Deploying ASP.NET Core to Fargate

# Overview

AWS Fargate is a technology for Amazon ECS that allows you to run containers without having to manage servers or clusters. With AWS Fargate, you no longer have to provision, configure, and scale clusters of virtual machines to run containers. Fargate supports Linux containers, so for this walk-through we’ll create a containerized ASP.NET Core MVC application in Visual Studio 2017, then use the publishing wizard to deploy it to Fargate.

*Note: You can also create the application and deploy to Fargate from Mac OS X or Linux using the command line. That’s outside the scope of this walk-through, but* ***we’ll provide the basic steps at the end if you want to try it****.*

We’ll complete the following tasks in order to get our ASP.NET Core MVC application running in Fargate containers behind a load balancer in AWS:

1. Create a new ASP.NET Core MVC web application
2. Publish our application using the AWS Toolkit’s publishing wizard for ECS
3. View the ECS cluster and Tasks in the AWS Toolkit plugin for Visual Studio
4. View our application in a browser via the container public IPs and also the ALB’s public DNS
5. *Optional: Creating and Publishing the App from the Command Line*

## Prerequisites

* .NET Core 2.0 or higher installed
* [Docker for Windows](https://docs.docker.com/docker-for-windows/?install_site=vsonwin)\*
* Visual Studio 2017 (the free Community Edition is sufficient)\*
* AWS Toolkit for Visual Studio\*
* AWS Account with credentials configured in Visual Studio

\*If you are following the steps in Task 5, “Perform this walk-thru from the command line”, you don’t need Visual Studio, and can deploy the application to Fargate from a Mac or Linux machine.

# 1. Create a new ASP.NET Core MVC Web Application

For this task, we’re just creating a new default ASP.NET Core MVC web application using the Microsoft-supplied project template, and then adding a line of code that writes the hostname of the container in which the app is running out to the UI as HTML.

1. In Visual Studio, from the menu, select *File 🡪 New 🡪 Project* to launch the New Project dialog.
2. In the New Project dialog, select the ASP.NET Core Web Application project type. See Figure 1 below.
3. Select a name for your project, and then click the “Ok” button.

Disregard the .NET Framework selection in the drop-down at the top of the dialog – the project will use .NET Core; this drop-down is a shortcoming of the Visual Studio IDE.

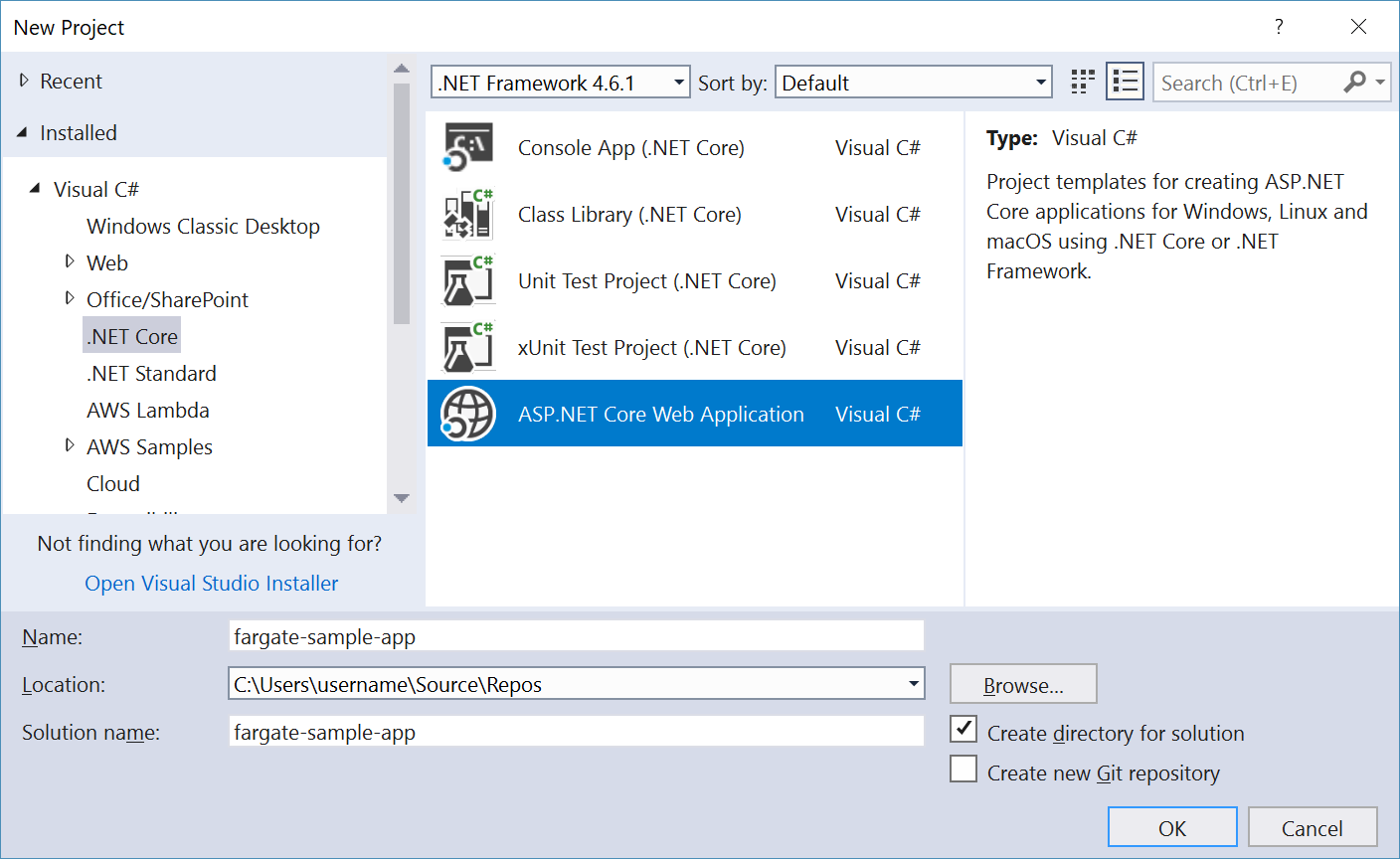


Figure 1 – New ASP.NET Core Web App in New Project Dialog

1. In the New ASP.NET Core Web Application dialog, select *Web Application (Model-View-Controller)*, ensuring that *ASP.NET Core 2.0* (or higher) is selected for the .NET Core version.  
   See Figure 2 below.
2. Check the box to Enable Docker Support, and select *Linux* as the container OS from the drop-down, then click the OK button to generate the project.

