Course Overview

Course Overview

Hello there. We need to talk about containers. The Docker revolution, along with container technologies in general, promise to add deep automation and customizability to both application development cycles and deployments. Microservices, in which each individual container within a cluster is built to perform only one specific task, is, as they say, a thing right now. But the complexity of managing large clusters of containers across a distributed hardware platform can be daunting. Running your clusters on AWS's managed Elastic Beanstalk service can abstract away a number of administration levels, as AWS itself will invisibly take care of hardware provisioning and administration, networking, and much of the software stack for you. If you're already familiar with the basics of the working with Docker containers, my Using Docker with AWS Elastic Beanstalk course will be a great way to introduce you to Docker cluster management, the various AWS service resources the Beanstalk platform will use as it takes your application through its lifecycle, and the process of quickly getting Beanstalk itself up and running with your own application projects. I hope you'll join me.

Introduction to Elastic Beanstalk and Docker

Elastic Beanstalk: What's In It for You?

Welcome. Great to have you here for this course on getting your Docker containers running in the AWS cloud. Now, don't think I can't hear your wisecracks. Come on Clinton, you can't fool us, you've already got a Pluralsight course on getting our Docker containers running in the AWS cloud. Well, that is true. But that course covered the hard way to get it done using the EC2 container service. This is where I tell you about how you can use Amazon's Elastic Beanstalk to make most of the infrastructure complexity magically disappear. Docker lets you easily share and deploy absolutely predictable operating system images. Whether you want remote members of your developing team to work with exactly the same software environments that you built, or you need to reliably deploy an image to production over and over again, Docker can help. But while running Docker Engine on a local machine, and even exposing some public service to the web, isn't necessarily going to be all that difficult, tapping into its real power by launching automated clusters of containers is a different story, and particularly if you need to launch those clusters into a public cloud like AWS. There are, of course, whole categories of deployment management tools, like Ansible, Chef, and AWS's own ECS, that could be used to help out, but the complexity they hide is often matched by the new learning curve needed to get the thing off the ground. Enter Elastic Beanstalk. Beanstalk is a service that accepts your application code as input, and then invisibly embeds the code within a complete network and compute infrastructure. You select a platform like .NET or Python or Docker, and the service does the rest. That means that the EC2 and database instances, security groups, load balancers, and auto scaling your application will need, will be automatically added, integrated, and continuously maintained for you. That's the value proposition for the technology. Now here's what we'll actually cover. In this introductory module, we'll take a fairly close look at Docker and Elastic Beanstalk themselves, and in particular, at how Docker handles clusters and microservices, how Beanstalk abstracts the hardware that runs it, and how it fits in with Amazon's other Docker service, ECS. Next, we'll explore the Beanstalk tools you'll use for managing Docker workloads, including the EB Command Line Interface, and both versions of the Dockerrun.aws.json scripting format. We'll learn about how Beanstalk integrates Docker deployments with all the same AWS tools that will be used by less automated approaches, including ECS itself. And finally, we'll put all those pieces together into a full-sized Elastic Beanstalk-based Docker project. Let's go. Now don't forget my using Docker with AWS Elastic Beanstalk web page, where all the commands and lab setup details from the course have been uploaded, anxiously waiting for your visit.

Dockerfile: Scripting Docker Containers

I'm not going to show you how to install Docker Engine on your local PC, and then pull an image from a repo, like Docker Hub. Those are things I expect you've done on your own. If you aren't yet comfortable with all that, feel free to watch the first handful of clips of my Using Docker on AWS course. Similarly, I'm going to assume that you're already familiar with the way containers and images are used within the Docker lifecycle process. Not so sure? Well there's always my Managing Docker Images course. But what I will do, is show you a simple Dockerfile to quickly demonstrate how scriptable Docker can be. This example will use the latest official Ubuntu image as its base, update the local apt repository index, install the Apache web server package, take an existing file in the local directory on my host machine called index.html, and copy it to the web server root directory, make sure that Apache is active, and open port 80 for browser traffic. I'll build a new image out of that Dockerfile using docker build, and pause the recording to spare you the 45 seconds or so that the operation will take. Now, running docker images should show us our new myserver image. This image can be archived, shared, and run over and over and over again, each time providing an exactly identical environment. In our case, including a working web server offering the same index.html file. I'll launch a container using docker run, confirm it's running with docker ps, get its IP address using docker network inspect, and use curl to access its web page. All systems are go, and all of that automatic environment goodness from a half dozen lines of a script, that's how things work when we only need to run our containers one at a time, but running containers one at a time is actually rather unusual. The real power of Docker is in how they can be orchestrated in clusters, which we'll explore next.

