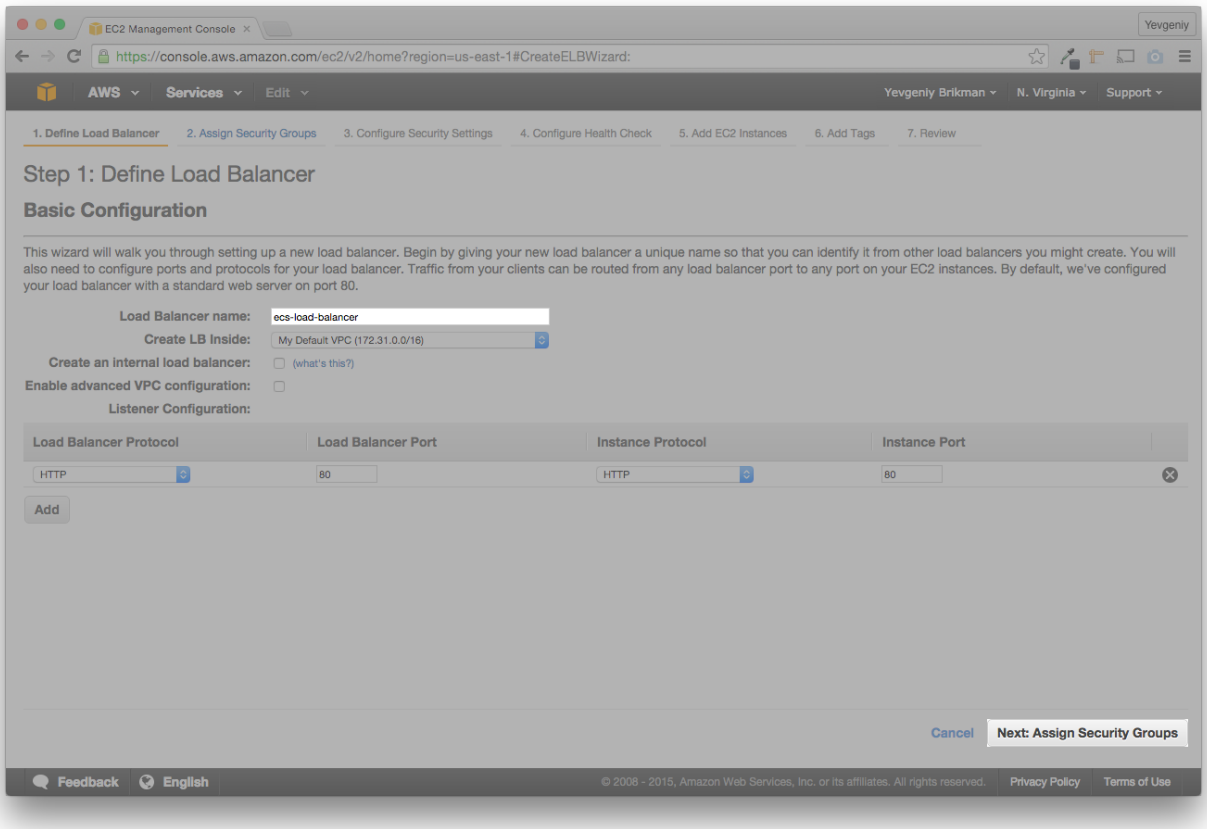
To create an ELB, open the EC2 Console (mouse over the "Services" menu at the top and click "EC2"), click the "Load Balancers" link in the bottom left, and click the blue "Create Load Balancer" button:



Create a Load Balancer

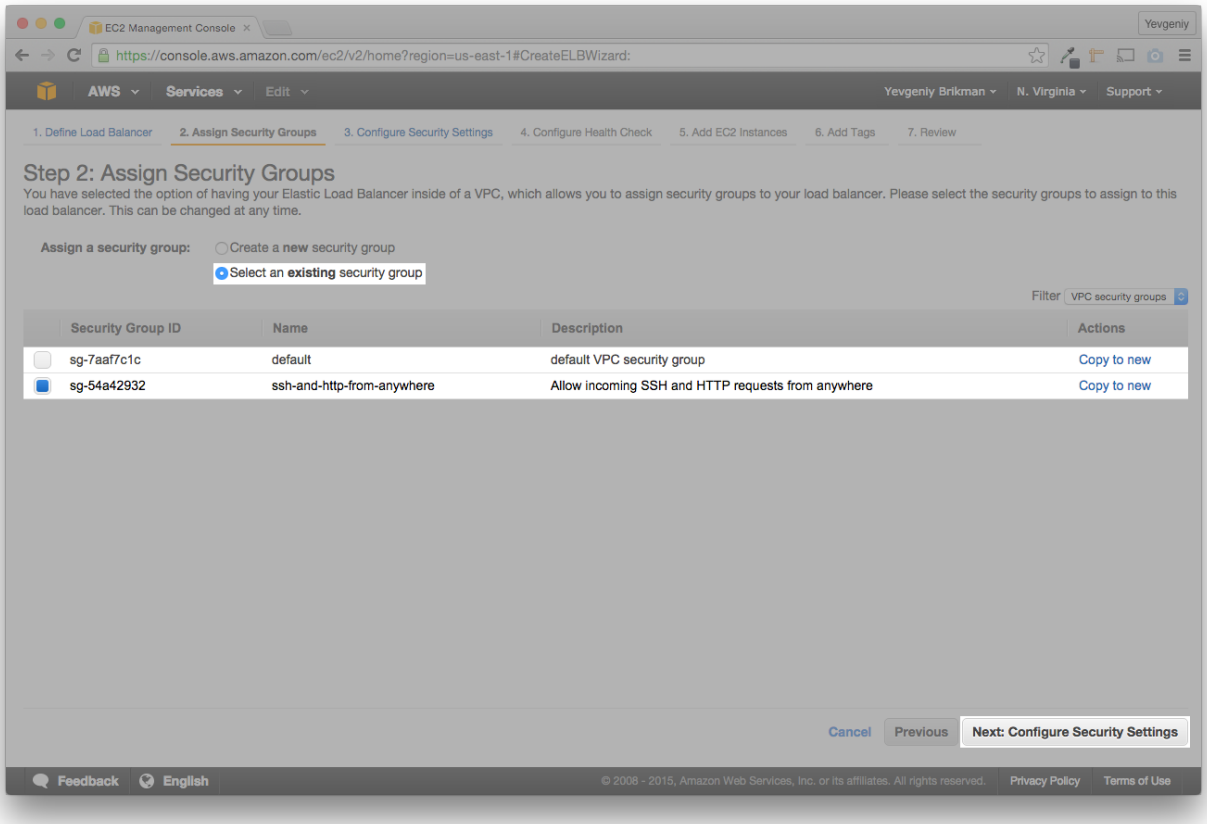
Give the ELB a name such as `ecs-load-balancer` and take a look at the "Listener Configuration" settings. The ELB can only route traffic from one

port to another, such as routing all HTTP traffic that it gets on port 80 to port 80 of any EC2 Instances you attach to it. This limited feature set can be a problem, as we'll discuss towards the end of the blog post. However, this configuration will work for our example, so leave those settings as-is and click the gray "Next: Assign Security Groups" button:



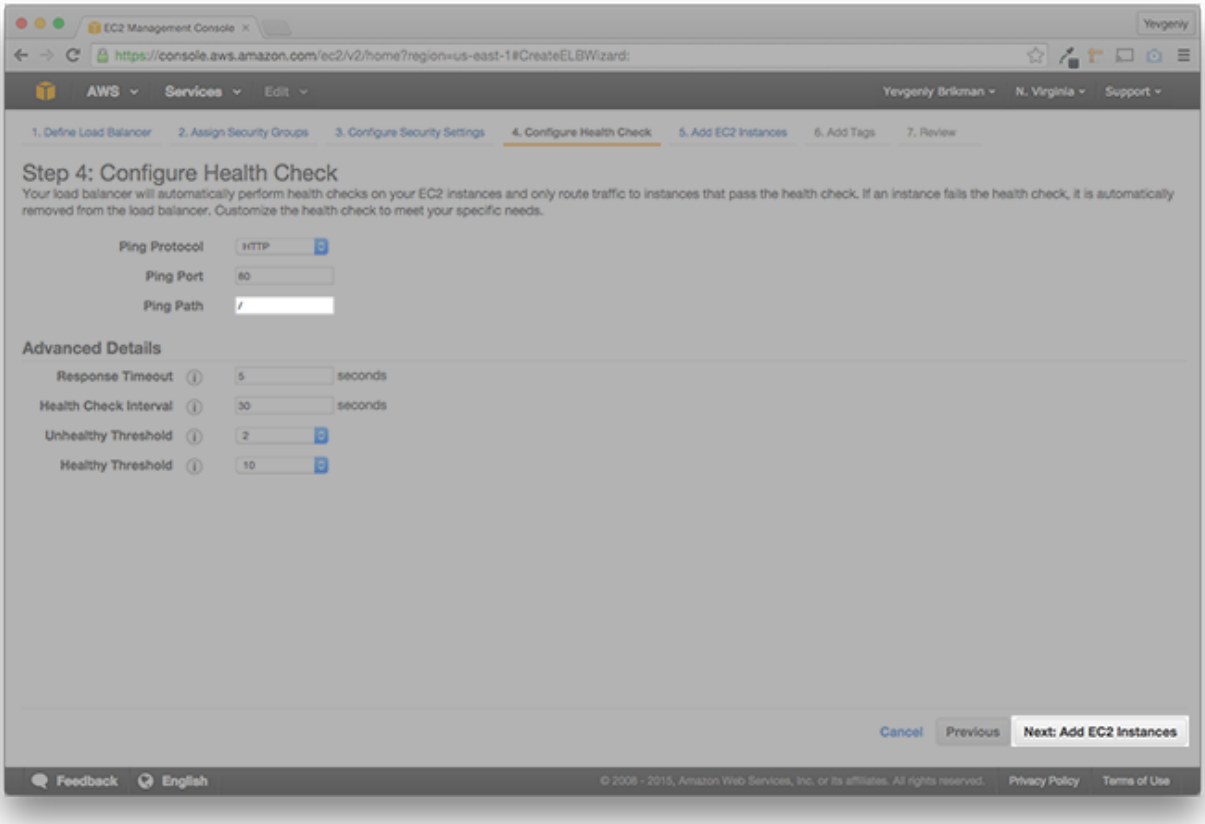
Give the Load Balancer a name

On the next page, click the "Select an existing security group" radio button, click the checkbox next to the Security Group you created earlier (`ssh-and-http-from-anywhere`), and click the gray "Next: Configure Security Settings" button:



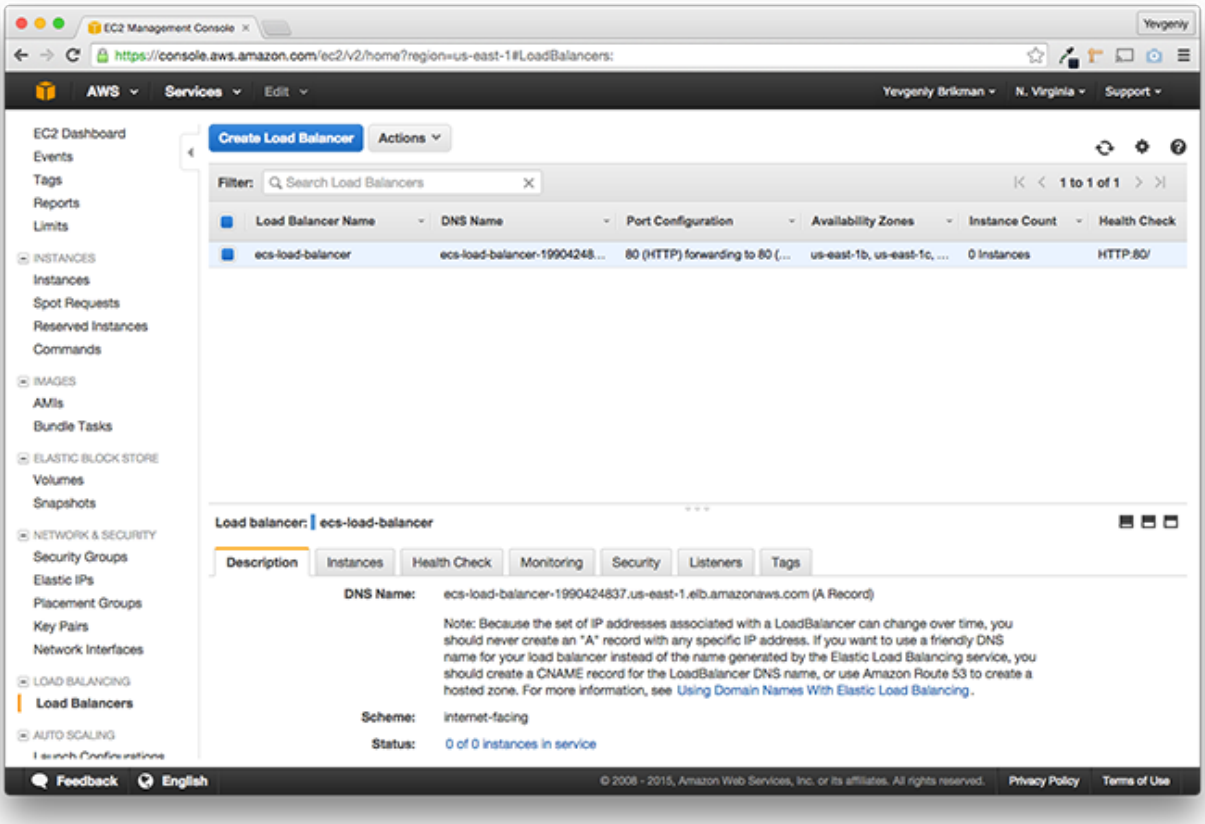
Select the Security Group you created earlier

Ignore the warning (we aren't using SSL in this example) and click the gray "Next: Configure Health Check" button. The ELB uses a Health Check to periodically check if the EC2 Instances its routing to are actually up and running. It does this by sending a *Ping* (usually, an HTTP request) to a configurable URL on each EC2 Instance at a configurable interval. If the EC2 Instance responds with a 200 OK within a configurable period of time, it is considered healthy; if it doesn't, it is taken out of the ELB's rotation until future Pings mark it as healthy. For this tutorial, set the *Ping Path* to `/` (the only URL our Docker container knows how to handle) and click the gray "Add EC2 Instances" button:



Set the Health Check Ping Path to /

On the next page, you can manually add EC2 Instances to the ELB, but we're going to add Instances a different way (using an Auto Scaling Group), so skip this for now by clicking the gray "Next: Add Tags" button, then the blue "Review and Create" button, and finally, the blue "Create" button. Once the ELB is created, click the blue "Close" button on the confirmation page and you should see your new ELB in the list:



Your newly created ELB

Before creating the Auto Scaling Group, we need to take a brief aside to deal with security in the form of IAM Roles.

Creating IAM Roles

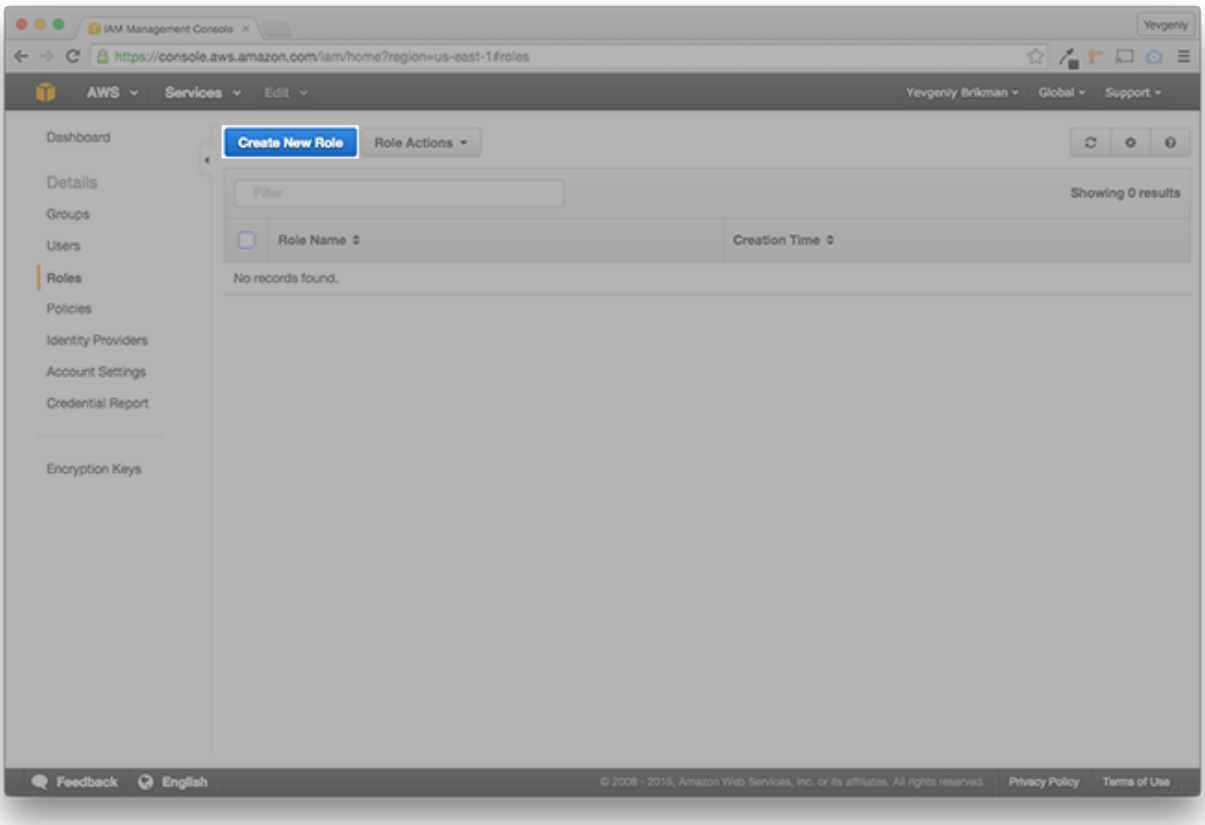
IAM, which stands for Identity and Access Management, is the mechanism AWS uses to:

1. Define Identities, such as a User, Group, or Role (authentication).
2. Define Permissions and Policies that specify what an Identity is or isn't allowed to do (authorization).