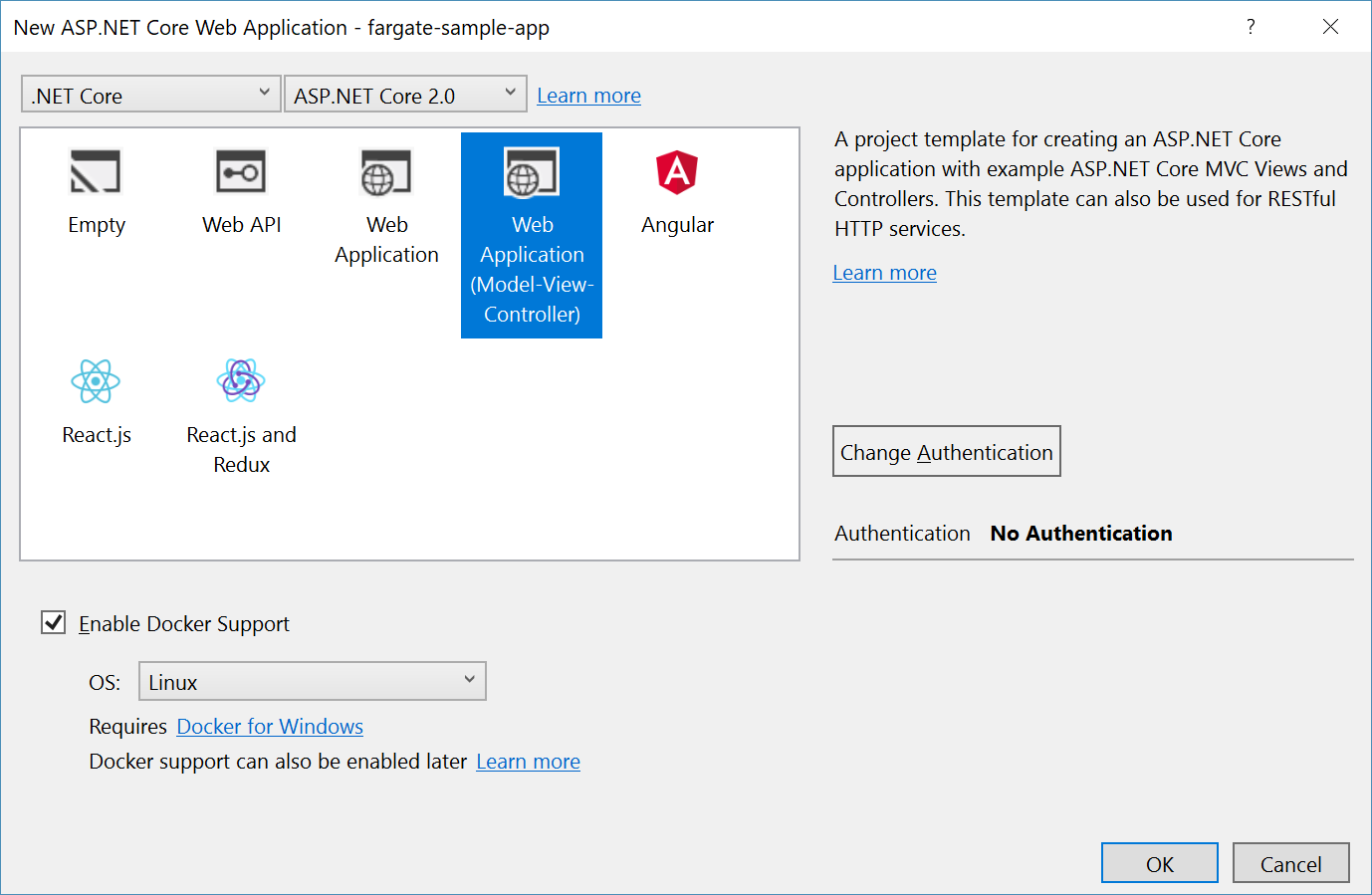


Figure 2 – Web Project Options Dialog

1. In the Solution Explorer pane, expand the project node for the app and navigate into the   
   Views 🡪 Home folder, and double-click the *Index.cshtml* file to open it in the editor. This is the HTML template file that will render the home page for our web application.
2. In *Index.cshtml*, locate the HTML DIV element with the id attribute of myCarasoul, and collapse it. The full HTML element looks like this:  
   <div id="myCarousel" class="carousel slide" data-ride="carousel" data-interval="6000">
3. After the closing tag for the div (if you have collapsed the element, then the next line), add the following to a new blank line, in order to render the hostname:  
   <h2>Hostname = @System.Environment.MachineName</h2>
4. Save your changes to the file.

*Note: If you want to test out the web project locally, you have two options. You can run it in a container by clicking the    button on the toolbar to build the container locally and run it (****warning****: this is slow). Or you can launch it using your configured web-server by right-clicking project node in Solution Explorer and selecting* Debug 🡪 Start new instance (*much faster).*

You’ve now completed this task, and are ready to deploy the application to Fargate.

# 2. Publish to Fargate using the AWS Toolkit for Visual Studio Wizard

Follow the steps below to deploy two containers running your application, in two AZs, behind an Application Load Balancer.

1. Right-click the project node in Solution Explorer then select *Publish Container to AWS*. This will launch the Publish Container to AWS wizard.
2. On the first step of the wizard, select US East (N. Virginia) as the region, and “*Service on an ECS Cluster*” as the Deployment Target, then click the *Next* button.  
   See Figure 3 below.

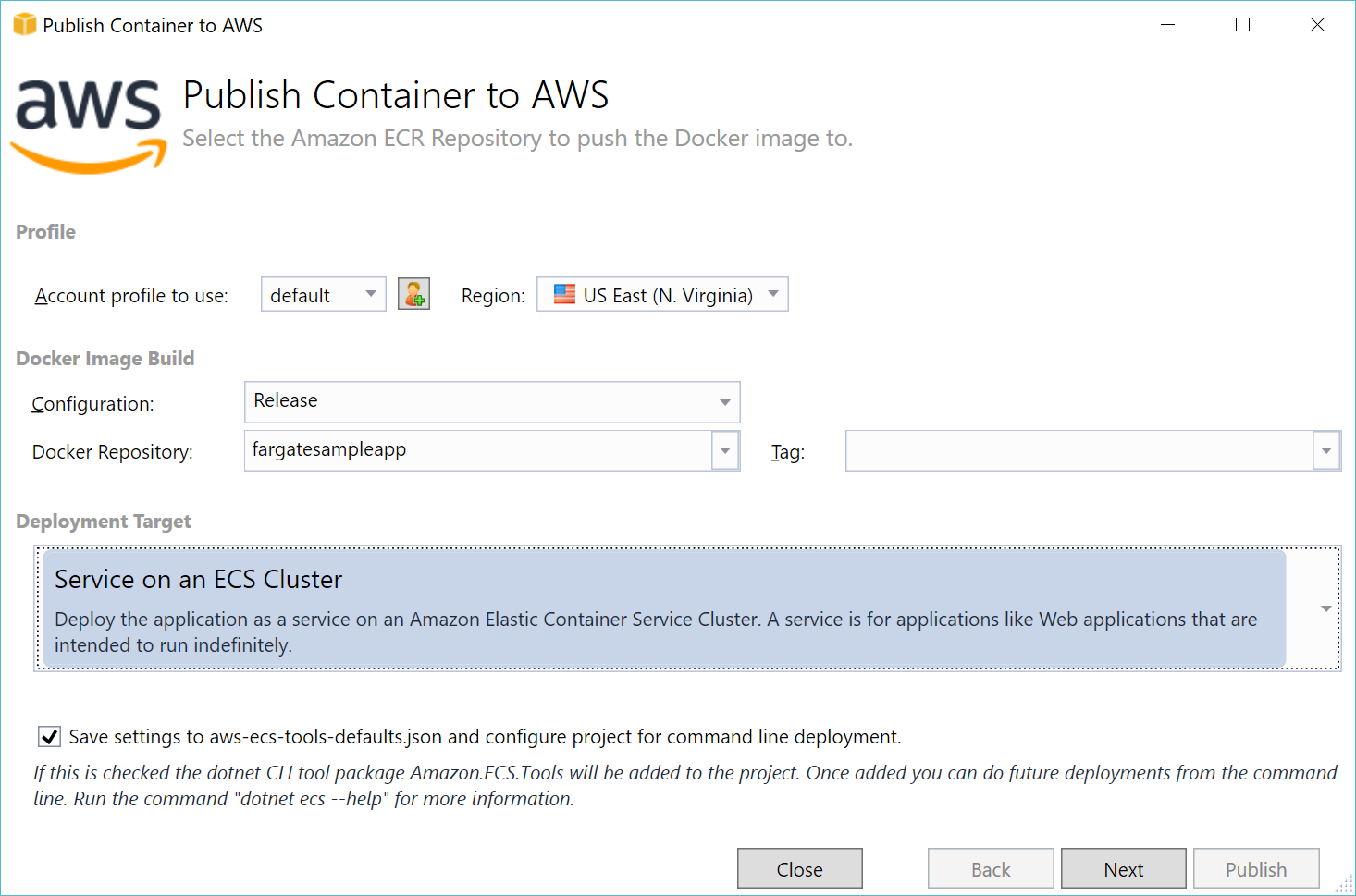


Figure 3 – Container Deployment Wizard, Step 1.

