:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#ecs-example] = Example: Create an {aws} Fargate service using the {aws} CDK :info\_titleabbrev: Example: Create a Fargate service

== [abstract]

## Learn how to create an {aws} Fargate service.

// Content start

In this example, we show you how to create an {aws} Fargate service running on an Amazon Elastic Container Service (Amazon ECS) cluster that’s fronted by an internet-facing Application Load Balancer from an image on Amazon ECR.

Amazon ECS is a highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker containers on a cluster. You can host your cluster on serverless infrastructure that’s managed by Amazon ECS by launching your services or tasks using the Fargate launch type. For more control, you can host your tasks on a cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances that you manage by using the Amazon EC2 launch type.

In this example, we launch some services using the Fargate launch type. If you’ve used the {aws} Management Console to create a Fargate service, you know that there are many steps to follow to accomplish that task. {aws} has several tutorials and documentation topics that walk you through creating a Fargate service, including:

* link:https://aws.amazon.com/getting-started/tutorials/deploy-docker-containers[How to Deploy Docker Containers - {aws}]
* link:https://docs.aws.amazon.com/AmazonECS/latest/developerguide/get-set-up-for-amazon-ecs.html[Setting Up with Amazon ECS]
* link:https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS\_GetStarted.html[Getting Started with Amazon ECS Using Fargate]

This example creates a similar Fargate service using the {aws} CDK.

The Amazon ECS construct used in this example helps you use {aws} services by providing the following benefits:

* Automatically configures a load balancer.
* Automatically opens a security group for load balancers. This enables load balancers to communicate with instances without having to explicitly create a security group.
* Automatically orders dependency between the service and the load balancer attaching to a target group, where the {aws} CDK enforces the correct order of creating the listener before an instance is created.
* Automatically configures user data on automatically scaling groups. This creates the correct configuration to associate a cluster to AMIs.
* Validates parameter combinations early. This exposes {aws} CloudFormation issues earlier, thus saving deployment time. For example, depending on the task, it’s easy to improperly configure the memory settings. Previously, we would not encounter an error until we deployed our app. But now the {aws} CDK can detect a misconfiguration and emit an error when we synthesize our app.
* Automatically adds permissions for Amazon Elastic Container Registry (Amazon ECR) if we use an image from Amazon ECR.
* Automatically scales. The {aws} CDK supplies a method so we can auto scale instances when we use an Amazon EC2 cluster. This happens automatically when we use an instance in a Fargate cluster.
* In addition, the {aws} CDK prevents an instance from being deleted when automatic scaling tries to stop an instance, but either a task is running or is scheduled on that instance.
* Previously, we had to create a Lambda function to have this functionality.
* Provides asset support, so that we can deploy a source from our machine to Amazon ECS in one step. Previously, to use an application source, we had to perform several manual steps, such as uploading to Amazon ECR and creating a Docker image.

= [IMPORTANT]

The ApplicationLoadBalancedFargateService constructs we’ll be using includes numerous {aws} components, some of which have non-trivial costs if left provisioned in our {aws} account, even if we don’t use them. Be sure to clean up (cdk destroy) if you follow along with this example.

====

[#ecs-example-initialize] == Create a CDK project

We start by creating a CDK project. This is a directory that stores our {aws} CDK code, including our CDK app.

==== [role=“tablist”] TypeScript:: + [source,none,subs=“verbatim,attributes”] — mkdir MyEcsConstruct cd MyEcsConstruct cdk init –language typescript —

JavaScript:: + [source,none,subs=“verbatim,attributes”] — mkdir MyEcsConstruct cd MyEcsConstruct cdk init –language javascript —

Python:: + [source,none,subs=“verbatim,attributes”] — mkdir MyEcsConstruct cd MyEcsConstruct cdk init –language python source .venv/bin/activate # On Windows, run ‘.’ instead pip install -r requirements.txt —

Java:: + [source,java,subs=“verbatim,attributes”] — mkdir MyEcsConstruct cd MyEcsConstruct cdk init –language java — + We may now import the Maven project into our IDE.

C#:: + [source,none,subs=“verbatim,attributes”] — mkdir MyEcsConstruct cd MyEcsConstruct cdk init –language csharp — + We may now open src/MyEcsConstruct.sln in Visual Studio. ====

Next, we run the app and confirm that it creates an empty stack.

== [source,none,subs=“verbatim,attributes”]

## cdk synth

[#ecs-example-create-fargate-service] == Create a Fargate service

There are two different ways that we can run our container tasks with Amazon ECS:

* Use the Fargate launch type, where Amazon ECS manages the physical machines that oour containers are running on for us.
* Use the EC2 launch type, where we do the managing, such as specifying automatic scaling.

For this example, we’ll create a Fargate service running on an Amazon ECS cluster, fronted by an internet-facing Application Load Balancer.