For example, later in this post, you are going to create a bunch of EC2 Instances that, when they boot up, need to register themselves in your ECS Cluster. By default, your EC2 Instances do not have the Permissions to talk to an ECS Cluster, so you need to create an IAM Role—an Identity with set of Permissions—and attach it to your EC2 Instances (for more info, see

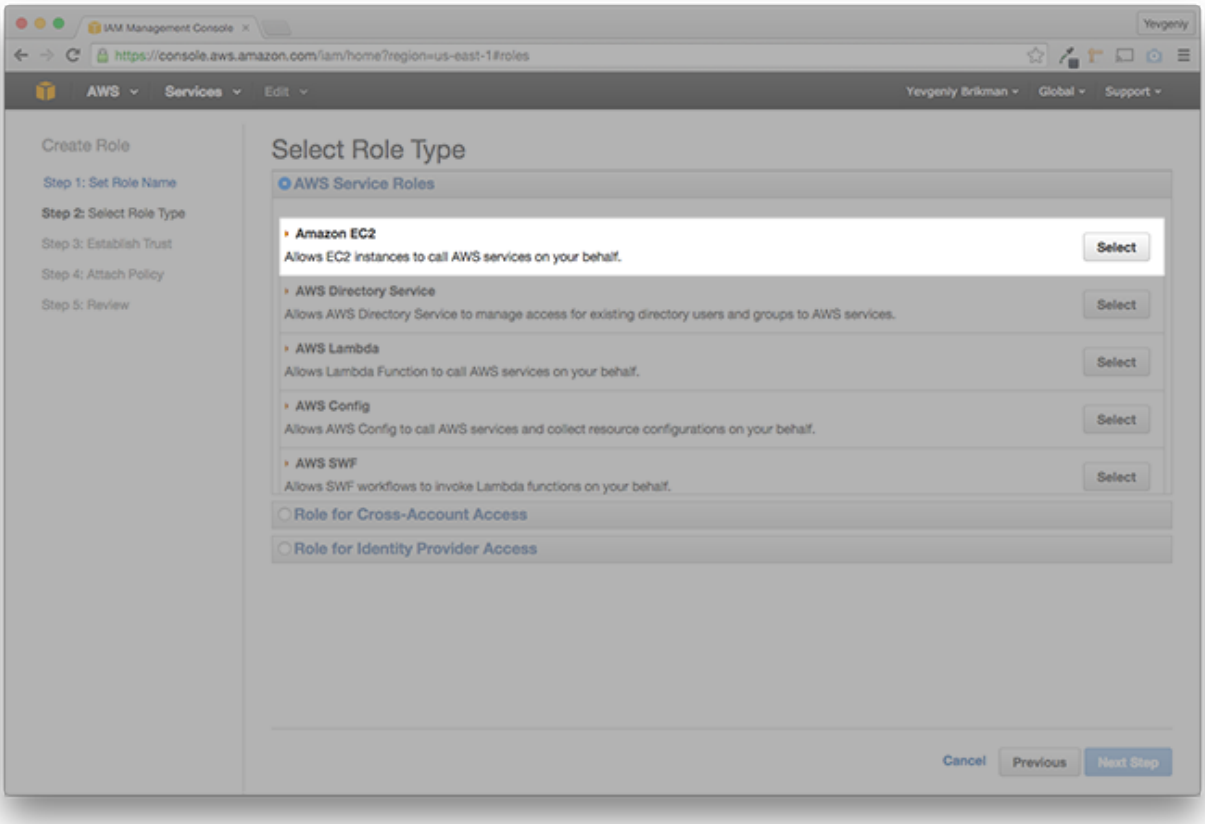
Amazon ECS Container Instance IAM Role).

Head over to the IAM Console (you can mouse over the "Services" menu and click the "IAM" link), click the "Roles" link on the left side, and click the blue "Create New Role" button:



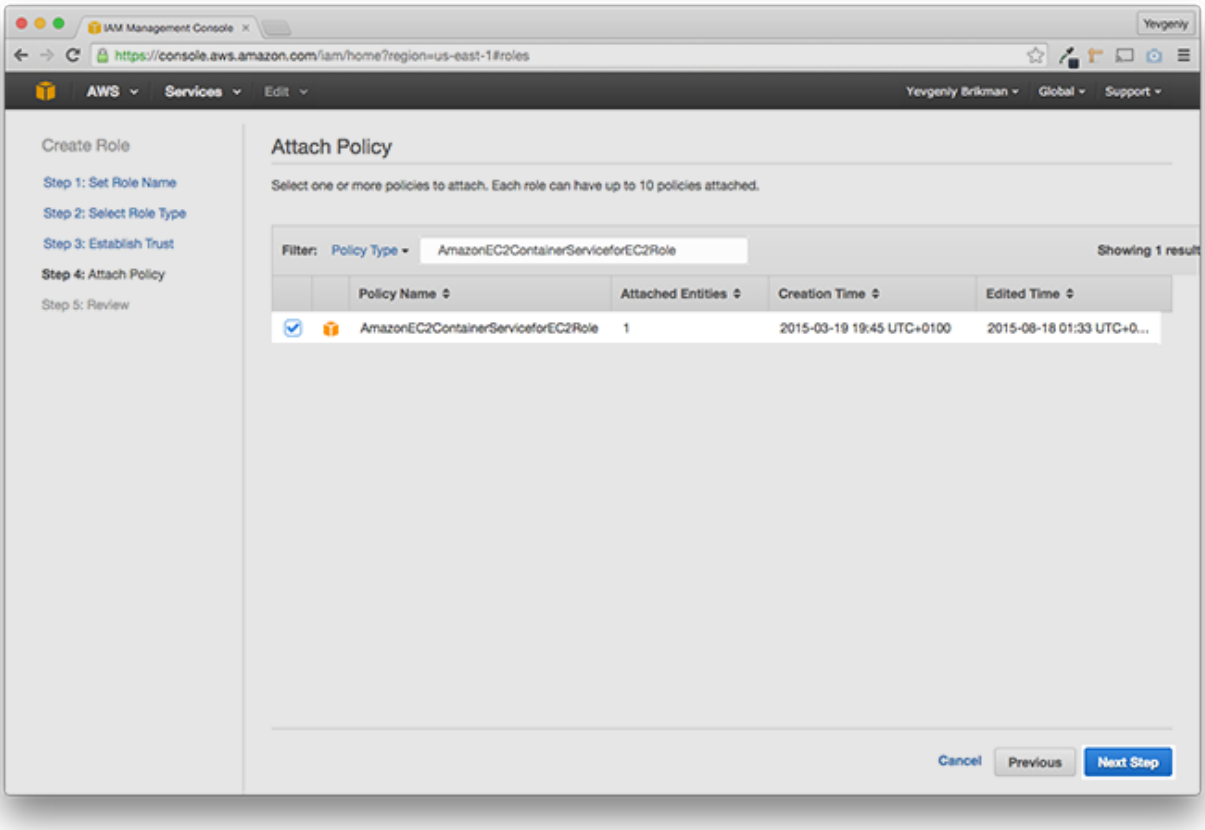
Create an IAM Role

Give the role a name, such as `ecs-instance-role`, and click the blue "Next Step" button in the bottom right. From the "AWS Service Roles" list, click the gray "Select" button next to `Amazon EC2`:



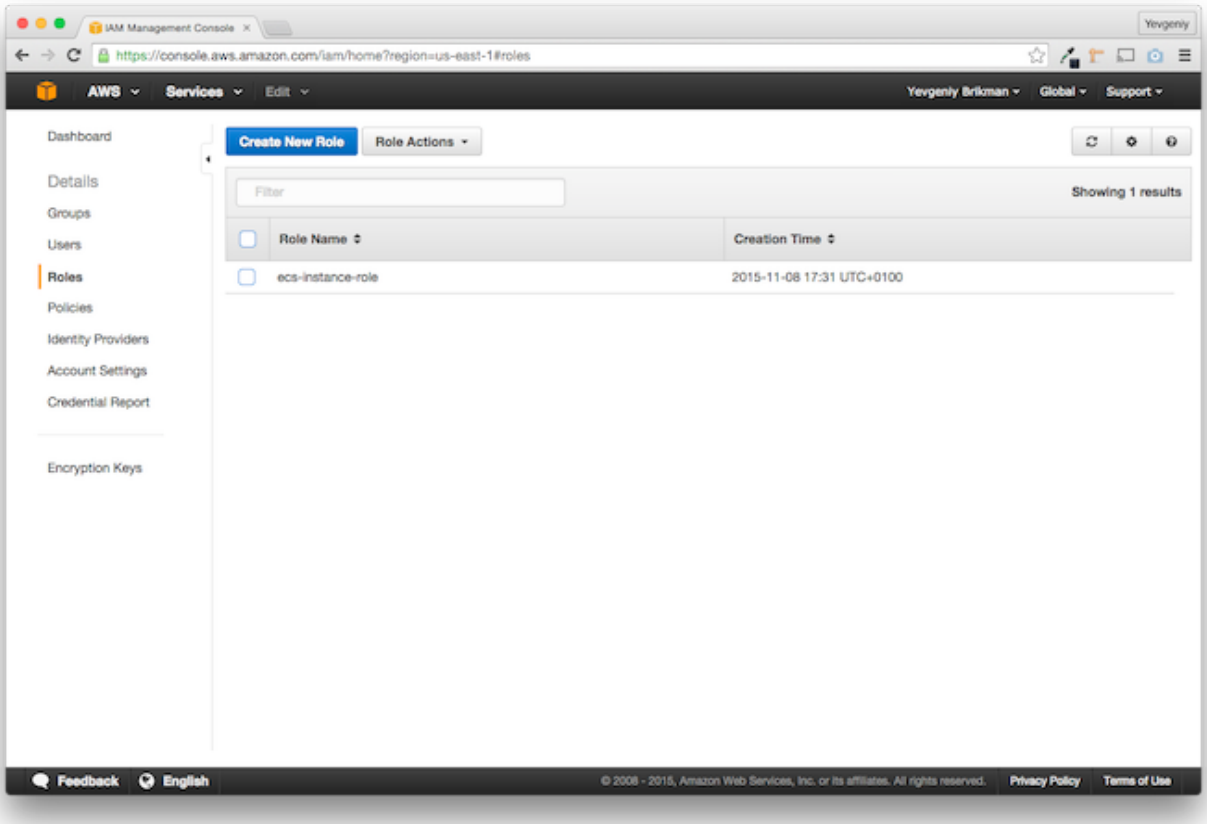
Select the Amazon EC2 Service Role

On the next page, search for `AmazonEC2ContainerServiceforEC2Role` (this is a pre-defined Policy that has all the Permissions you need), click the checkbox next to `AmazonEC2ContainerServiceforEC2Role`, and click the blue "Next Step button":



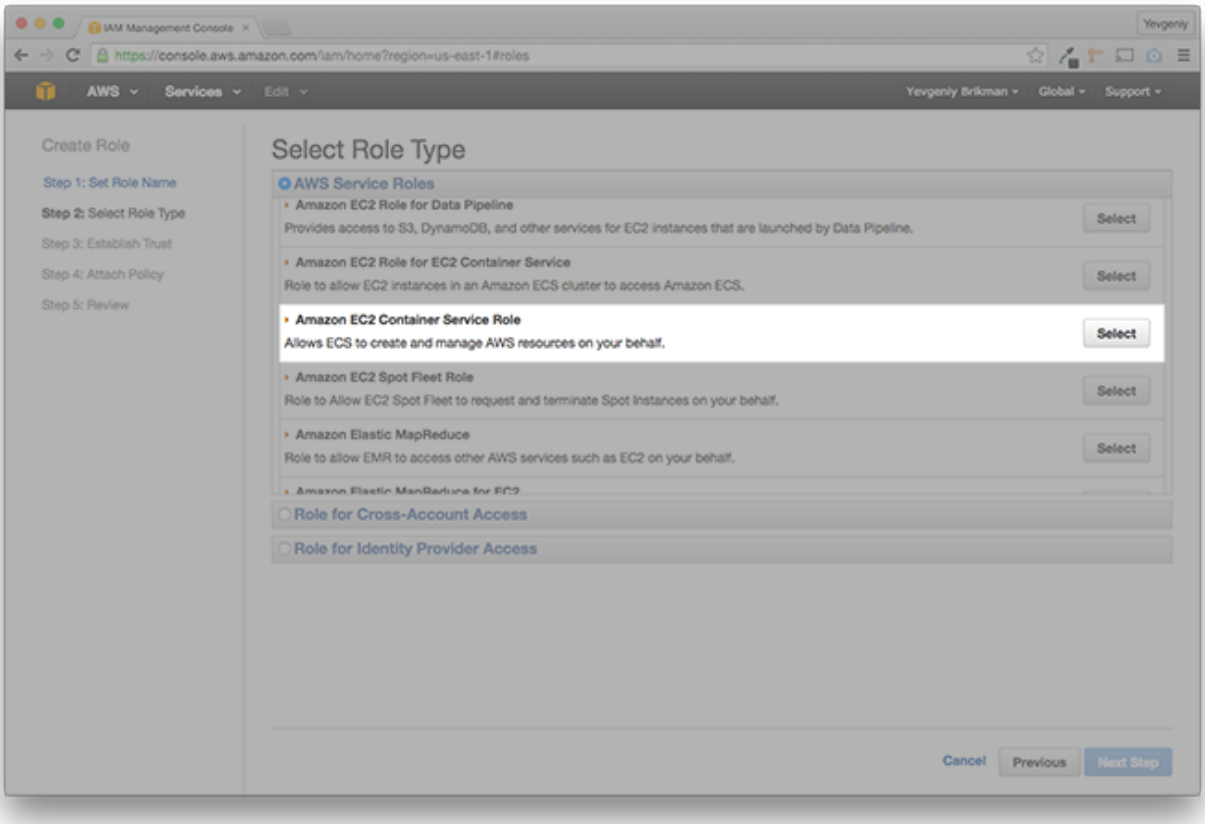
Check AmazonEC2ContainerServiceforEC2Role

Click the blue "Create Role" button and you should see your new IAM Role in the list:



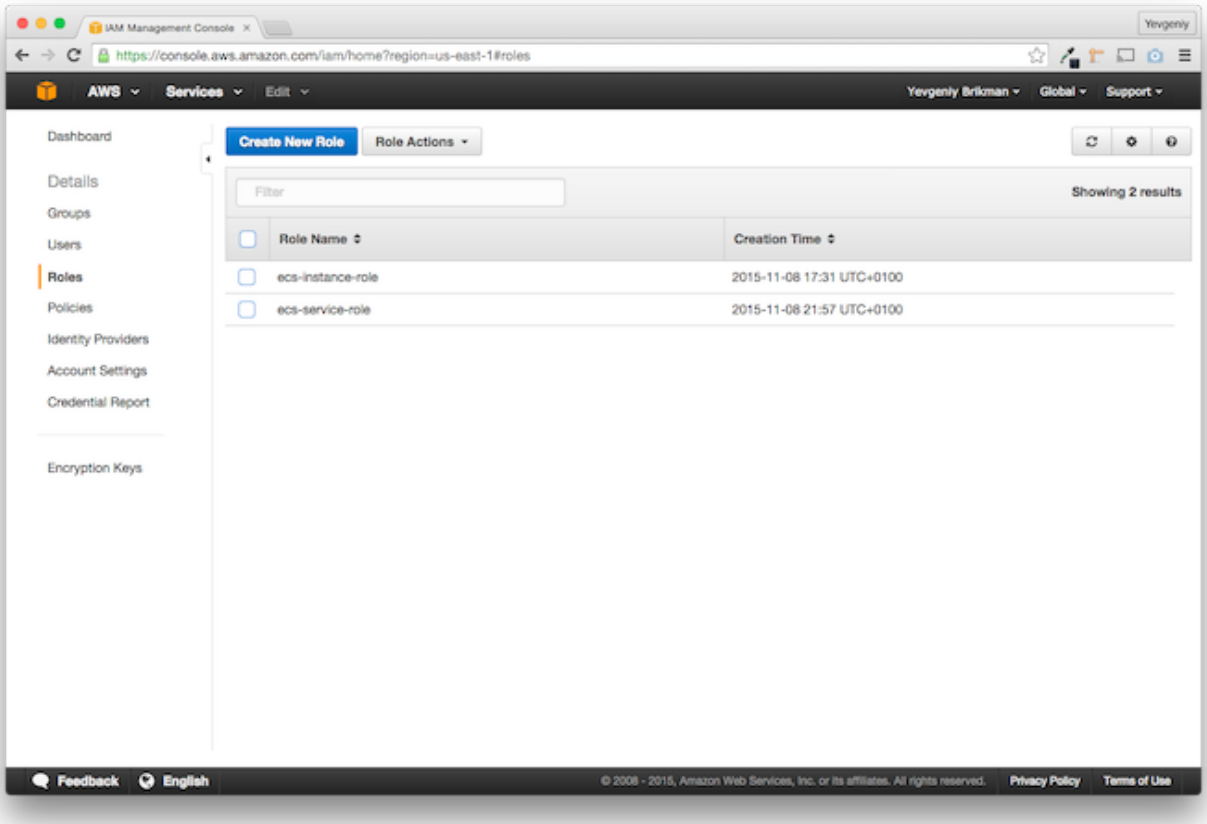
The new IAM Role

You need to create a similar IAM Role to allow the ECS Cluster to talk to your ELB so it can notify the ELB when it is deploying or undeploying Docker containers (see Amazon ECS Service Scheduler IAM Role for more info). Click the blue "Create New Role" button again, give the role a name such as `ecs-service-role`, and click the blue "Next Step" button in the bottom right. From the "AWS Service Roles" list, click the gray "Select" button next to `Amazon EC2 Container Service Role`:



Select the Amazon EC2 Container Service Role

Click the checkbox next to the `AmazonEC2ContainerServiceRole` (another pre-defined Policy, and the only one in the list), click the blue "Next Step" button in the bottom right, and then click the blue "Create Role" button. You should now have two IAM roles:

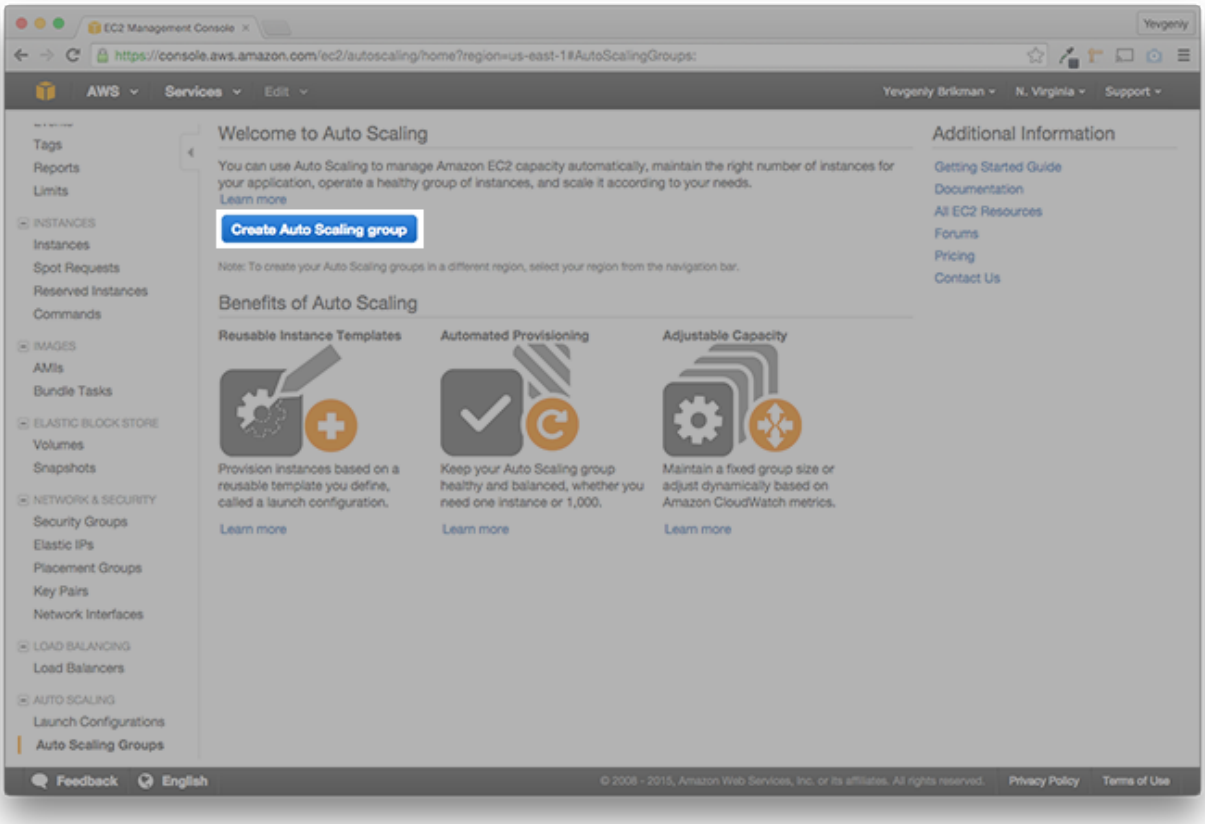


The EC2 Instance Role and the ECS Service Role

With all that out of the way, you can finally create your Auto Scaling Group.

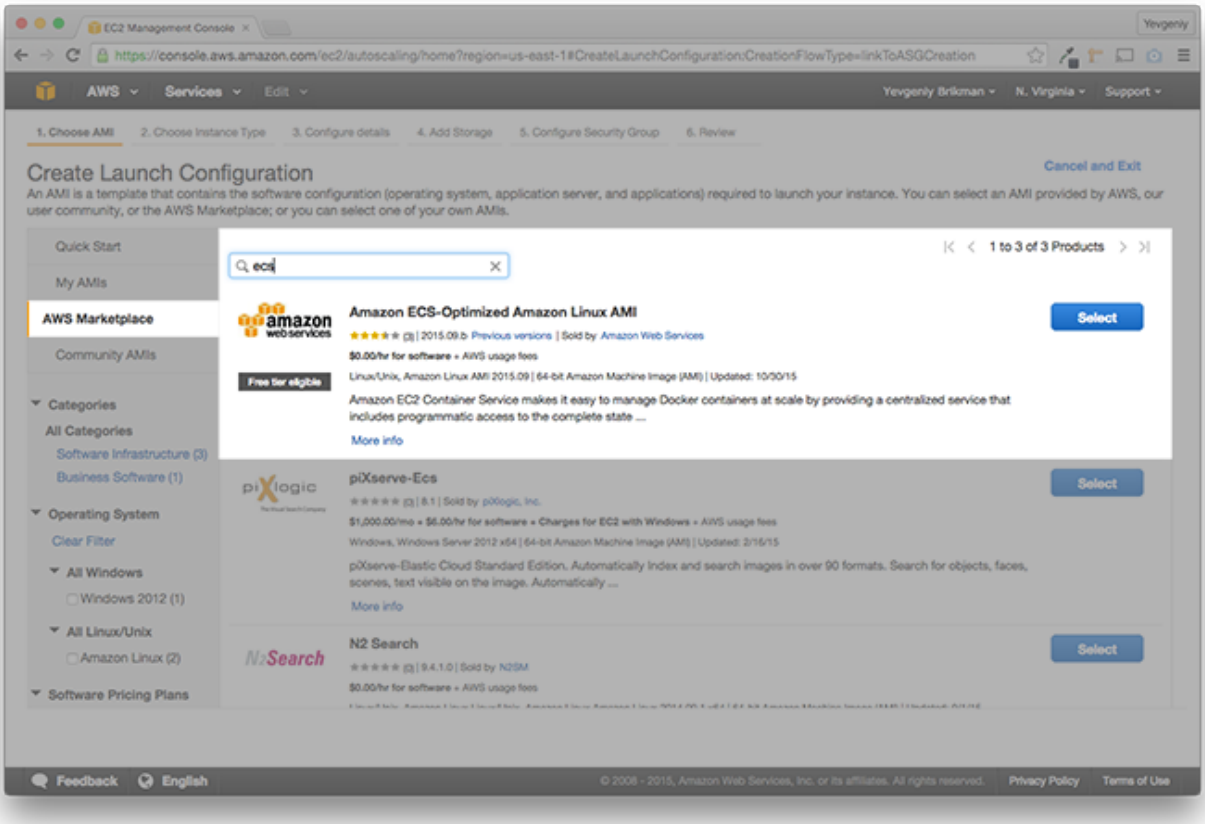
Creating an Auto Scaling Group

To create an Auto Scaling Group, open the EC2 Console (mouse over the "Services" menu at the top and click "EC2"), click the "Auto Scaling Groups" link in the bottom left, and click the blue "Create Auto Scaling Group" button:



Create an Auto Scaling Group

The first step in creating an Auto Scaling Group is to define a Launch Configuration. This is a reusable template that defines what kind of EC2 Instances the Auto Scaling Group should launch, including the AMI, instance type, security group, and all the other details you saw when launching an EC2 Instance manually in the first part of this tutorial. Click the blue "Create launch configuration" button in the bottom-right corner to go to the AMI selection page. Instead of using the Amazon Linux AMI as before, click the "AWS Marketplace" tab on the left, search for `ECS`, and select the `Amazon ECS-Optimized Amazon Linux AMI` (don't worry, it's part of the AWS free tier):



Use the Amazon ECS-Optimized Amazon Linux AMI

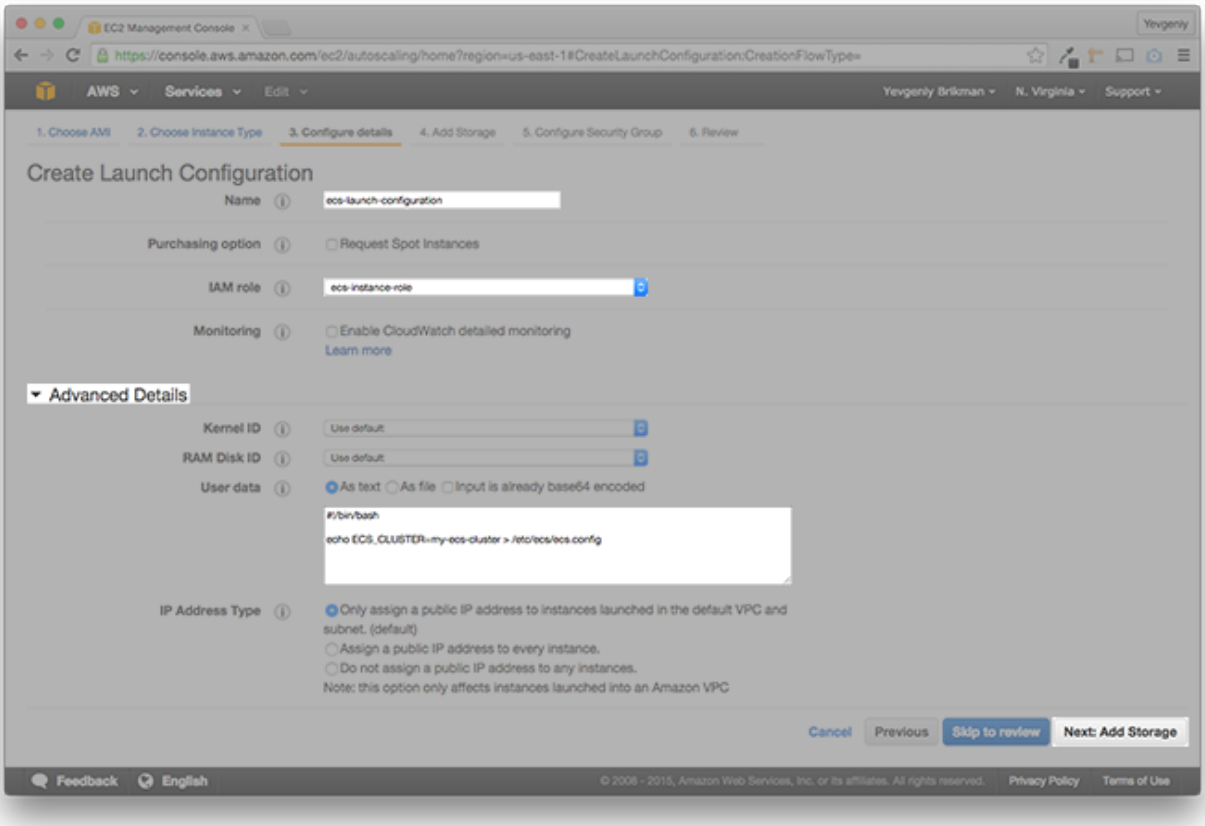
(Note, if this is your first time using the AWS Marketplace, you may have to accept the terms of service, as mentioned in this comment by Petri Sirkkala.)

This Amazon ECS-Optimized Amazon Linux AMI includes the ECS Container Agent software that knows how to register this EC2 Instance in your ECS Cluster. How convenient! The only thing it needs is the name of your ECS Cluster, which we'll get to in just a moment. On the next page, select `t2.micro` as the instance type, and click the gray "Next: Configure details" button. Give the Launch Configuration a name, such as `ecs-launch-configuration`, select the instance IAM Role you created earlier from the drop-down list (`ecs-instance-role`), and then click the "Advanced Details" link to open up the bottom section. Find the text box labeled "User data" and enter the following shell script into it:

```
#!/bin/bash
```

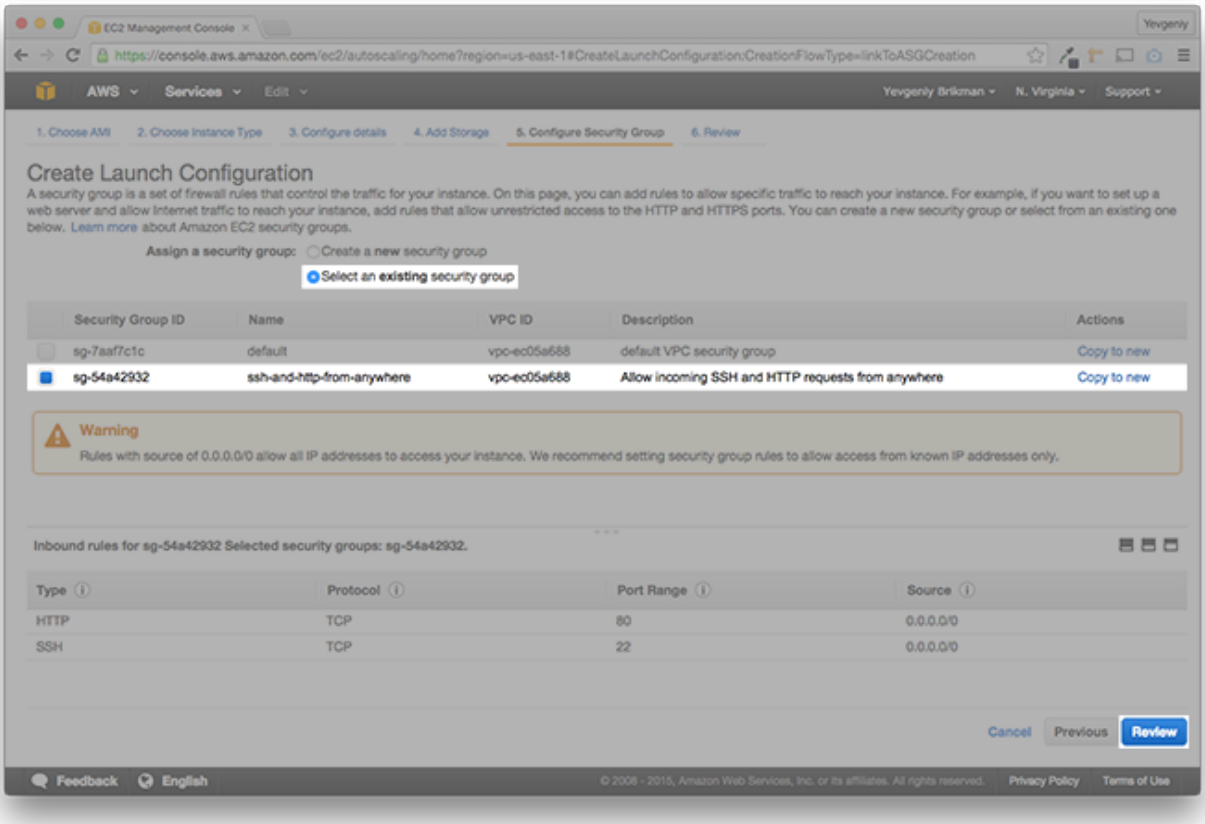
```
echo ECS_CLUSTER=my-ecs-cluster > /etc/ecs/ecs.config
```