1. On the Launch Configuration step, choose “*Create an empty cluster*” (unless you already have a cluster you intend to use), and provide a name for the cluster. See Figure 4 below.
2. If the Launch Type does not default to *FARGATE*, select *FARGATE*.

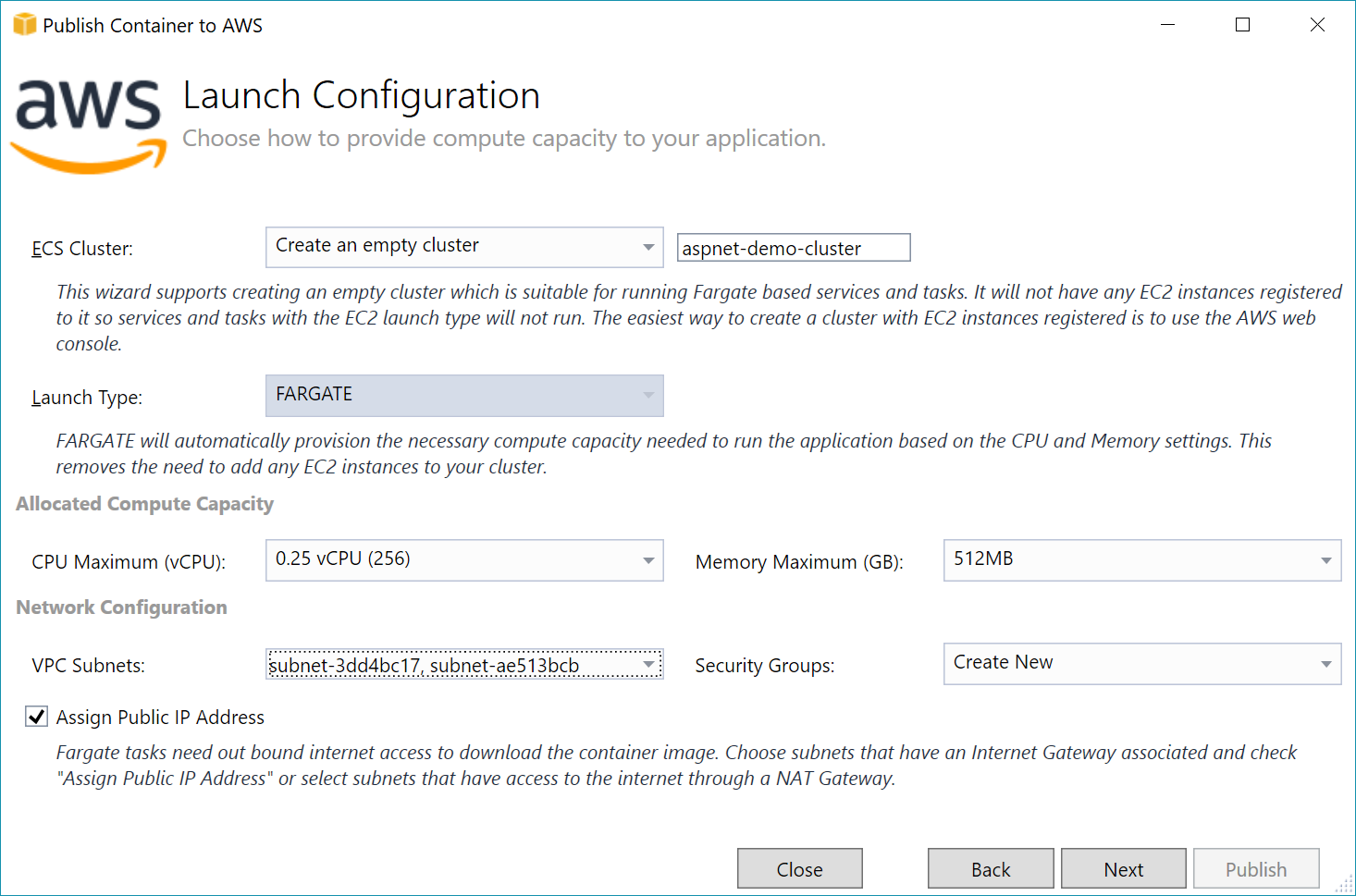


Figure 4 – Launch Configuration Step of Container Deployment Wizard

1. Under *Network Configuration*, select two subnets in two separate AZs. You can pick more than two, but for this walk-through, we’ll keep it simple and choose subnets in AZs “a” and “b”.
2. Select “*Create New*” next to Security Groups. This will create a new security group with TCP port 80 open to the world. If you are creating a container service for use inside a VPC only, choose a security group you created for that purpose outside of this walk-through.
3. Check the box to *Assign Public IP Address*, then click the *Next* button.

*Note: In a real-world scenario, we wouldn’t need to assign public IPs to each task (container) individually. Instead, we would access the services via the application load balancer. However, for demo purposes, it’s nice to be able to show that you can reach each task (container) directly by its public IP as well as via the ALB’s public FQDN.*

1. On the Service Configuration step, choose *Create New* for the service, and provide a name for your service. In Figure 5 below, the service will be called *fargatesampleapp*.
2. For the number of tasks, we’ll enter *2*, which will place one task in each of the two AZs we selected earlier. Leave the other numbers as the defaults and click the *Next* button. See Figure 5 below.