We add the following {aws} Construct Library module imports to our *stack file*:

==== [role=“tablist”] TypeScript:: File: lib/my\_ecs\_construct-stack.ts + [source,javascript,subs=“verbatim,attributes”] — import \* as ec2 from “aws-cdk-lib/aws-ec2”; import \* as ecs from “aws-cdk-lib/aws-ecs”; import \* as ecs\_patterns from “aws-cdk-lib/aws-ecs-patterns”; —

JavaScript:: File: lib/my\_ecs\_construct-stack.js + [source,javascript,subs=“verbatim,attributes”] — const ec2 = require(“aws-cdk-lib/aws-ec2”); const ecs = require(“aws-cdk-lib/aws-ecs”); const ecs\_patterns = require(“aws-cdk-lib/aws-ecs-patterns”); —

Python:: File: my\_ecs\_construct/my\_ecs\_construct\_stack.py + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import (aws\_ec2 as ec2, aws\_ecs as ecs, aws\_ecs\_patterns as ecs\_patterns) —

Java:: File: src/main/java/com/myorg/MyEcsConstructStack.java + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.services.ec2.\_; import software.amazon.awscdk.services.ecs.\_; import software.amazon.awscdk.services.ecs.patterns.\*; —

C#:: File: src/MyEcsConstruct/MyEcsConstructStack.cs + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK.{aws}.EC2; using Amazon.CDK.{aws}.ECS; using Amazon.CDK.{aws}.ECS.Patterns; — ====

Within our stack, we add the following code:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const vpc = new ec2.Vpc(this, “MyVpc”, { maxAzs: 3 // Default is all AZs in region });

…. const cluster = new ecs.Cluster(this, “MyCluster”, { vpc: vpc });

// Create a load-balanced Fargate service and make it public new ecs\_patterns.ApplicationLoadBalancedFargateService(this, “MyFargateService”, { cluster: cluster, // Required cpu: 512, // Default is 256 desiredCount: 6, // Default is 1 taskImageOptions: { image: ecs.ContainerImage.fromRegistry(“amazon/amazon-ecs-sample”) }, memoryLimitMiB: 2048, // Default is 512 publicLoadBalancer: true // Default is true }); —- ….

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const vpc = new ec2.Vpc(this, “MyVpc”, { maxAzs: 3 // Default is all AZs in region });

…. const cluster = new ecs.Cluster(this, “MyCluster”, { vpc: vpc });

// Create a load-balanced Fargate service and make it public new ecs\_patterns.ApplicationLoadBalancedFargateService(this, “MyFargateService”, { cluster: cluster, // Required cpu: 512, // Default is 256 desiredCount: 6, // Default is 1 taskImageOptions: { image: ecs.ContainerImage.fromRegistry(“amazon/amazon-ecs-sample”) }, memoryLimitMiB: 2048, // Default is 512 publicLoadBalancer: true // Default is true }); —- ….

Python:: + [source,python,subs=“verbatim,attributes”] — vpc = ec2.Vpc(self, “MyVpc”, max\_azs=3) # default is all AZs in region

…. cluster = ecs.Cluster(self, “MyCluster”, vpc=vpc)

ecs\_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",  
 cluster=cluster, # Required  
 cpu=512, # Default is 256  
 desired\_count=6, # Default is 1  
 task\_image\_options=ecs\_patterns.ApplicationLoadBalancedTaskImageOptions(  
 image=ecs.ContainerImage.from\_registry("amazon/amazon-ecs-sample")),  
 memory\_limit\_mib=2048, # Default is 512  
 public\_load\_balancer=True) # Default is True ----

….

Java:: + [source,java,subs=“verbatim,attributes”] — Vpc vpc = Vpc.Builder.create(this, “MyVpc”) .maxAzs(3) // Default is all AZs in region .build();

…. Cluster cluster = Cluster.Builder.create(this, “MyCluster”) .vpc(vpc).build();

// Create a load-balanced Fargate service and make it public  
ApplicationLoadBalancedFargateService.Builder.create(this, "MyFargateService")  
 .cluster(cluster) // Required  
 .cpu(512) // Default is 256  
 .desiredCount(6) // Default is 1  
 .taskImageOptions(  
 ApplicationLoadBalancedTaskImageOptions.builder()  
 .image(ContainerImage.fromRegistry("amazon/amazon-ecs-sample"))  
 .build())  
 .memoryLimitMiB(2048) // Default is 512  
 .publicLoadBalancer(true) // Default is true  
 .build(); ----

….

C#:: + [source,csharp,subs=“verbatim,attributes”] — var vpc = new Vpc(this, “MyVpc”, new VpcProps { MaxAzs = 3 // Default is all AZs in region });

…. var cluster = new Cluster(this, “MyCluster”, new ClusterProps { Vpc = vpc });

// Create a load-balanced Fargate service and make it public  
 new ApplicationLoadBalancedFargateService(this, "MyFargateService",  
 new ApplicationLoadBalancedFargateServiceProps  
 {  
 Cluster = cluster, // Required  
 DesiredCount = 6, // Default is 1  
 TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions  
 {  
 Image = ContainerImage.FromRegistry("amazon/amazon-ecs-sample")  
 },  
 MemoryLimitMiB = 2048, // Default is 256  
 PublicLoadBalancer = true // Default is true  
 }  
 ); ---- ====

….

Next, we validate our code by running the following to synthesize our stack:

== [source,none,subs=“verbatim,attributes”]

## cdk synth

The stack is hundreds of lines, so we won’t show it here. The stack should contain one default instance, a private subnet and a public subnet for the three Availability Zones, and a security group.

To deploy the stack, we run the following:

== [source,none,subs=“verbatim,attributes”]

## cdk deploy

{aws} CloudFormation displays information about the dozens of steps that it takes as it deploys our app.

Once deployment completes, we have successfully created a Fargate powered Amazon ECS service to run a Docker image.

[#ecs-example-destroy] == Clean up

As a general maintenance best practice, and to minimize unnecessary costs, we delete our stack when complete:

== [source,none,subs=“verbatim,attributes”]

## cdk destroy