User data is a place you can put custom shell scripts that the EC2 Instance will run right after booting. The shell script above puts the name of your ECS Cluster (`my-ecs-cluster`, in this example) into the `/etc/ecs/ecs.config` file. The ECS Container Agent knows to look into this file, so this is how you provide it the name of your ECS Cluster. If you don't specify a name, the Agent will use `Default` (see Amazon ECS Container Agent Configuration for more info). Your Launch Configuration should look something like this:



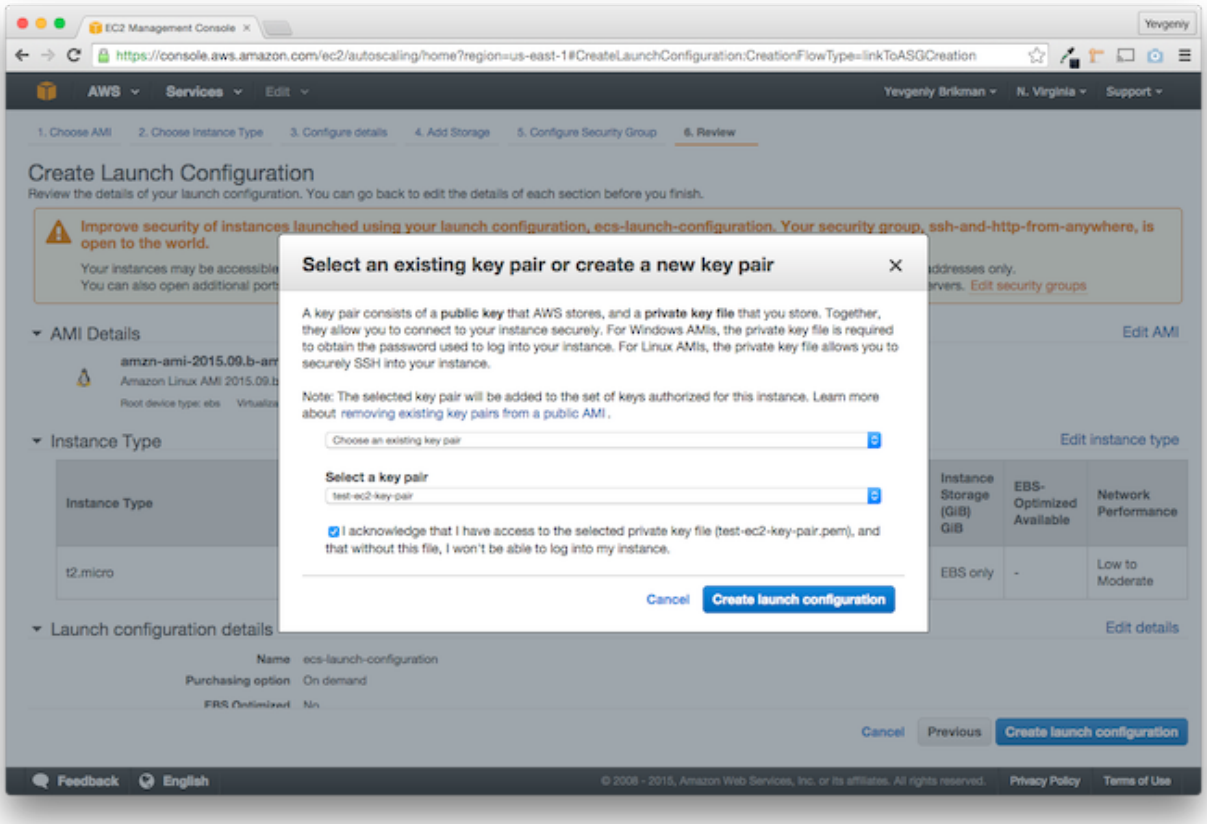
Name the launch configuration and use your newly created IAM Role

Click the gray "Next: Add Storage" button, leave all the default Storage configuration options, and click the gray "Next: Configure Security Group button". On the next page, click the "Select an existing security group" radio button, click the checkbox next to the Security Group you created earlier (`ssh-and-http-from-anywhere`), and click the blue "Review" button:



Use the Security Group you created earlier

Now click the blue "Create launch configuration button" and a modal will pop up asking you to select a Key Pair. Select "Choose an existing key pair" from the first drop-down box, select the Key Pair you created earlier (`my-ec2-key-pair`) from the second drop-down, click the "I acknowledge that I have access to the selected private key file..." checkbox, and click the blue "Create launch configuration" button:



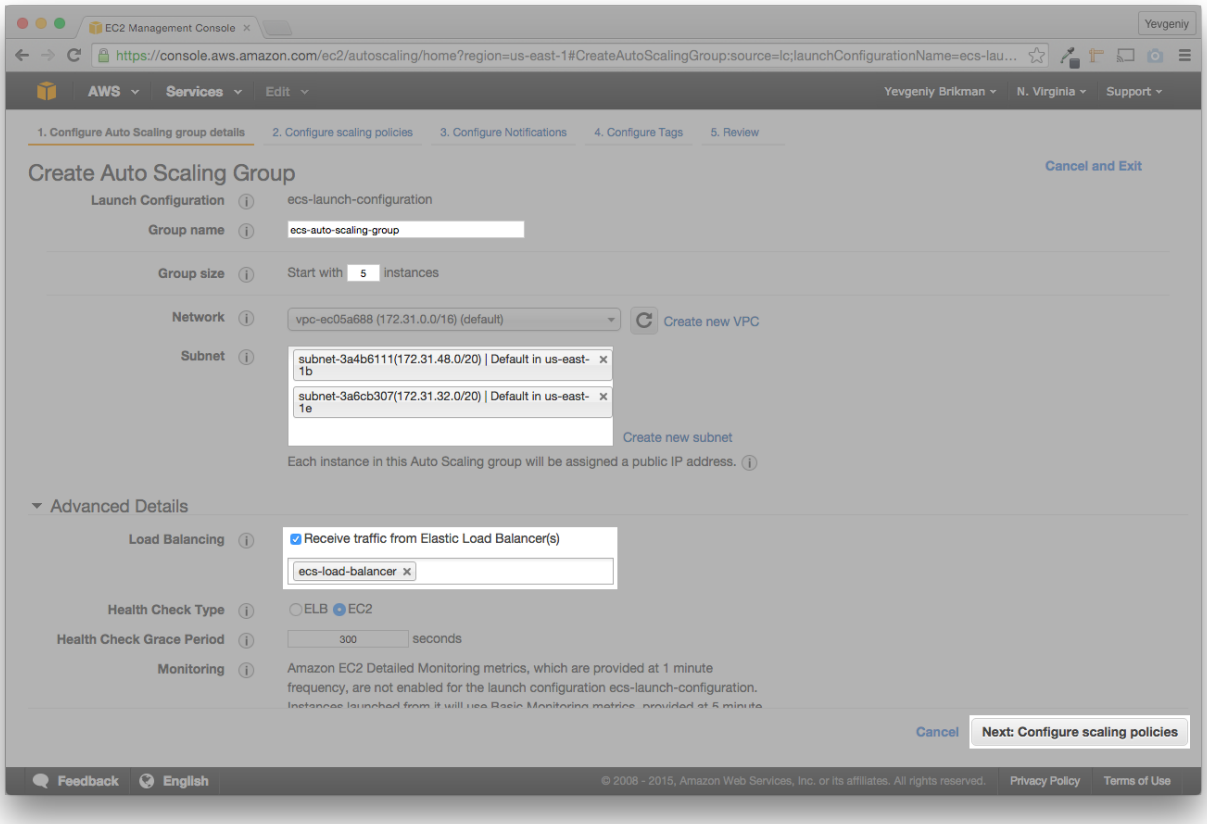
Use the Key Pair you created earlier

Now that you've created a Launch Configuration, AWS should take you to a screen that is prompting you to create an Auto Scaling Group from that Launch Configuration. Give the Auto Scaling Group a name, such as `ecs-auto-scaling-group` and specify a Group size of 5, which will tell the Auto Scaling Group to initially launch 5 EC2 Instances.

Next, you need to pick what Subnet(s) to use. A Subnet is a range of IP addresses used to segregate AWS Resources (such as your EC2 Instances) from each other or from the public Internet. For example, you might put your front-end App Servers in a *Public Subnet*, with rules that make it accessible from the public Internet and you might put your databases in a *Private Subnet*, with rules that make it *only* accessible from the App Servers but not the public Internet or anywhere else. AWS also uses different Subnets for different Availability Zones. AWS has data centers in different locations all over the world. Each location is composed of different Regions and Availability Zones, where a Region represents a geographical area (e.g.,

`us-east-1` and `eu-west-1`) and an Availability Zone is an isolated location within a Region (e.g., `us-east-1a`, `us-east-1b`, `us-east-1c`).

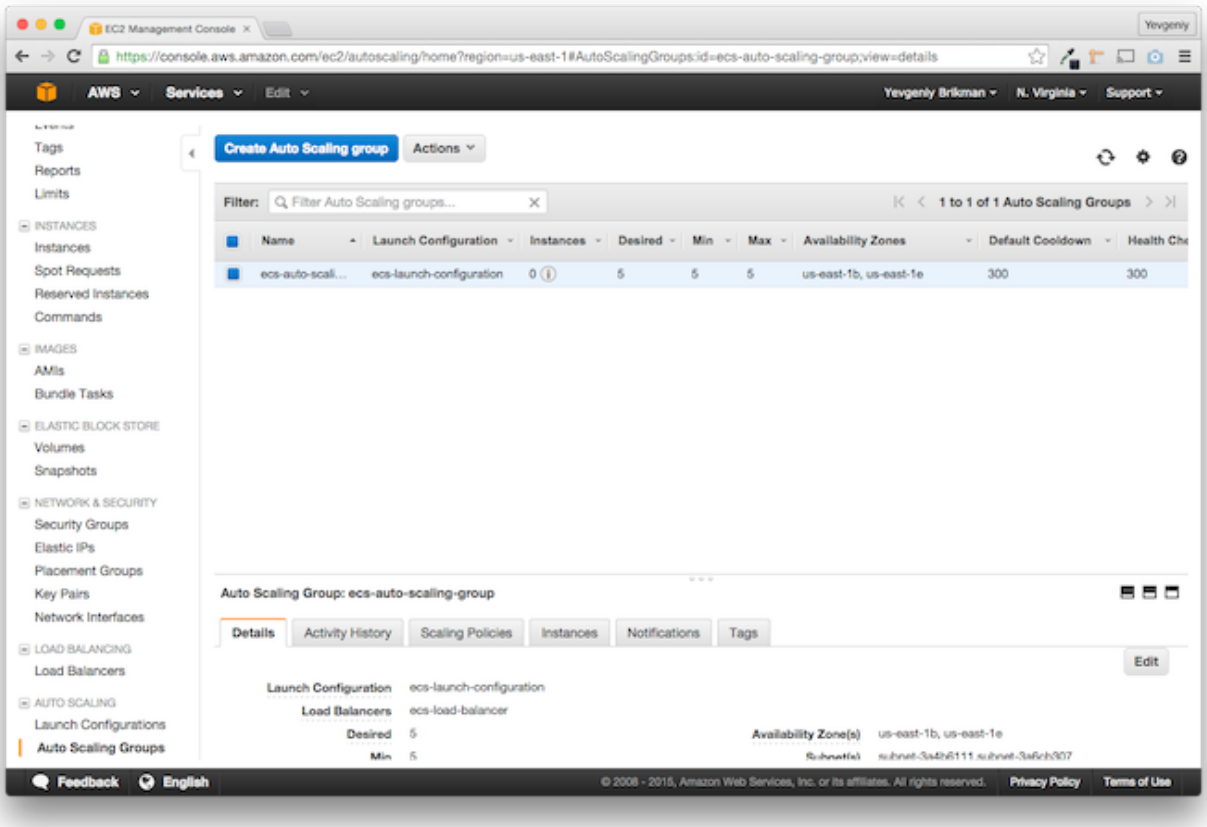
Subnets, Regions, and Availability Zones are all large topics of their own, so I won't cover them here, but you may want to read the AWS VPC and Regions and Availability Zones documentation for more info. For now, pick any of the default Subnets from the drop-down list, and the Auto Scaling Group will deploy your EC2 Instances across them. In the "Advanced Details" section, click the "Receive traffic from Elastic Load Balancer(s)" check box, and select the ELB you created earlier (`ecs-load-balancer`). The page should look something like this:



Give the Auto Scaling Group a name, a size of 5, a couple Subnets, select the ELB you created earlier

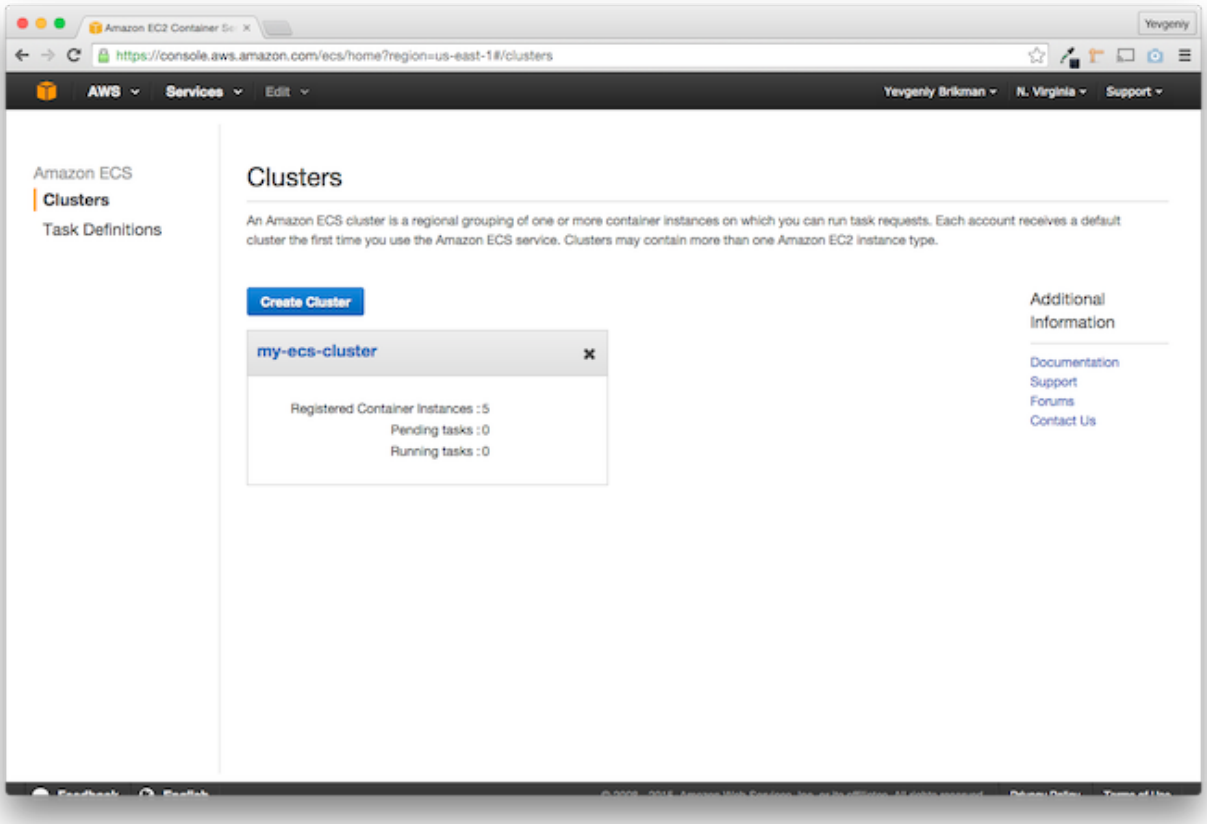
Click the gray "Next: Configure scaling polices" button. On the next page, you could configure rules for how to change the number of EC2 Instances in the Auto Scaling Group, but for this tutorial, you can leave the group at its initial size of 5, so just skip this section and click the blue "Review"

button, followed by the blue "Create Auto Scaling Group" button. Once your Auto Scaling Group has been created, click the blue "Close" button on the confirmation screen and you should see your Auto Scaling Group in the list:



Your newly created Auto Scaling Group

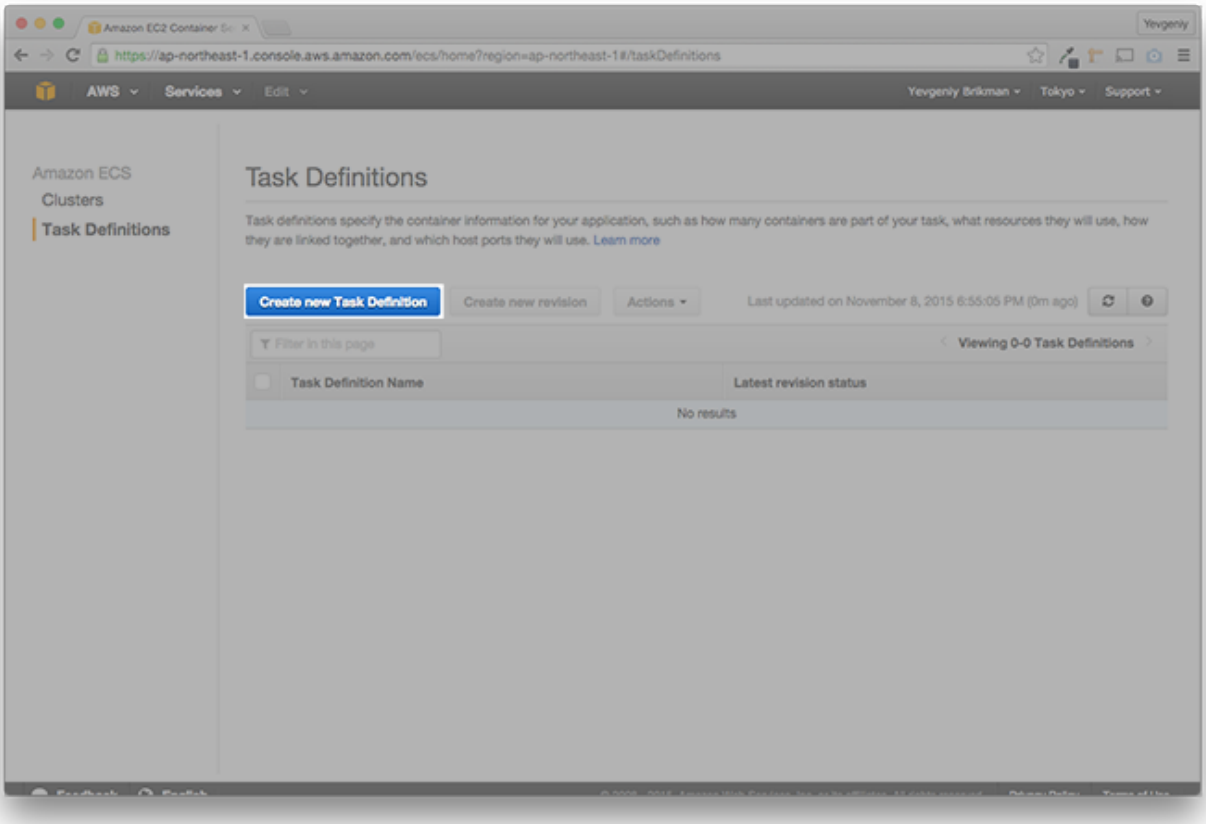
Initially, the Auto Scaling Group will show 5 "Desired Instances", but 0 actually launched Instances. If you wait a minute and refresh the list, the number of launched Instances will go to 5. Head back to the ECS Console, and you should now see five "Registered Container Instances" in your ECS Cluster:



Your ECS Cluster should now have 5 registered instances

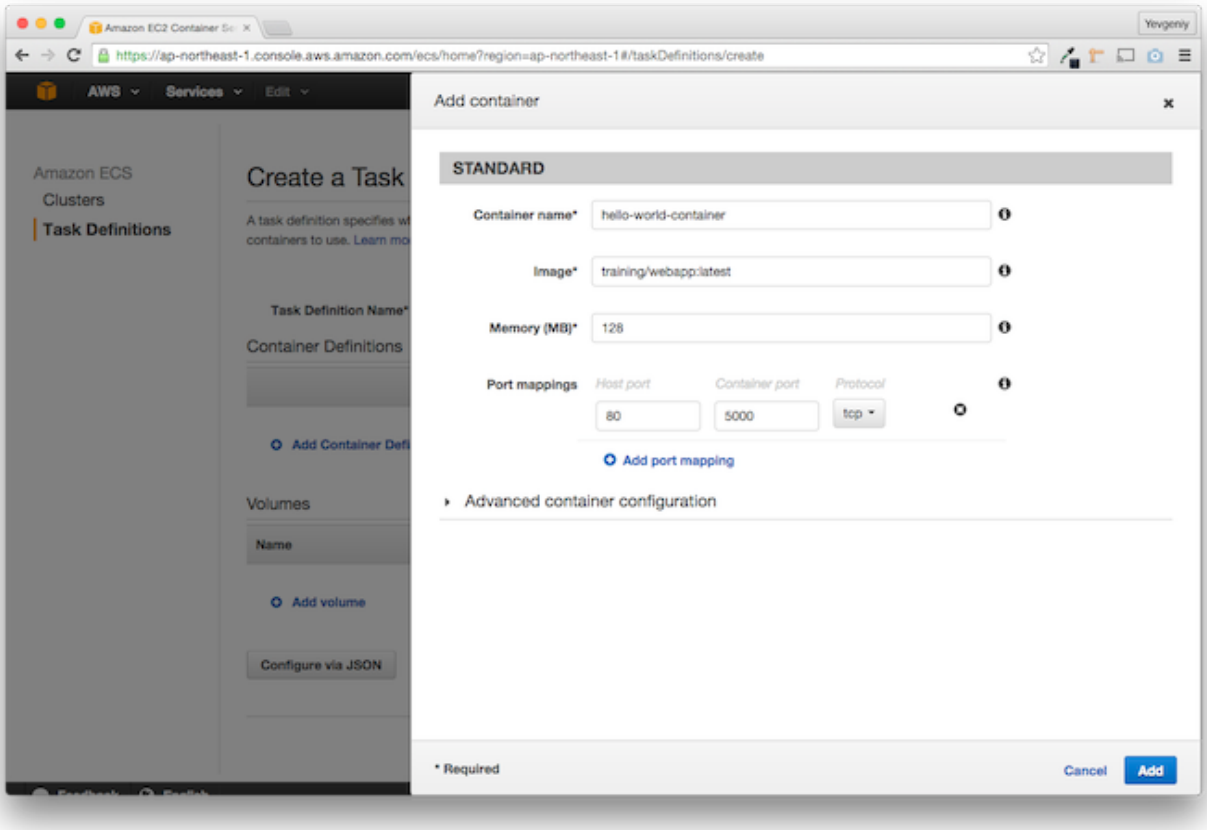
Running Docker containers in your Cluster

Now that you have a working Cluster, you can finally run some Docker containers in it. To do that, you first have to create an ECS Task, which defines the Docker image(s) to run, the resources (CPU, memory, ports) you need, what volumes to mount, etc. Click the "Task Definitions" link on the left and then the blue "Create new Task Definition" button:



Create a new ECS Task

Give the Task a name, such as `hello-world-task` and click the "Add Container Definition" link. In the section that slides out, specify the Container name (e.g., `hello-world-container`), the Docker image to run (`training/webapp:latest`), the amount of memory to allocate (128 is plenty for this tutorial), the port mapping (map port 80 on the host to port 5000 in the Docker container), and click the blue "Add" button:

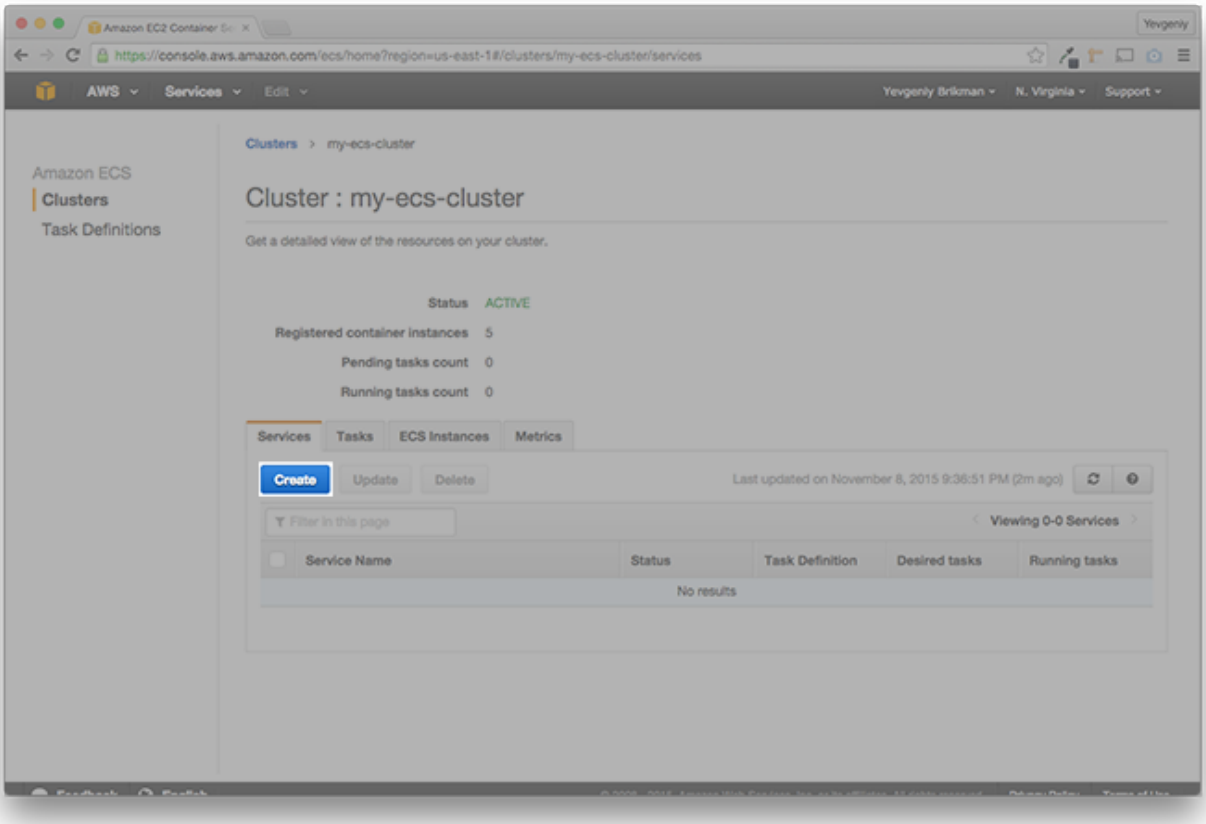


Task definition

Click the blue "Create" button to create the Task. Now it's time to run the Task in your Cluster. Click on the "Clusters" link in the menu on the left and then click on the name of your Cluster (`my-ecs-cluster`). There are two ways to run Tasks:

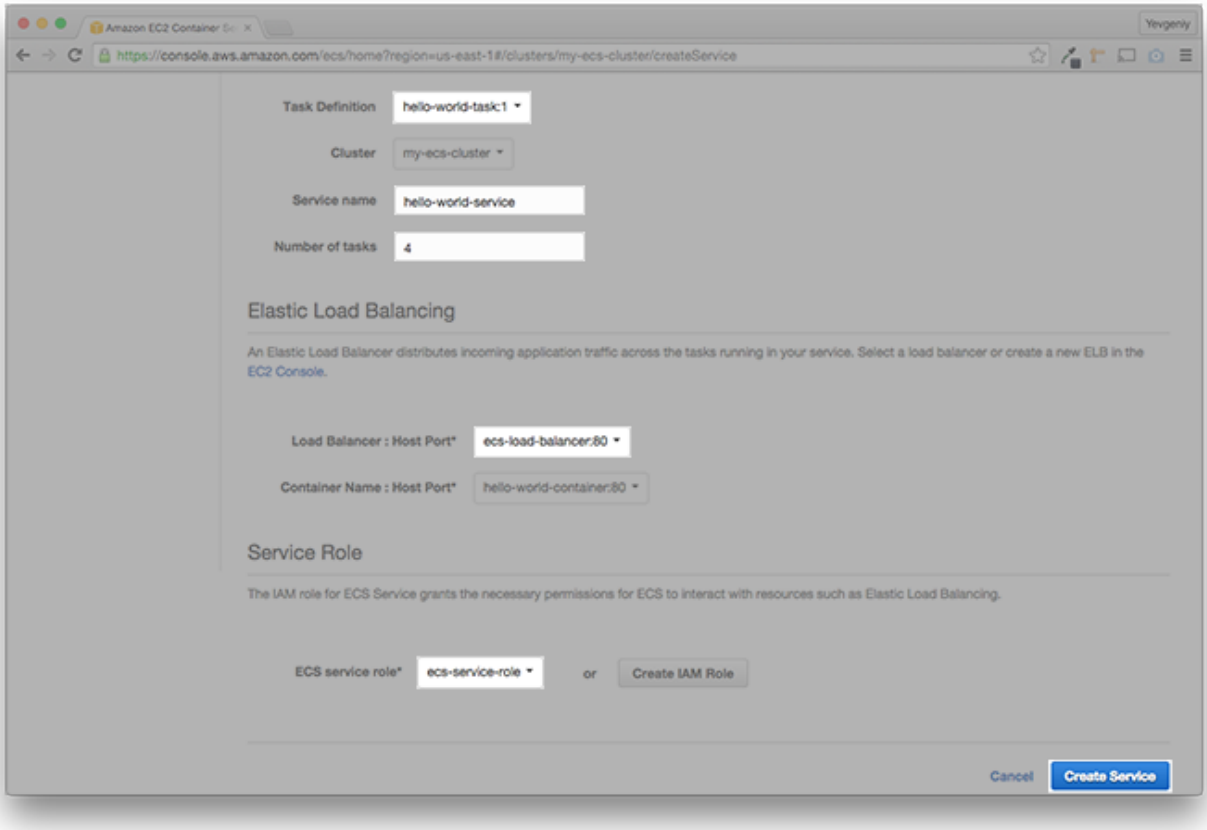
1. **One-off tasks**. This is useful for a Task that runs once to completion and exits. See the "Tasks" tab in your Cluster.
2. **Services**. This is useful for Tasks that run continuously, such as a web service. See the "Services" tab in your Cluster.