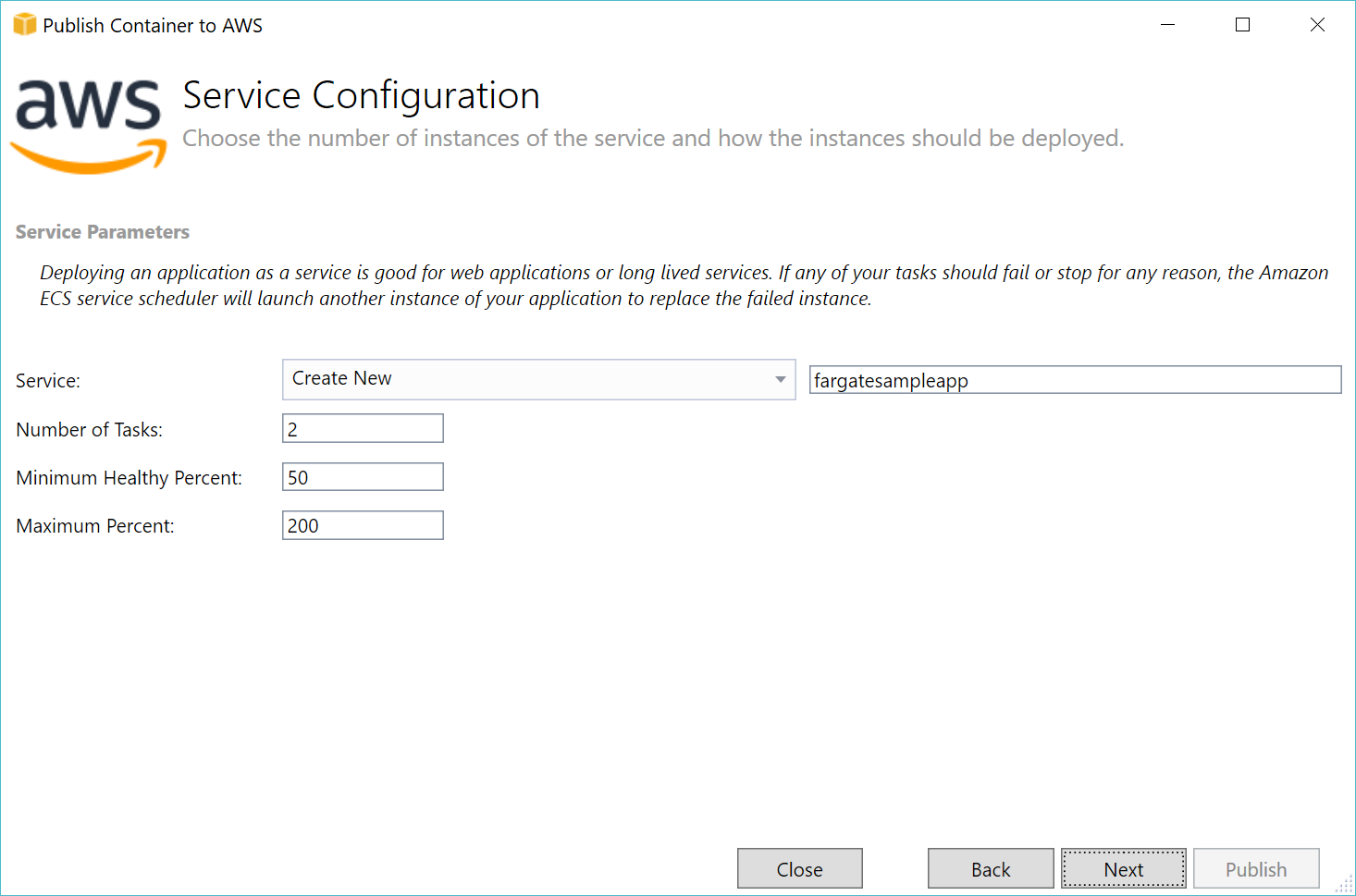


Figure 5 – Service Configuration Step of Container Deployment Wizard

1. On the Application Load Balancer Configuration step, check the box next to *Configure Application Load Balancer*.
2. Choose *Create New* in the load balancer drop-down and provide a name for your load balancer.
3. Enter *80* as the listener port in the box to the right of the (defaulted) *Create New* selection. See Figure 6 below.

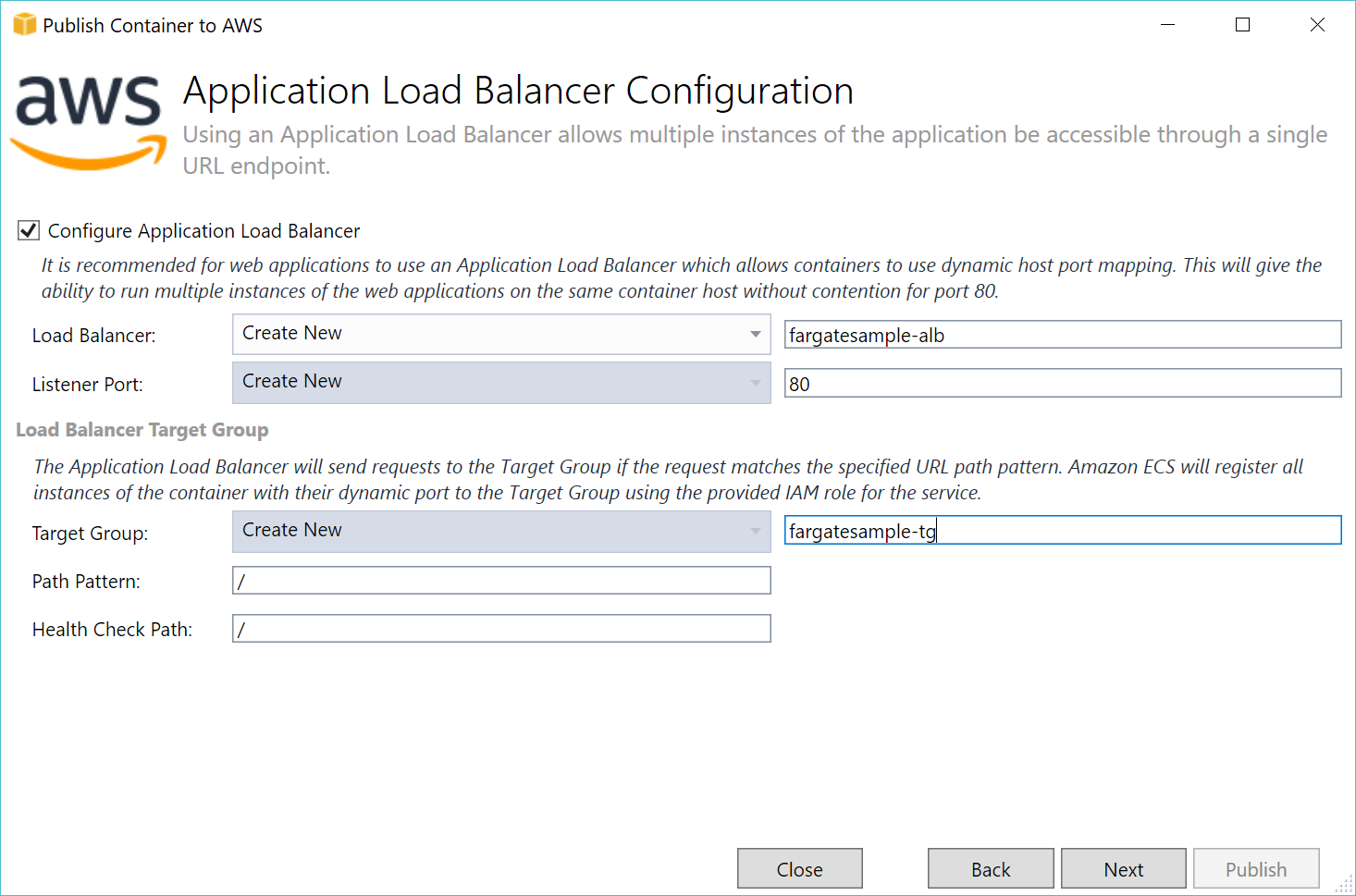


Figure 6 – Application Load Balancer Configuration step of Container Deployment Wizard

1. In the Load Balancer Target Group section, enter a name for the new target group into which your Tasks (containers) will be placed. In the example in Figure 6, we have chosen *fargatesample-tg* as the name.
2. Leave the other entries as their defaults and click the *Next* button.
3. In the Task Definition step, choose *Create New* for the task definition and enter a name for it. In the example in Figure 7, we have chosen *fargatesample-task* as the name.
4. Enter a name for the container that will run inside of the task. In the example in Figure 7, we have chosen *fargatesample*.

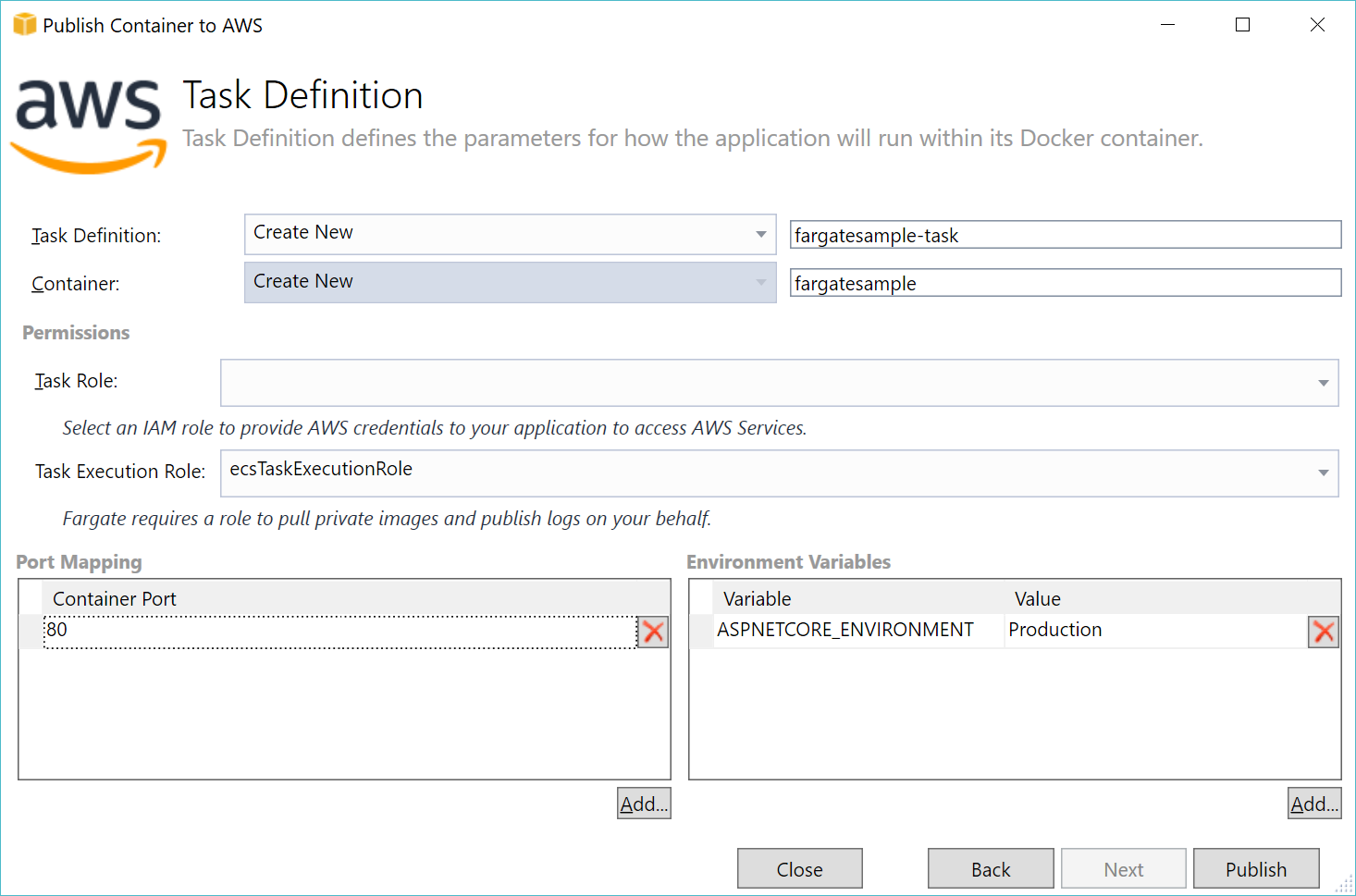


Figure 7 – Task Definition step of Container Deployment Wizard

1. Leave the Task Role selection blank since our app doesn’t need to access any AWS services. Leave the Task Execution Role to the default value.
2. The Port Mapping should already have default values of port 80, and *ASPNETCORE\_ENVIRONMENT* as an environment variable. Click the *Publish* button to begin the publish process.

During the publish process, the AWS Toolkit will call *docker-compose build* to build the solution and container image on your computer, then call the ECR API to get credentials for ECR. It will then tag the container image, and push it to ECR – you will see the container being pushed up by Docker in a command window that will appear. See Figure 8 below.

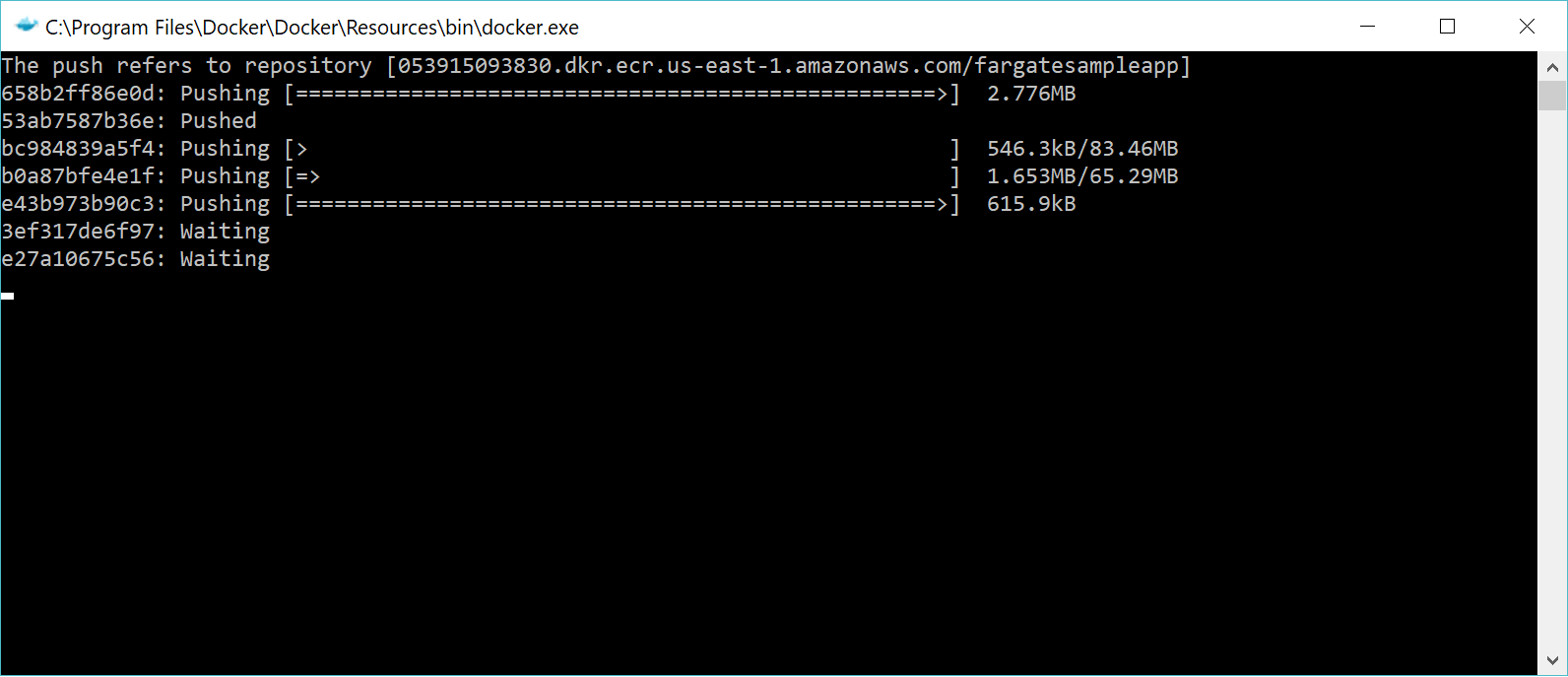


Figure 8 – Docker.exe in Windows Command Prompt window pushing to ECR

# 3. View the Cluster and Tasks in AWS Toolkit

In the AWS Explorer pane, find the *Elastic Container Service* node and expand it, then expand the *Clusters* node inside the ECS node. You should see the clusters in your currently-selected region. Double-click the cluster you created to open the ECS Cluster pane in Visual Studio. See Figure 9 below. If you don’t see your cluster, check that your region is set to *US East (N. Virginia)* in the AWS Explorer (near the top), and try clicking the refresh icon to refresh the list of clusters.