Since `training/webapp` is a web app, we will run it as a Service. In the Services tab, click the blue "Create button":



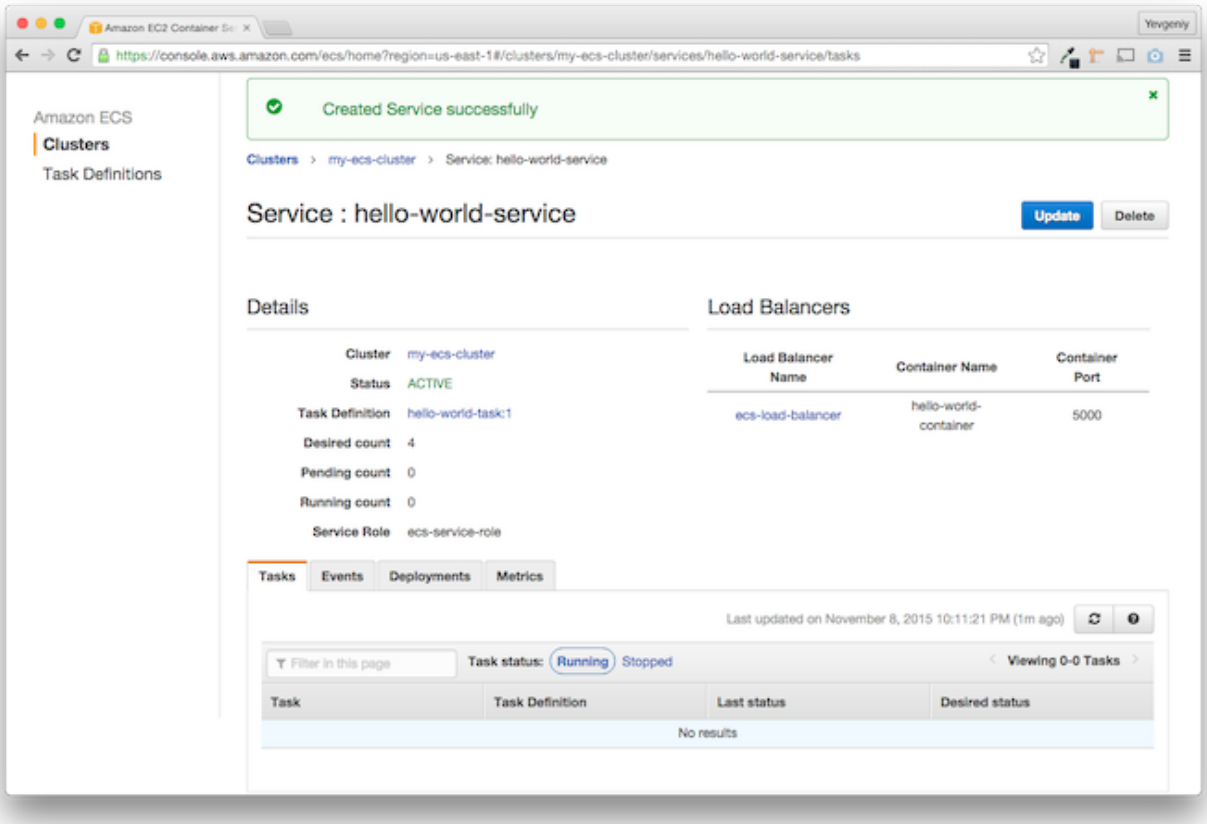
Create an ECS Service

Select the Task you created earlier (`hello-world-task:1`), give the Service a name (e.g., `hello-world-service`), specify that you want 4 tasks (one less than the number of EC2 Instances in your ECS Cluster, as we'll discuss later), select the ELB you created earlier (`ecs-load-balancer`), select the service IAM Role you created earlier (`ecs-service-role`), and click the blue "Create Service" button:



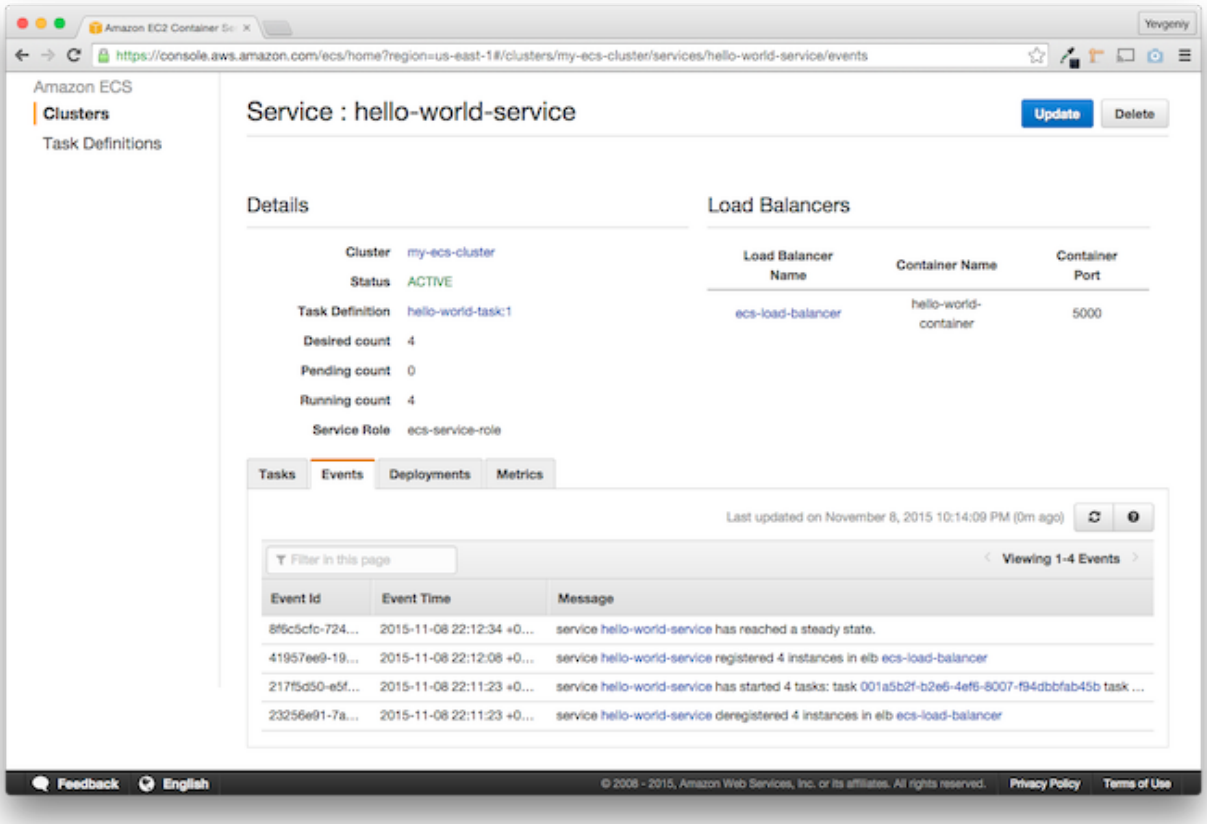
ECS Service Config

You should now see that your ECS Service has been created. Initially, the "desired count" will be at 4 and the "running count" will be at 0:



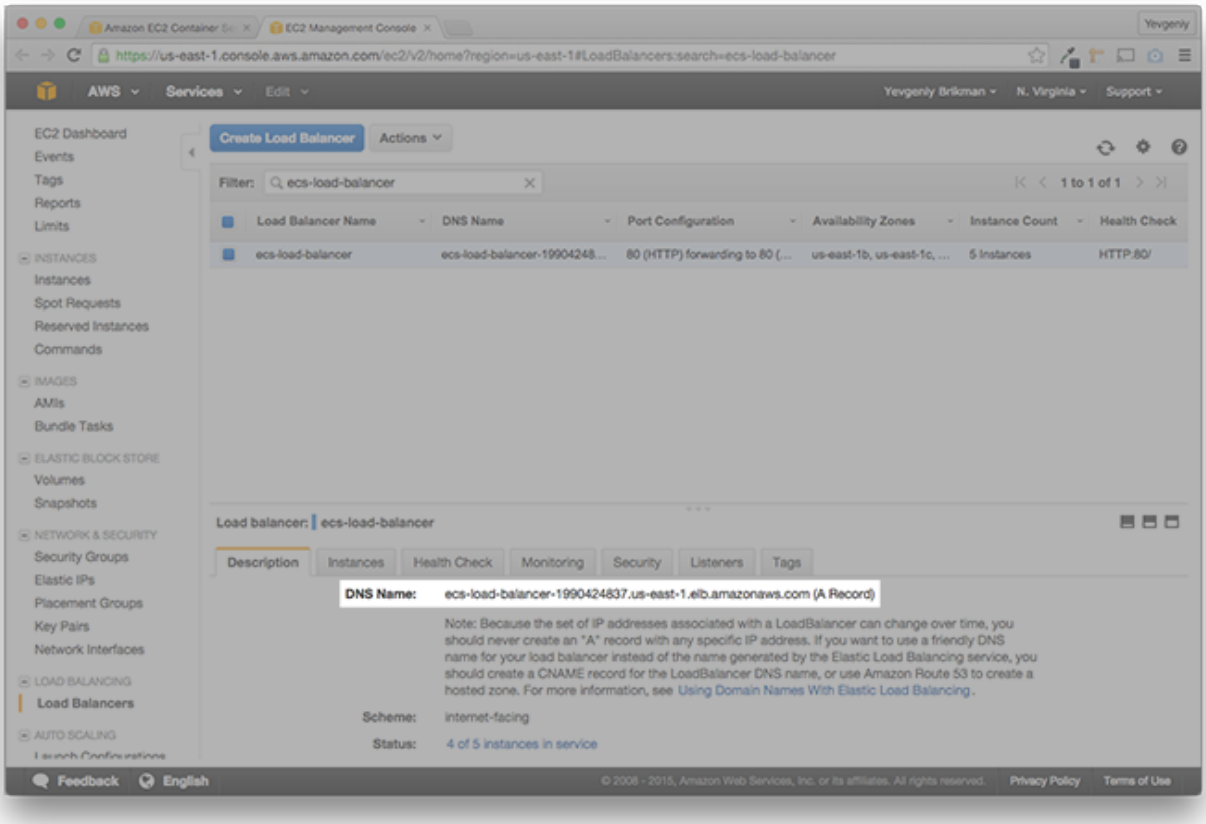
Your new ECS Service

Click the “Events” tab to see the deployment process:



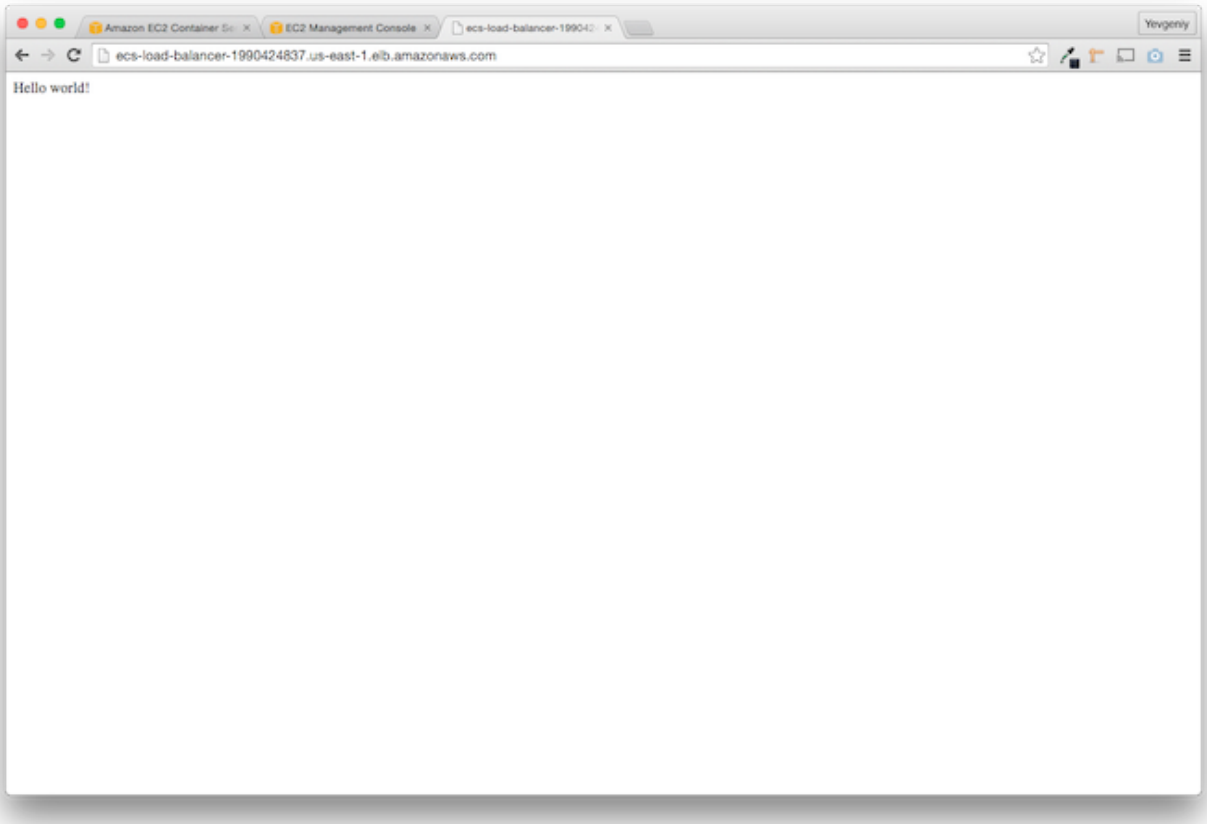
ECS Service deployment events

You may have to refresh a few times, but you should see your ECS Service starting 4 tasks, then registering 4 EC2 Instances in the ELB, and finally, reaching a "steady state", which means the deployment has completed. That means you now have 4 Docker containers running on 4 EC2 Instances and an ELB distributing load between them. To test it out, click on your ELB (the Events tab should make the ELB's name, `ecs-load-balancer` a clickable link) to go to the EC2 Console and copy its DNS Name:



ELB DNS

If you open this DNS in your browser, you should see the familiar `Hello world!` text:



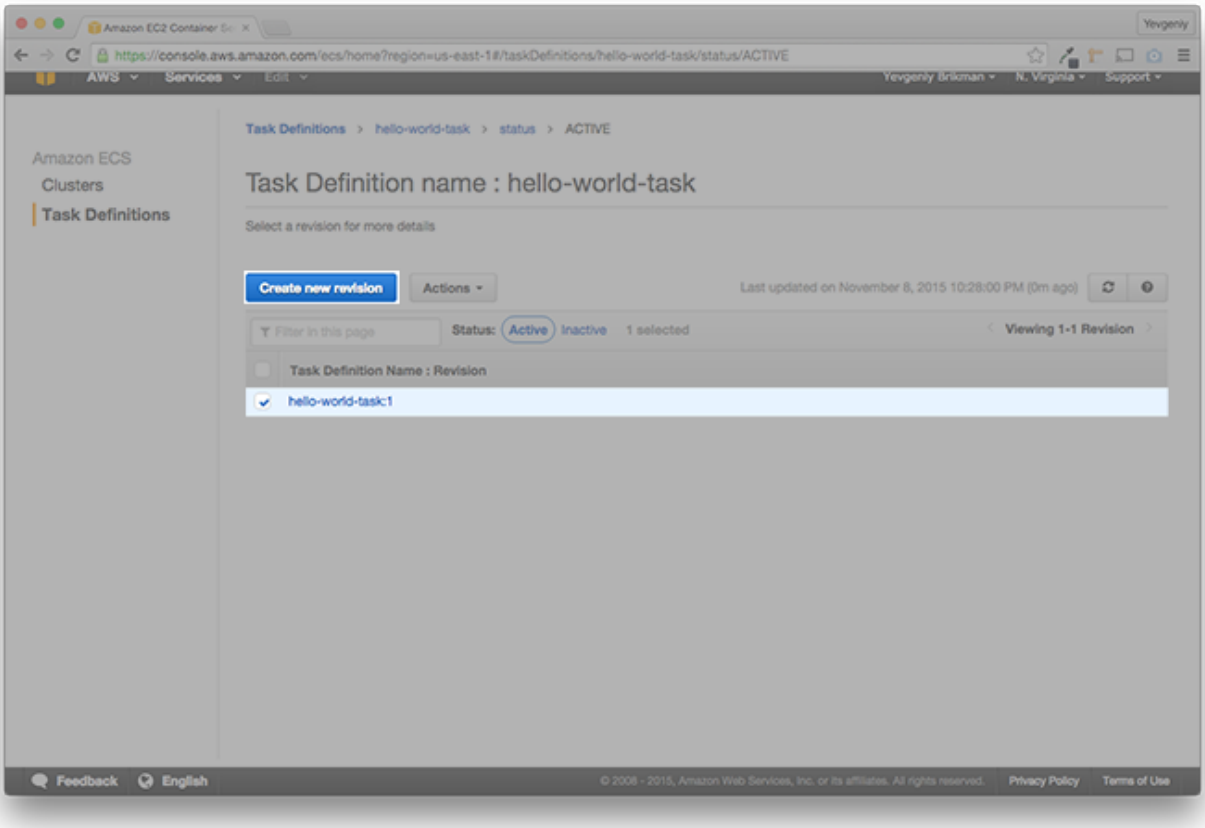
Accessing the app via the ELB

Update Docker containers in your ECS Cluster

Now that you have your Docker containers running in the ECS Cluster, let's go through an example of how you could update them to a new version. Normally, you would make some changes to your app, check them in, kick off a build, and produce a new Docker image with a new tag. However, for this example, the `training/webapp` Docker image only has the `latest` tag, so we'll update something else. If you look at the source of the training/webapp Docker container, you'll see that it uses the value of the environment variable `PROVIDER` as the second word after "Hello", and only falls back to "world" if `PROVIDER` is not set. Let's modify our ECS Task to set the `PROVIDER` value to `ECS` for our Docker container.