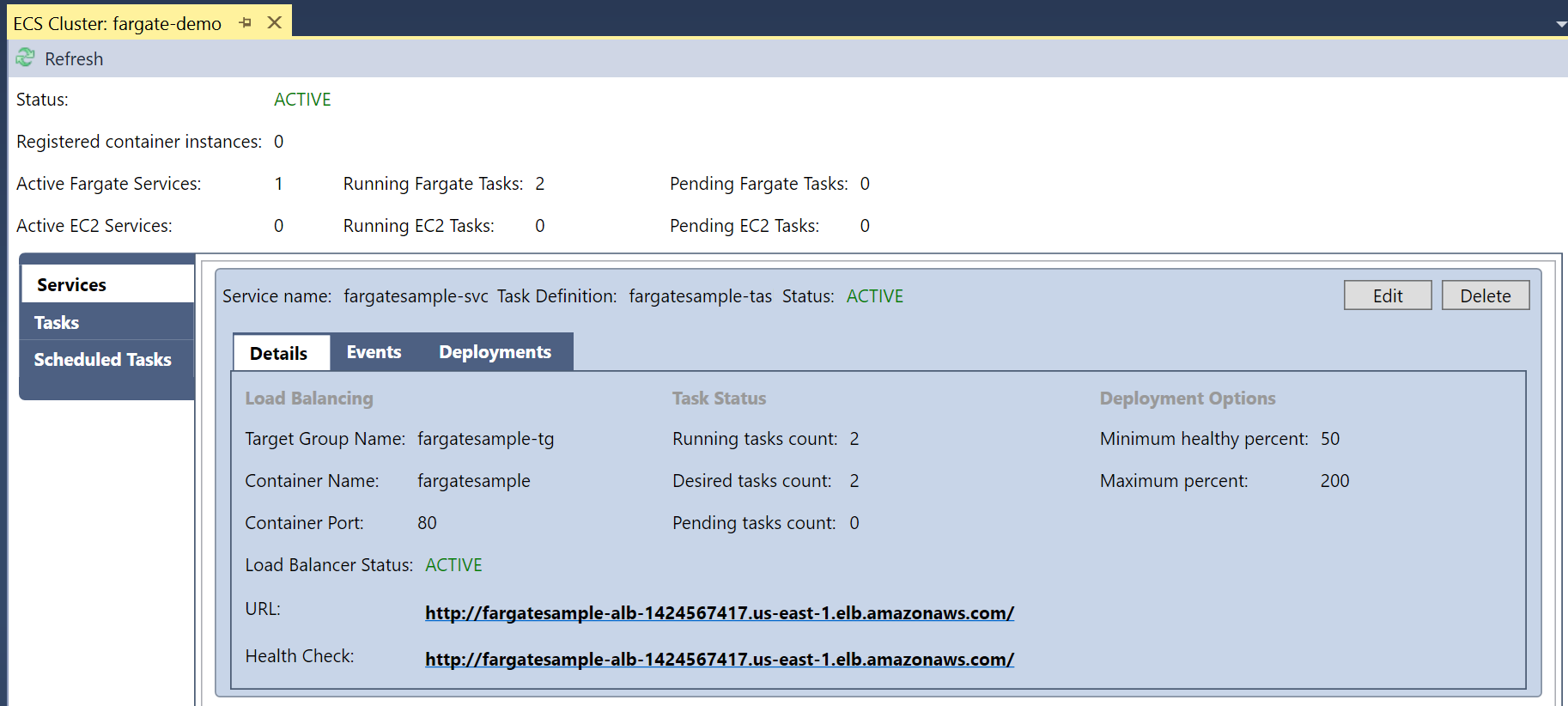


Figure 9 – ECS Cluster pane in Visual Studio (AWS Toolkit) showing Services details

The default selection in the ECS Cluster pane (in the shaded left menu) is “Services”. You should see your service here, including the *Desired tasks count* and *Running tasks count*. When the ALB is created, its URL will appear and be clickable. If you select *Tasks* from the left shaded menu, you will see the two tasks we created and their status. See Figure 10 below.

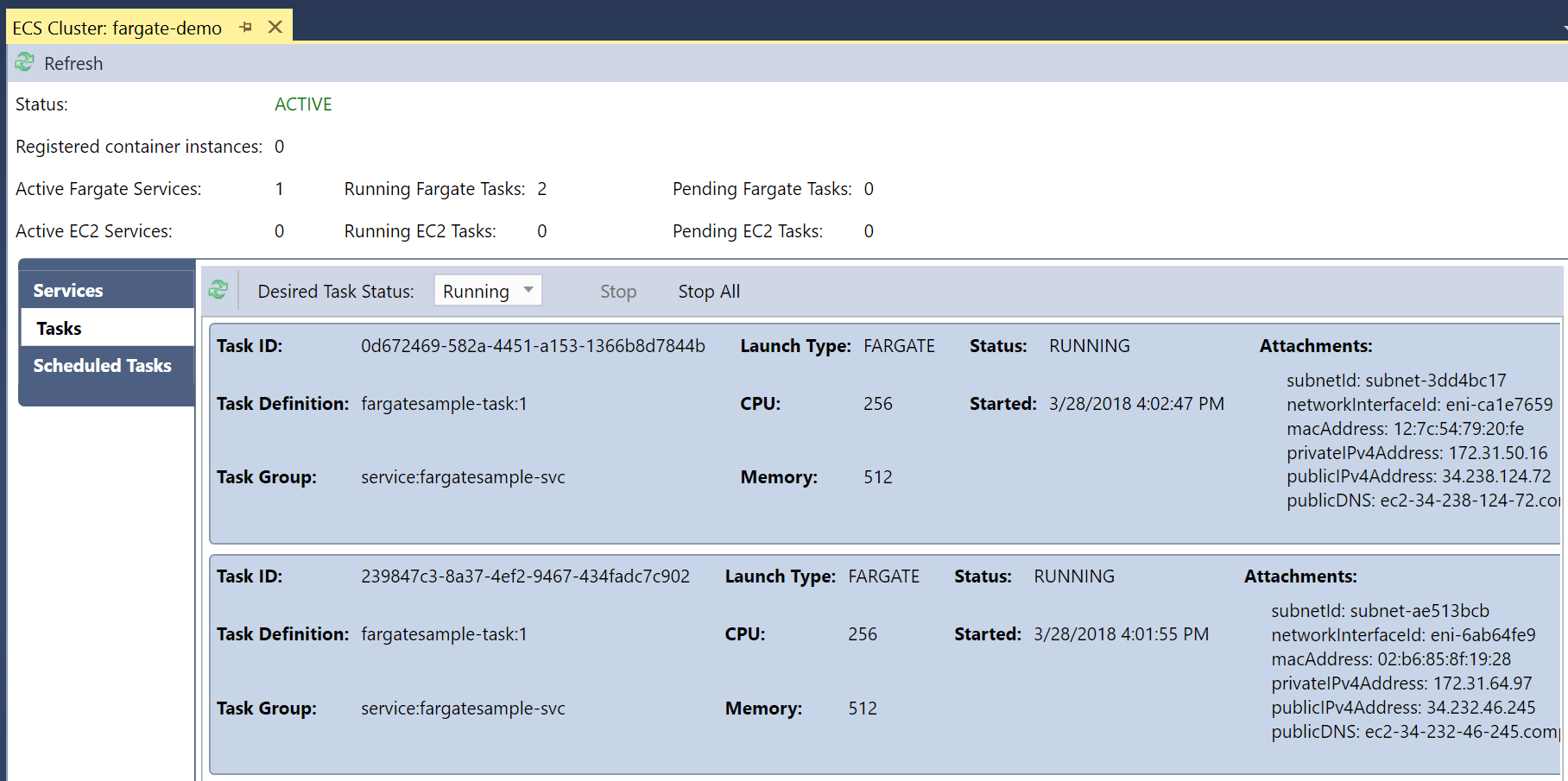


Figure 10 - ECS Cluster pane in Visual Studio (AWS Toolkit) showing Tasks details

# 4. View the Application in a Browser

1. In the ECS Cluster pane (see Figure 10), find the status of the tasks. Wait until either of the tasks shows a status of *RUNNING*. You may need to click the refresh icon that is above the tasks list.
2. At the right side of each task row, in the *Attachments* section, you will find the public IP address and DNS for the task. Copy the *publicIPv4Address* value or the *publicDNS* value (the DNS value resolves to the IP address, so use either) for a running task and paste it into a browser.
3. If the task is running, you should see the website which shows the hostname of the container in which the app is running.
4. Next, click back to *Services* in the shaded left menu, and click on the load balancer URL link. The website should open in a new tab or browser using the ALB’s public DNS. See Figure 11 below.
5. When both tasks are running, click the refresh button in your browser a few times. You should see the hostname that is displayed flip back and forth as your requests are routed to the two containers.

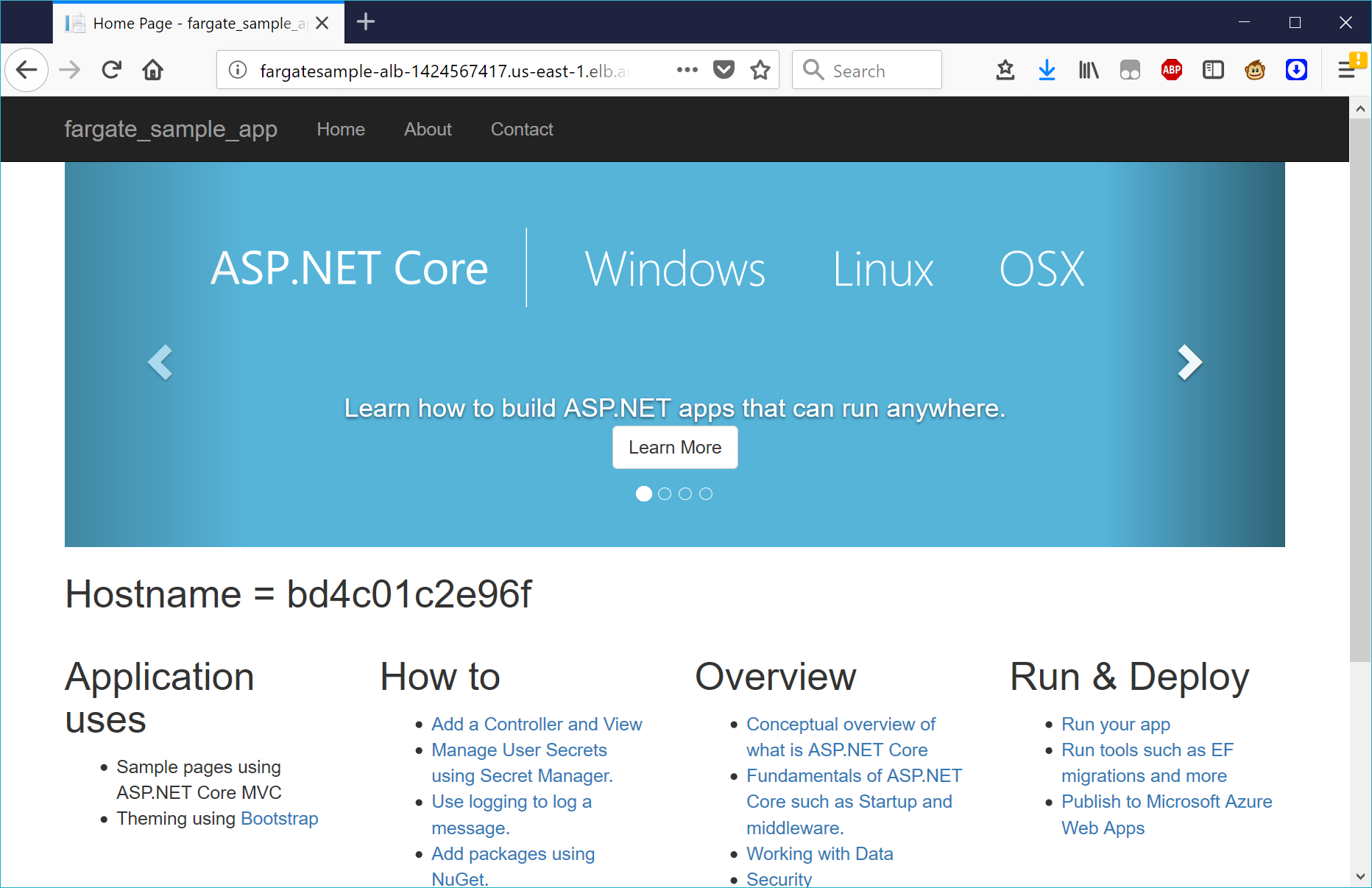


Figure 11 – the ASP.NET MVC app accessed via the Application Load Balancer

# 5. Optional: Creating and Publishing the App from the Command Line

If you are not on Windows, or do not want to use Visual Studio, you can still create an ASP.NET MVC Core web application and deploy it as a Fargate task in a container. The below steps are not at the same level of detail as the above walk-through, so you’ll need to be comfortable with the AWS Management console and the CLI in order to complete them.

1. **Create ECR Repository and Application Load Balancer**
2. In the ECS Management Console, create a new ECR repository called *aspnetmvcfargate*.
3. In the EC2 Management Console, create a new public-facing Application Load Balancer, with a listener on port 80 (the default), and a new target group with target type “*ip*” (not “instance”) using HTTP over port 80. Do not register any targets.
4. Ensure your ALB will be in at least two AZs – for simplicity sake, choose AZs “*a*” and “*b*” for this walk-through.
5. **Create and Configure an ASP.NET Core MVC Application**
6. Create a new directory on your computer called *fargate-sample-app* and navigate to that directory from a command window or shell (eg, use “cd fargate-sample-app”)
7. Create the ASP.NET MVC Core application on your computer using the dotnet CLI command in the directory you just created:  
   dotnet new mvc  
   This will generate the web application project inside the fargate-sample-app directory.
8. Navigate into the Views/Home directory using cd Views/Home
9. Use the editor of your choice (Vi, emacs, nano, etc) to edit the *Index.cshtml file*, and add the following line, as detailed in *Task I*, Steps *g* and *h* of this walk-through.  
   <h2>Hostname = @System.Environment.MachineName</h2>
10. Save your changes. You can test out the web app by typing dotnet run (and press enter). After the project builds and is available via the ASP.NET Core’s built-in web server (Kestrel), you should see a message that includes the local URL at which to view it. It will usually be something like this:  
    Now listening on: http://localhost:5000
11. Copy/paste or type the URL into a browser, and wait for the website to appear. When you’re ready to continue, press *Ctrl+C* to return to the command line.
12. **Add Docker Support to your ASP.NET Core MVC Application**
13. Create a new text file called *Dockerfile* in the same directory (*fargate-sample-app* directory). Do not give it any file extension.
14. Edit your Dockerfile so that it matches the following lines (feel free to copy/paste):

***Dockerfile contents:***

FROM microsoft/aspnetcore:2.0 AS base

WORKDIR /app

EXPOSE 80

FROM microsoft/aspnetcore-build:2.0 AS build

WORKDIR /src

COPY \*.csproj ./

RUN dotnet restore

COPY . .

WORKDIR /src

RUN dotnet build -c Release -o /app

FROM build AS publish

RUN dotnet publish -c Release -o /app

FROM base AS final

WORKDIR /app

COPY --from=publish /app .

ENTRYPOINT ["dotnet", "fargate-sample-app.dll"]

1. Next, back in the ECS Management Console, return to the ECR repository you created earlier, and click the “*View Push Commands*” button to view the commands we’ll use next.
2. **Build the Container and Push to the ECR Repository**
3. Retrieve the docker login command used to authenticate to your ECR repo with this command:  
   aws ecr get-login --no-include-email --region us-east-1  
   This will return a long login command starting with docker login -u AWS –p, followed by a very long encrypted password and then a URL.  
   Copy and paste the entire command (including the URL) onto the command prompt or bash prompt and press *Enter* to issue the command.
4. Build your container image using the docker build command, passing the current directory as the context and a tag by using the command from the View Push Commands dialog:  
   docker build –t aspnetmvcfargate **.**  
   (Note: include the period!)
5. Next, tag your container as the latest using the next command in the View Push Commands dialog (*if you copy from this below command, replace <aws-account-id> with your AWS account id, without dashes*):  
   docker tag aspnetmvcfargate:latest <aws-account-id>.dkr.ecr.us-east-1.amazonaws.com/aspnetmvcfargate:latest
6. Next, push the container up to ECR using the command from the View Push Commands dialog:  
   docker push <aws-account-id>.dkr.ecr.us-east-1.amazonaws.com/aspnetmvcfargate:latest

*Note: It may take a couple minutes for the container image to be pushed to ECR. You will see the upload progress in the command or bash window.*

1. **Create an ECS Task**
2. Next, we need to create an ECS task definition file. Create a new file called *aspmvc-task-def.json*.
3. Edit your task definition file so that it matches the below lines (feel free to copy/paste), replacing *<aws-account-id>* with your AWS account ID (no dashes).