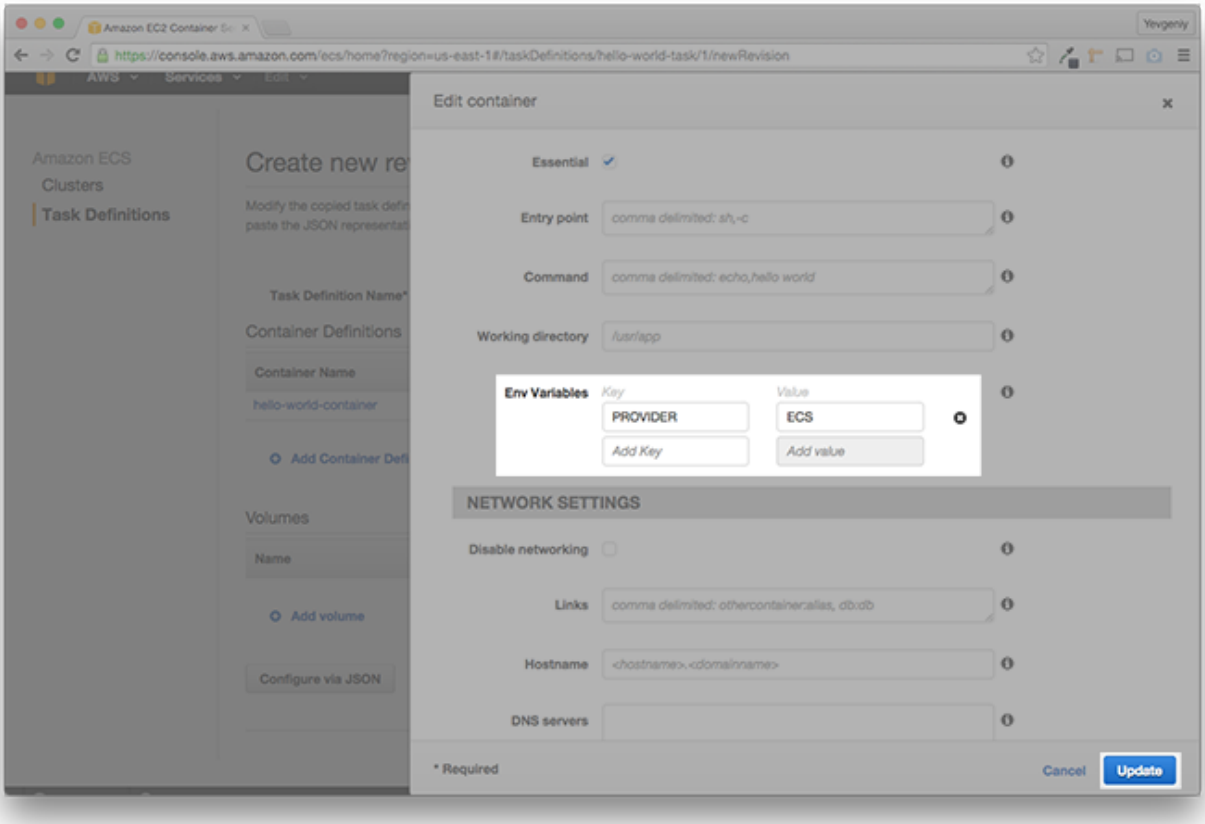
Note: ECS Tasks are immutable. You can't change the old definition—which is actually a good thing, as it allows you to easily roll back to an older version if you hit a big in a newer one—but you can create a new

Revision of the Task. To do that, click the "Task Definitions" link in the menu on the left, click on your ECS Task (`hello-world-task`), click the checkbox next to the first Revision of your Task (`hello-world-task:1`), and click the blue "Create new revision" button:



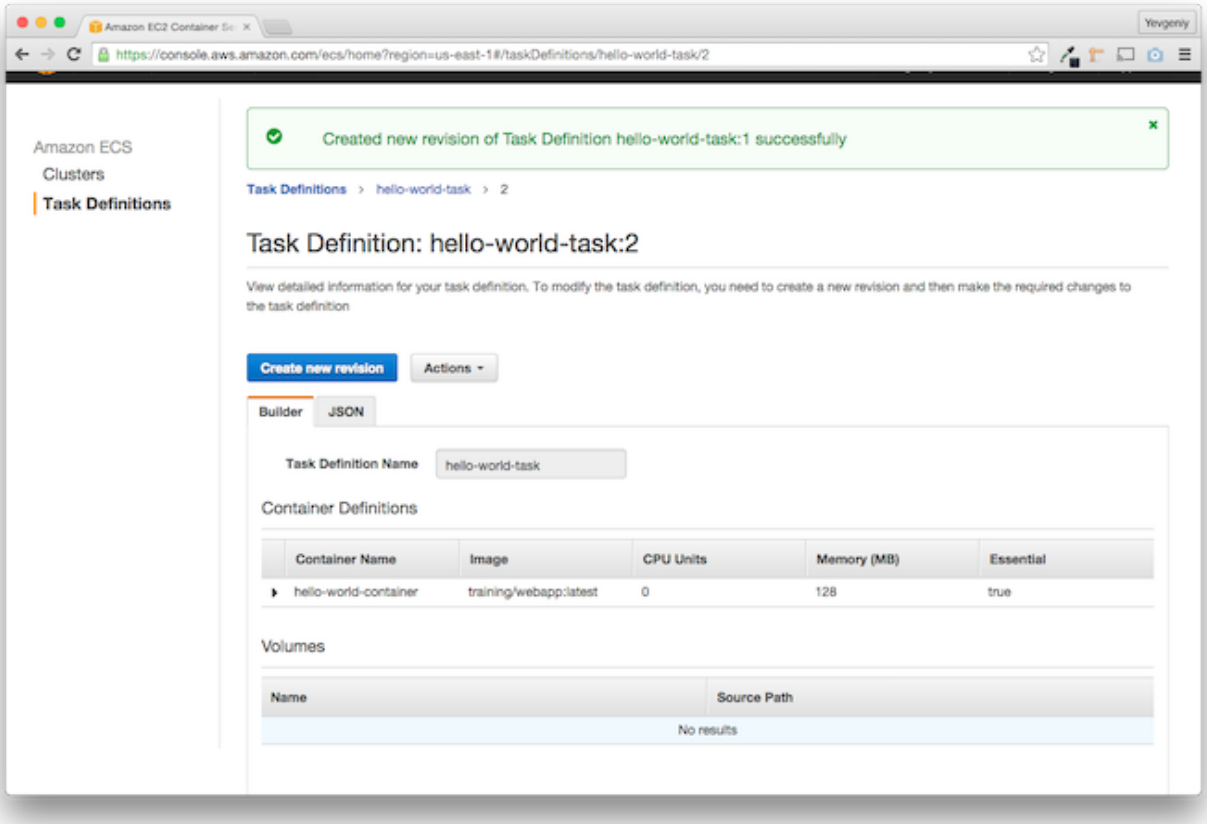
Create a new Revision of your ECS Task

Click on the Container Definition (`hello-world-container`), click the "Advanced container configuration" link, and scroll down to "Env Variables". Here, enter `PROVIDER` as the Key and `ECS` as the Value:



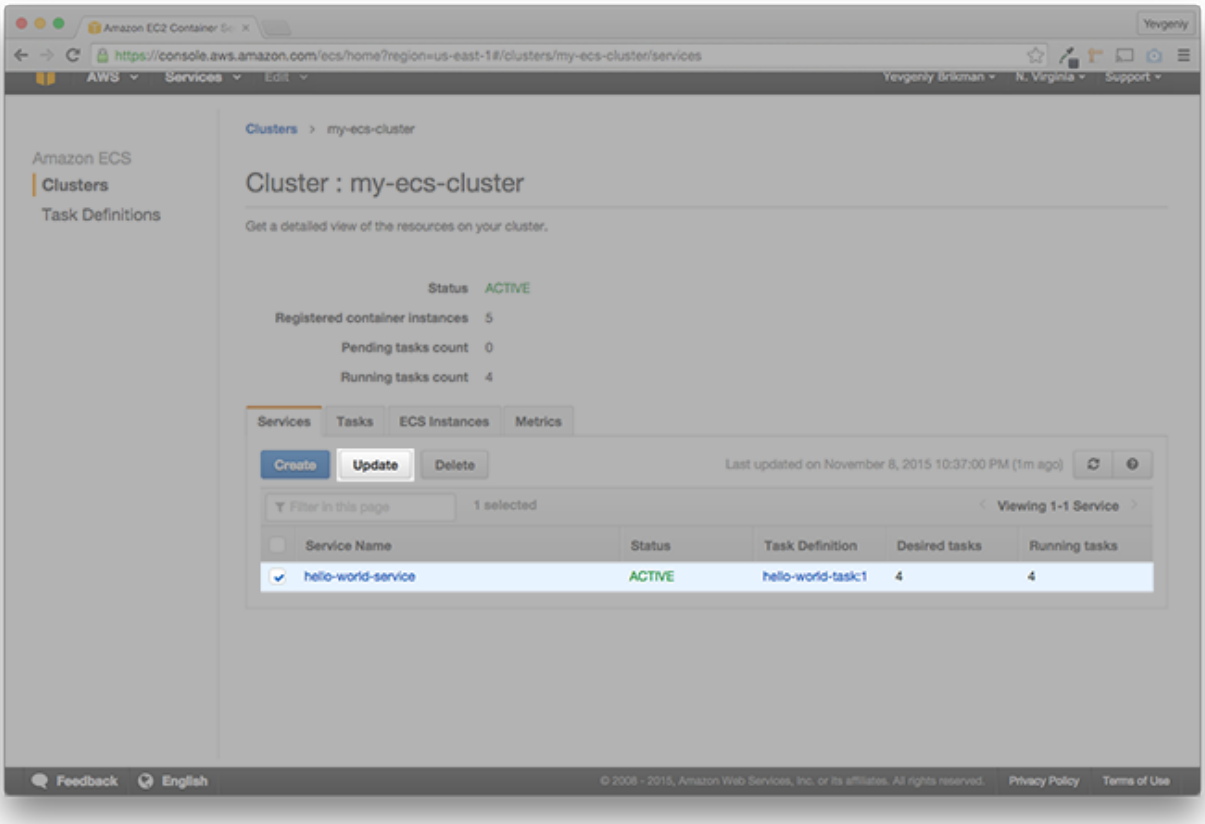
Add an environment variable to the container definition

This would also be the time to enter the new tag of your updated Docker image, but for this tutorial, we're going to stick with `latest`, so just click the blue "Update" button, then the blue "Create" button, and you should now have Revision 2 of your Task:



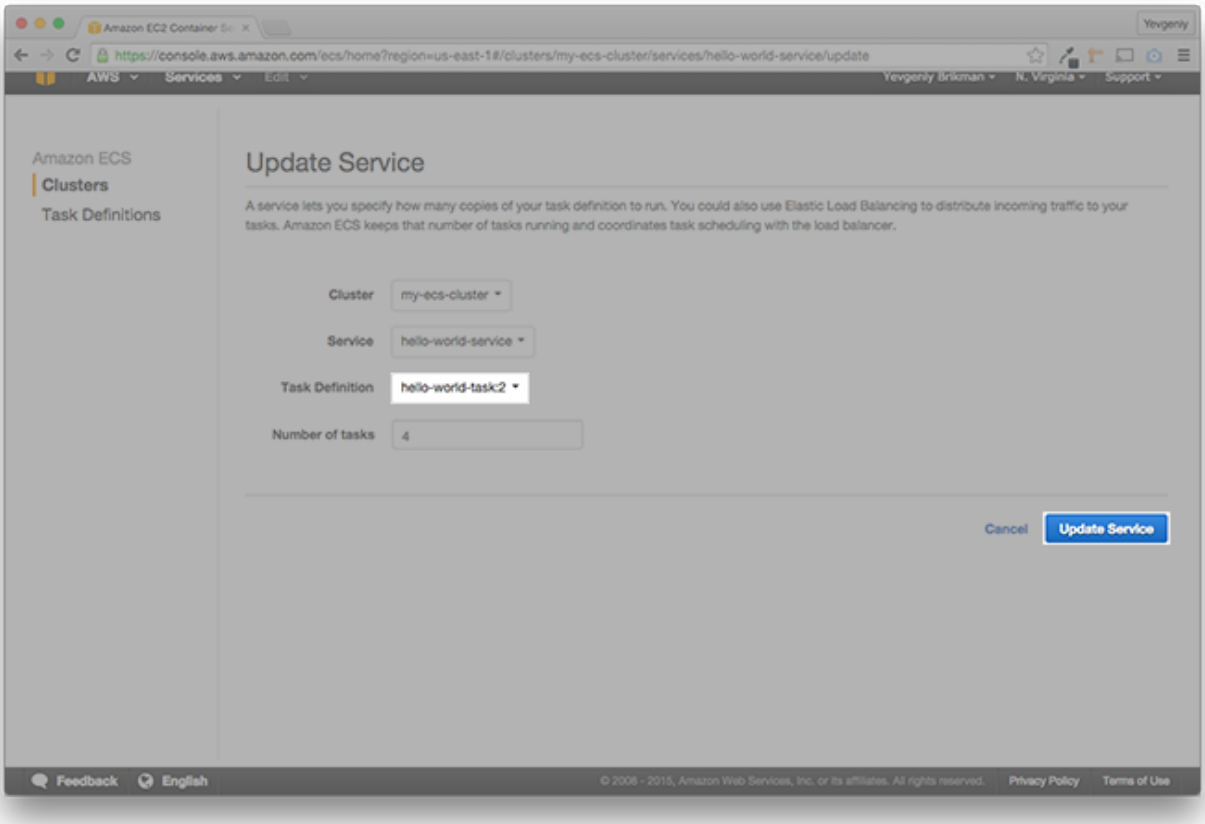
Revision 2 of the ECS Task

Now it's time to deploy the new Revision in the ECS Service. Click the "Clusters" link in the menu on the left, then click on your ECS Cluster (`my-ecs-cluster`), then click the checkbox next to your ECS Service (`hello-world-service`), and click the gray "Update" button:



Update ECS Service

Change the Task Definition from Revision 1 (`hello-world-task:1`) to Revision 2 (`hello-world-task:2`) and click the blue "Update Service" button:



Update the Task Revision in the ECS Service

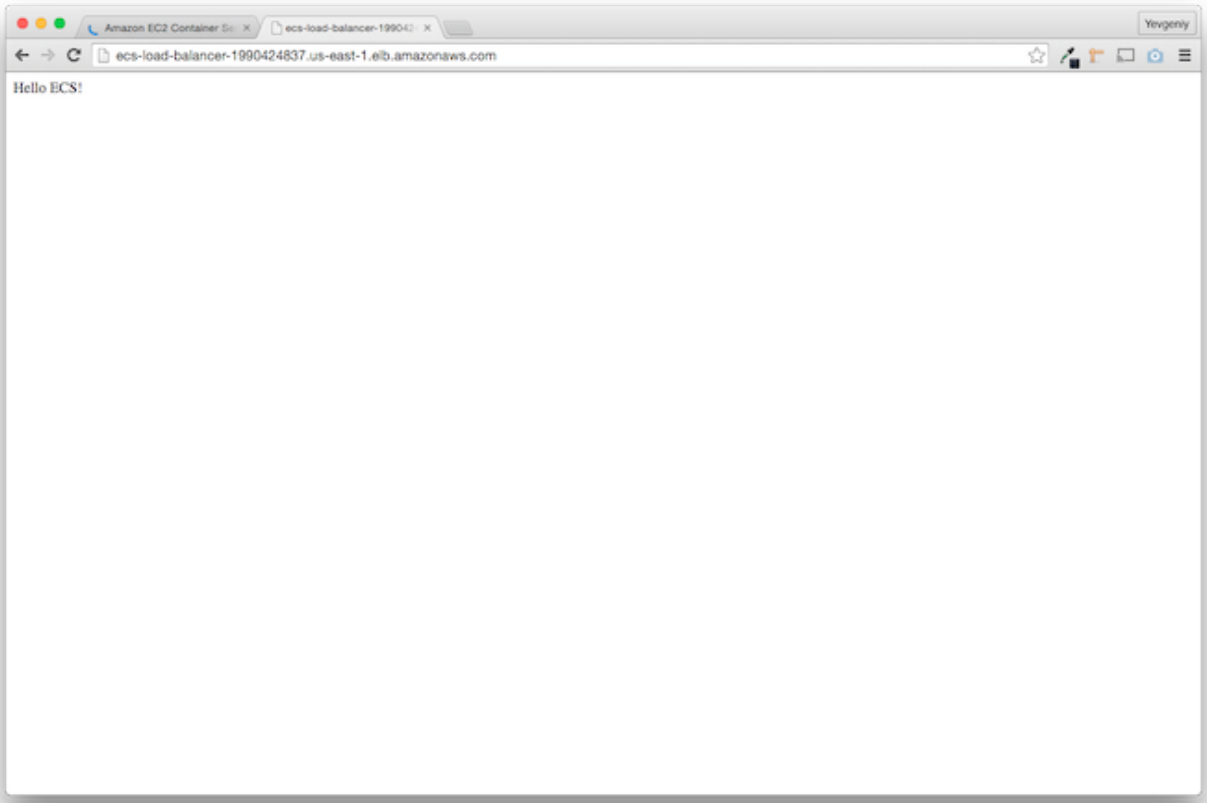
That's it! Now your new Task is deploying. If you click on the "Events" tab, you will see the ECS deployment process, which is roughly the following:

1. Look at the Container Definition in the Task to find out what CPU, memory, and ports it is requesting.
2. Find an EC2 Instance in the ECS Cluster that has the requested CPU, memory, and ports available. Finding the optimal server to run each Task is called *Task Scheduling*. In fact, since Docker containers provide isolation, if an EC2 Instance is not using all of its CPU or memory, a Scheduler could even decide to run multiple Docker containers on the same EC2 Instance, making more efficient usage of your resources.
3. If there is no available EC2 Instance that has the requested resources, show the error "service hello-world-service was unable to place a task because the resources could not be found" and try again in the future.