***aspmvc-task-def.json contents:***

{

"executionRoleArn": "arn:aws:iam::<aws-account-id>:role/ecsTaskExecutionRole",

"containerDefinitions": [

{

"name": "AspNetMvcDemo",

"image": "<aws-account-id>.dkr.ecr.us-east-1.amazonaws.com/aspnetmvcfargate",

"portMappings": [

{

"hostPort": 80,

"protocol": "tcp",

"containerPort": 80

}

],

"essential": true,

"environment": [

{

"name": "ASPNETCORE\_ENVIRONMENT",

"value": "Production"

}

]

}

],

"memory": "512",

"cpu": "256",

"taskRoleArn": "arn:aws:iam::<aws-account-id>:role/ecsTaskExecutionRole",

"family": "AspNetMvcDemo",

"requiresCompatibilities": [

"FARGATE"

],

"networkMode": "awsvpc",

"volumes": []

}

1. In the IAM Management Console, check to see if you have an existing role called *ecsTaskExecutionRole* – if you do not, follow the steps in the documentation to create the role and assign the correct policy to grant ECS permissions to pull containers from the registry:

<https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_execution_IAM_role.html>

1. Register your newly-created task with ECS using the following command:  
   aws ecs register-task-definition --cli-input-json file://aspmvc-task-def.json --region us-east-1  
   The command will display the JSON for the created task definition after it succeeds.
2. **Create an ECS Cluster and Service**
3. In the ECS Management Console, go to Clusters, and create a new cluster, selecting the “*Networking Only, Powered by AWS Fargate*” template.
4. Assign a name, like *aspnetmvc-cluster*, and click the “Create” button.
5. Navigate to your cluster from the Clusters list, find the “Services” tab, and click the “Create” button to create a new service.
6. Choose the Fargate launch type, select the task you just created from the Task Definition drop-down (*AspNetMvcDemo:1*), and assign a name. See Figure 12 below.

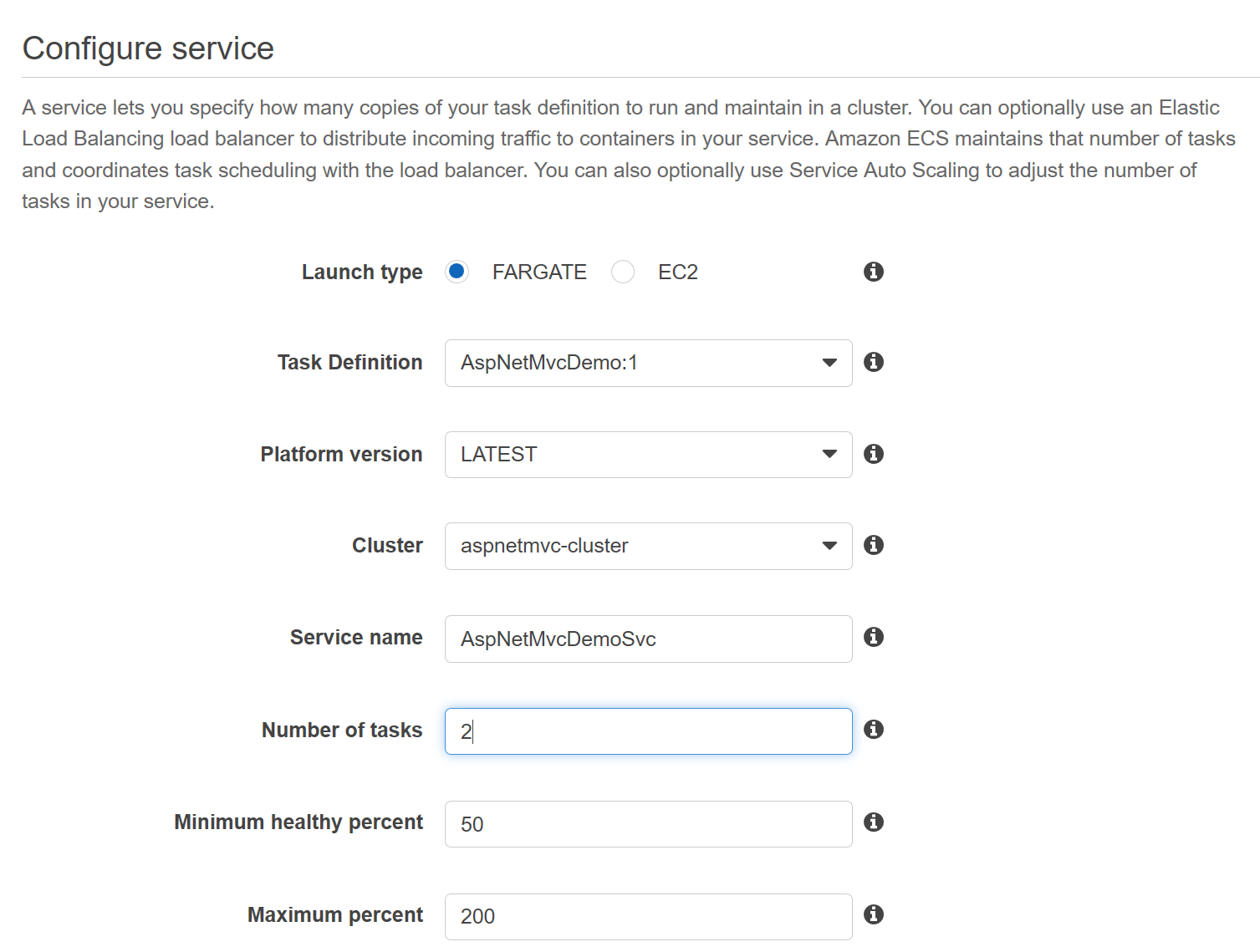


Figure 12 – Configure Service view in ECS Management Console

1. Set the number of tasks as *2*, and click the “Next step” button.
2. On the *Configure Network* view, select a VPC and two subnets. Ensure you select two subnets that are in the same AZs as the ALB you created in step 1b (in AZs “*a*” and “*b*”).
3. Next to “*Auto-assign public IP*”, select *Enable* to assign public IPs to your tasks.

*Note: In a real-world scenario, we wouldn’t need to assign public IPs to each task (container) individually. Instead, we would access the services via the ALB. However, for demo purposes, it’s nice to be able to show that you can reach each task (container) directly by its public IP as well as via the ALB’s public FQDN.*

1. Under *Load Balancing*, choose *Application Load Balancer*, then in the drop-down list, select the ALB you created earlier, and add your container to the load balancer by clicking the “Add to load balancer” button.
2. Select 80:*HTTP* as the listener port in the drop-down, then for target group, choose the one you created when creating the ALB. Ensure the target group *Path Pattern* and *Health Check Path* are both “/” (no quotes).
3. Uncheck “*Enable Service Discovery Integration*” (we won’t use that for this walk-through, and using it has not been tested for this sample application), then click the *Next Step* button.
4. Do not configure service scaling – just click the *Next Step* button.
5. Review your selections on the final view, then click the *Create Service* button.
6. View your service, and click on the *Tasks* tab (of your service) to view the status of the two tasks. When a task shows as RUNNING, you can click the task to find details, including its public IP address.

You can now use the public IP addresses of the tasks in your browser to view the website directly from that task (container).

Next, return to the ALB you created in the EC2 Management Console, find its public DNS, and copy and paste that into a browser. The website should again display. Click the refresh button in your browser a few times. You should see the hostname that is displayed flip back and forth as your requests are routed to the two containers.

**You have completed this optional task!**