4. If there is an available EC2 Instance with the requested resources, run the new Task Revision on this Instance.
5. When the new Task is up and running, register it in the ELB.
6. Start draining connections for an old Revision of the Task.
7. Remove the old Revision of the Task from the ELB.
8. Stop the old Revision of the Task.

Steps #2 above is why you need to have more EC2 Instances in your Cluster (5) than desired Tasks in the ECS Service (4). That way, there is always at least one EC2 Instance available to deploy a new Revision of your Tasks. If you refresh the "Events" tab periodically, you'll see it go through the same cycle 4 times: it will start a new Task on the spare EC2 Instance, register it in the ELB, deregister an old Task from the ELB, and then stop the old Task. This is essentially a "rolling deployment", where you are updating one EC2 Instance at a time. If you have twice as many EC2 Instances as Tasks (e.g., 8 EC2 Instances and 4 desired Tasks), ECS will be able to do the update much faster, similar to a blue-green deployment (however, you'll have to pay for twice as many EC2 Instances for this luxury).

After a minute or two, if you go to the URL of your ELB, you should see the new `Hello ECS!` text:



The new Task Revision is working

Advantages of ECS

It takes *many* steps to set it up, but using an Auto Scaling Group, ELB, and ECS offers many benefits:

1. Every time you refresh the page, the ELB will send your request to a different EC2 Instance, so the load will be evenly distributed across all the servers in your Cluster.
2. ECS is monitoring the status of your Docker containers, so if one goes down, ECS will automatically deploy a new one.
3. Similarly, the Auto Scaling Group is monitoring the status of your EC2 Instances, so if one goes down, it will automatically deploy a new one. Once the new Instance is up, ECS will automatically deploy Docker containers onto it.
4. Since user requests only go to the ELB, most of your down time is

hidden from users, since the ELB will only route requests to a server that is up and running.

5. ECS can do automatic, zero-downtime deployments of new versions of your Docker image.

Of course, most importantly, the deployment process is no longer manual. You just create a new Revision of your ECS Task, update your ECS Service to that new Revision, and all the deployment details are handled for you automatically.

Disadvantages of ECS

I have a bunch of minor gripes with ECS—e.g., it takes a lot of steps to get started, the documentation is not great unless you already know what to look for, and it doesn't provide enough visibility into what's going wrong when you make a mistake—but the biggest disadvantage of ECS is that it does not integrate with Auto Scaling. That is, Auto Scaling works just fine to scale your *EC2 Instances* up or down, but it does not have any direct effect on the number of desired *ECS Tasks* you run across those Instances. So your Auto Scaling Group going from 5 to 10 EC2 Instances during peak usage hours will have no effect if your ECS Service still has desired Tasks set to 4.

There are two possible workarounds for this problem:

1. **Notifications + Lambda**: As described in this post on the AWS Blog, you can create a notification every time an Auto Scaling Event is about to take place and create an AWS Lambda Job that listens for those notifications and changes the number of desired Tasks accordingly.
2. **Max out desired Tasks**: Set the number of desired Tasks in your ECS Service to a number *much* higher than the number of EC2 Instances in your Cluster, so if Auto Scaling adds more EC2 Instances, the ECS Scheduler will automatically fill that extra capacity with running

Tasks.

The problem with both of these workarounds, besides the fact that they require extra work & maintenance on your behalf, is that, unless you take special care, they will interfere with the way an ECS Service deploys new revisions of your Tasks. If the number of desired Tasks is equal to the number of EC2 Instances, then you won't have any more room to deploy a new version of your Task. In theory, this shouldn't be a problem, as you should be able to deploy different versions of the same Task on a single EC2 Instance, but if you are using ELB—which can only do a 1:1 port mapping—this won't work (and the ECS Scheduler won't even attempt it) because both of those Tasks will be requesting the same port (see this thread on the AWS Forums for more info).

If you really want to use Auto Scaling and ECS and avoid hacky workarounds, there are two other options, although they are quite a bit more involved:

1. **Don't use ELB**. Instead, use a load balancer, such as HAProxy or nginx, that supports URL-based routing (e.g. route `/foo` to port 8080 and `/bar` to port 8081). You would need to configure your ECS Tasks to pick a random port number (apparently, you can do this by setting the port number to 0 in the Container Definition) and have those Tasks register their path(s) and port(s) with the load balancer when they boot up. The downside here is you'll have to replicate all the features AWS gives you for free with ELB, such as high availability, elastic scaling, health checks, and integration with other AWS services (e.g. CloudWatch, Auto Scaling, etc).
2. **Don't use the ECS Scheduler**. ECS allows you to swap out its Scheduler for your own. You can find instructions in the blog post How to create a custom scheduler for Amazon ECS. Some of the alternative Schedulers you could consider include Fleet, Marathon, Kubernetes, and Mesos.

Conclusion

Before I jump into some final thoughts, an important reminder: you should probably shut down any of the ECS Tasks and EC2 Instances you created during this tutorial so you don't get charged for them. To do that, first, go to the ECS Console, find your ECS Service, and update it to set the number of desired Tasks to 0. Once all the Tasks are stopped, you can delete the ECS Service and Cluster. After that, go to the EC2 Console, and delete any Auto Scaling Groups, Load Balancers, and terminate any EC2 Instances (in that order).

With that out of the way, let's talk a little more about Docker deployment. The first thing to say is that this blog post only touches on one of *many* options. In addition to the Schedulers mentioned above, some of the other options include Docker Swarm, Deis, DigitalOcean's Docker supoprt, and Tutum. For a good comparison, check out Choosing the Right Framework for Running Docker Containers in Production.

My own take is that while Docker is a fantastic tool and the future of DevOps, it is a relatively young technology and the ecosystem around it is still immature. Be prepared for bugs, missing features, unnecessary complexity, and poor documentation. When faced with such an ecosystem (the JavaScript MVC ecosystem is another one), I usually try to go with simplest solution that can possibly work. That is, something that I can understand fully, teach to others, maintain, debug, and evolve. By these measures, the only Docker deployment tools that meet my bar are (a) ECS and (b) DIY automation.

ECS, believe it or not, is one of the *simplest* Schedulers out there. Most of the other alternatives I've tried offer all sorts of fancy bells & whistles, but they are either significantly more complicated to understand (lots of new concepts), take too much effort to set up (lots of new technologies to install and run), are too magical (and therefore impossible to debug), or some combination of all three. That said, ECS also leaves a lot to be

desired.

The good news is that the Docker ecosystem is improving at an incredible rate, so it'll be interesting to revisit this question in 6-12 months to see how everything has progressed. I'm especially keeping my eye on Tutum, as they have been acquired by Docker, and may become the officially recommended solution.

If you've made it this far, check out my next blog post in this series to learn how to automate this entire process: Infrastructure as code: running microservices on AWS using Docker, Terraform, and ECS.

PREVIOUS

Should I start a company or work for one?

NEXT

Shit Google Voice Says

Yevgeniy Brikman

Did you enjoy this post? If so, check out my book, Hello, Startup. Need help with DevOps or scaling your infrastructure? Reach out to me at Atomic Squirrel.

Share this post

Comments

36 Comments Yevgeniy Brikman Home Page 1 Login

Recommend 4 Share Sort by Best

Join the discussion…

Petri Sirkkala · 5 months ago

At auto scaling creation my AWS returned a failure due to unaccepted AWS Marketplace product terms for the "Amazon ECS-Optimized Amazon Linux AMI".

So I had to revisit the link given there and confirm the order. ;)

2 · Reply · Share ›

Yevgeniy Brikman Mod → Petri Sirkkala · 5 months ago

Ah, good catch. Thanks for pointing that out!

Reply · Share ›

jeff · 4 days ago

If you have an application which runs within a single Docker container, does it always make sense to run within an ECS cluster? Is there an alternative to using ECS if you nonetheless want to deploy Docker in AWS?

Reply · Share ›

Yevgeniy Brikman Mod → jeff · 4 days ago

If you're just running a single server, it's not worth the overhead of running ECS. Just do "docker run" yourself :)

Reply · Share ›

jeff · 4 days ago

At the beginning of the tutorial the command curl http://dockerhost:5000 fails with can't resolve host dockerhost; curl http://localhost:5000 does work. Is this expected?

Reply · Share ›

Yevgeniy Brikman Mod ➚ jeff · 4 days ago

If you're on Linux, use localhost. If you're on OS X, and using docker-osx-dev (https://github.com/brikis98/do..., use dockerhost.

^ | v · Reply · Share ›

rory cawley · 9 days ago

Thanks for writing this! Going to give Docker and AWS a go now.

^ | v · Reply · Share ›

Aurélien Massiot · a month ago

Thanks a lot, really helpful.

^ | v · Reply · Share ›

Mr Shizzing · a month ago

Thanks very much, would have taken days and days of fiddling to get to the point you show here

^ | v · Reply · Share ›

River Lune · a month ago

Early in the article, you invite users to create `my-ec2-key-pair`, but later in the "Create an Auto Scaling Group" section it looks like you have a typo indicating that users should select `test-ec2-key-pair`...

^ | v · Reply · Share ›

Yevgeniy Brikman Mod ➚ River Lune · a month ago

Thanks, fixed!

^ | v · Reply · Share ›

Adrien Candiotti · a month ago

Brilliant tutorial, really.
But what did you mean when you said "ECS still has a lot to desire"?
What did you miss when using it?

And why do you recommand a custom deployment solution to small startups?
The price? Or maybe it's a bit overkill to start with?

^ | v · Reply · Share ›

Yevgeniy Brikman Mod ➚ Adrien Candiotti · a month ago

I listed some of the limitations with ECS in the blog post, not the least of which is that it's hard to figure out how to use it, so I had to write the blog post in the first place :)

ECS is useful when you have lots of Docker containers, ASGs, and other things to manage. When you have just a single app to deploy and you run one instance of that app per server, ECS is probably overkill, and it's usually easier to deploy without it.

1 · Reply · Share ›

Jon · 2 months ago

This is a fantastic tutorial, thank you for writing it.

I didn't see a discussion of maximumPercent and minimumHealthyPercent. It seems like they would have some bearing on the major drawback presented here, regarding auto scaling deployments and updated task definitions. Could you have maximumPercent at 200, and max out desired tasks, so that during a deploy new tasks with the updated revision are spun up?

There is a discussion of this on the AWS blog here: https://aws.amazon.com/blogs/c...

· Reply · Share ›

Yevgeniy Brikman Mod → Jon · 2 months ago

Oooh, interesting. I haven't experimented with maximumPercent and minimumHealthPercent (are they new?), but they look promising. I'll give them a shot, thanks!

· Reply · Share ›

Yevgeniy Brikman Mod → Yevgeniy Brikman · 2 months ago

Ah, just saw in your link that those are relatively new features. Awesome!

· Reply · Share ›

Kiran Koduru · 2 months ago

Thank you! You are a life saver.

· Reply · Share ›

bhaaratman · 2 months ago

Will ECS allow pulling docker image that is not on Docker Hub but, say, hosted locally on our private infrastructure? I usually SCP my docker image into EC2 but am wondering if I can take advantage of ECS

· Reply · Share ›

Yevgeniy Brikman Mod → bhaaratman · 2 months ago

AFAIK, ECS will pull from anywhere the EC2 instance can access. So if your "private infrastructure" is in AWS already, or accessible from

AWS, then as long as you configure the ECS_ENGINE_AUTH_DATA parameter correctly (see http://docs.aws.amazon.com/Ama... and http://docs.aws.amazon.com/Ama...), it should work. I've tried it with private Docker Hub repos, and it works fine.

Reply · Share ›

ismet özöztürk · 2 months ago

In your scenario, as you configure ELB at Auto-scaling level, ELB will route traffic to individual EC2 instances and there should be one of a kind of task running at each instance. However if you had registered ELB at Service level, ELB would route traffic to individual Tasks, hence multiple tasks of same kind could run on the same instance and they could get traffic of their own individually. Am I right or missing something? Thank you for the post btw it helped a lot.

Reply · Share ›

Yevgeniy Brikman Mod ➚ ismet özöztürk · 2 months ago

Amazon's ELB only supports static port number mapping, such as mapping all HTTP requests (port 80) that come into the ELB to port 12345 of EC2 instances connect to the ELB. Once you've created that mapping, any app that wants to receive HTTP traffic, whether it runs natively on the EC2 instance or as a Docker container, must listen on port 12345. And as only one process on a server can listen on port 12345 at a time, by default, that means you can only run app (i.e. one Task) per server that wants HTTP traffic.

There are, of course, a variety of workarounds, such as using non-standard ports for incoming traffic to the ELB (which works great for backend services that don't directly receive traffic from users) or running a load balancer (e.g. HAProxy) on each EC2 instance that listens on port 12345 and internally distributes the traffic to a variety of Tasks running on the same instance that are all listening on different ports. I just wish AWS had a way to handle this common case automatically.

Reply · Share ›

ismet özöztürk ➚ Yevgeniy Brikman · 2 months ago

This clarified my mind thank you Yevgeniy. Do you have any comment on comparing ECS to Google's Container Engine?

Reply · Share ›

Yevgeniy Brikman Mod ➚ ismet özöztürk · 2 months ago

I haven't tried Google's Container Engine, so I can't say anything about it.

Reply · Share ›

dcboy · 4 months ago

How to create a private registry on ec2 which is accessible from outside the server (f.e. with selfsigned certificates?)

Reply · Share ›

Yevgeniy Brikman Mod ➚ dcboy · 4 months ago

There are several guides online:

http://www.elastic.io/en/runni...
https://blog.codeship.com/runn...

Reply · Share ›

dcboy ➚ Yevgeniy Brikman · 4 months ago

Thanks but I want just a registry which is using selfsigned certificates like this https://docs.docker.com/regist... but I'm so stuck. Did you ever do something?

Reply · Share ›

Yevgeniy Brikman Mod ➚ dcboy · 4 months ago

I'm not sure what you're struggling with. The guides above talk about setting up the registry and the Docker link describes self-signed certs. What's missing?

Reply · Share ›

dcboy ➚ Yevgeniy Brikman · 4 months ago

I always get this error when I try to push an image to my registry: net/http: TLS handshake timeout
While the logs of my registry are showing: level=info msg="listening on [::]:5000, tls" go.version=go1.5.2 instance.id=731ead25-53de-4209-9475-98001e70fb68 version=v2.2.1
2015/12/16 09:05:02 http: TLS handshake error from 10.0.0.xx:36816: EOF

Reply · Share ›

Yevgeniy Brikman Mod ➚ dcboy · 4 months ago

Not sure, sorry

Reply · Share ›

Long Le · 5 months ago

For me,

this doesn't work

#!/bin/bash

echo ECS_CLUSTER=my-ecs-cluster > /etc/ecs/ecs.config

this works

#!/bin/bash

echo ECS_CLUSTER=my-ecs-cluster >> /etc/ecs/ecs.config

Reply • Share ›

Yevgeniy Brikman Mod → Long Le • 5 months ago

`>>` appends to a file, where as `>` overwrites it. Either one should work, but did you already have an `ecs.config` file of your own that you needed to avoid overwriting?

Reply • Share ›

Long Le → Yevgeniy Brikman • 4 months ago

Hi Brikman, you're right ! Both work, `>` and `>>`. My problem is that I didn't assign Public IP for the instances when launching them from Auto Scaling Group, so them couldn't talk with ECS to register.
Thank you for your useful post !

Reply • Share ›

Vishnu Nair • 5 months ago

One doubt, Does ECS support images stored in private docker registry ? Or it only supports Docker hub?

Reply • Share ›

Yevgeniy Brikman Mod → Vishnu Nair • 5 months ago

Yes, it does: http://docs.aws.amazon.com/Ama...

Reply • Share ›

Esteban • 5 months ago

How would you manage persistent storage (for example for a clustered DB) in such environment?

Reply • Share ›

Yevgeniy Brikman Mod → Esteban • 5 months ago

You use data volumes. http://docs.aws.amazon.com/Ama...