Docker Swarm and Microservice Deployments

It seems to me that the most foundational way to manage Docker clusters, since Docker Engine version 1.12 at least, has got to be Docker Swarm. I think it's worth taking a quick look at how Swarm, which comes with Docker Engine by default, works. In this demo, I'm going to initialize my local Docker host as a Swarm manager, join a remote Docker host to my new Swarm, give you a bit of a tour around the new Swarm neighborhood, launch a single container called a service into the Swarm, and then, using one short command, scale the Swarm to five containers distributed across both of my hosts. My goal, to go from 0 to a fully functioning Swarm in less than a minute, without counting the time I spent explaining what I'm doing. I've got virtual machines running within VirtualBox on two separate physical servers, known as nodes, within the same network. The one I'm logged into here is going to be my Swarm manager. I'll make that happen by running docker swarm init. Most of the text that Swarm spits back out at me is actually the command I can run from a second, or third, or fourth node, that I want to join to Swarm. I'll copy the command, move over to my second node, and run it. Notice how it includes a token, and importantly, the manager node's IP address and the 2377 port that it's listening on. From here on in, the manager will be able to launch and administrate containers locally, and on this second Swarm node. Let's move back to the manager because that's where all the Swarm commands will need to be run. Docker node ls will list all the nodes currently on the Swarm identifying the leader. Docker info prints the Swarm's current status and configuration settings. Now here's where things get just a bit magical. Docker manages its Swarm containers, not as individual containers, but as objects whose purpose it is to provide a service. The docker service command will do everything it takes to make the service you want available. In this case, I'm creating a new web server service that will listen on port 80, have the name webserver, and include the NGINX web server package. Take my word for it, at this point, I haven't got NGINX installed anywhere on this machine. Not even a Docker NGINX image. Running docker service ls, will list all the active services that I've got at the moment. Right now, it's just webserver, and there's a single replica of the service still loading. Docker service ps webserver shows that single replica, or container, along with the information that the container is running on our local node. Great. But so what? Okay, suppose I want to scale up the number of containers to meet demand. Perhaps I want to script a load balancer to order extra replicas whenever the existing instances seem to be having trouble with the load. No problem. Here's how it's done. Docker service scale webserver=5. I'll give it a couple of seconds, and then run service ps once again. We've got five containers spread among our nodes, but are they all actually doing anything? Let's find out. They're supposed to be web servers, right? So I'll use curl to access the index pages. Before I can do that, however, I'll need to know their IP addresses. This is Docker, so I'll all the local networks using network ls. The Docker gw bridge network is the one we're after. I'll run network inspect against that bridge, and look for ipv4 address entries. I see multiple addresses in the 172.18 network, and I'll bet those are my web servers. If I run curl against one of those, aha, the nginx default home page. Of course, our simple example won't do us a lot of good. We don't even have our own content in the web root. But just imagine how easy it would be to launch multiple services using Swarm, web servers, load balancers, databases, storage volumes, and since everything works through simple command line code, it's all just begging to be scripted. The Swarm model allows for the use of many highly decoupled microservices as parts of a complex application. By decoupled, I mean a service providing, say a back-end database, could be managed independently of all the other application services. This means that your database development team can be free to go about its work without having to wait for other teams to sign off on any changes. It also means that database service containers can be run on nodes that are optimized specifically for their needs, as other services can use nodes with the profiles that work best for them. But managing multiple microservices across multiple nodes can quickly become hopelessly complicated. Besides making sure that each node's available resources are effectively and efficiently utilized, you'll also need to keep track of all your endpoints and back-end elements, like databases, so the services can properly communicate with each other. Even moderately sophisticated deployments will soon become impossible to handle manually. To bring all this under control, you're going to need to use some kind of configuration and deployment package, like Jenkins or Ansible. One native Docker Swarm administration tool is Docker Machine, but diving into that would go a bit beyond the scope of this course. However, this does bring us nicely back to Elastic Beanstalk. Exactly the place we've been aiming for all along.

Elastic Beanstalk and Docker: A Solution

Amazon's Elastic Beanstalk has two huge advantages. As I mentioned earlier, it abstracts away layers of complex provisioning and maintenance, making it much easier to quickly incorporate your application into a reliable infrastructure, but it also places your application right at the center of the Amazon Cloud ecosystem, just where all the cool kids are playing these days. At its simplest, Beanstalk lets you focus on your application code without having to worry about the hardware and networking environment that'll run it. It can require only a couple of quick choices, and you're off to the races. Now, getting some kind of Docker microservices application going will require a bit of effort, in particular, it'll require that we have at least a general understanding of how AWS, through Beanstalk's wheels and cogs, handles your containers, and that's exactly what the next module is going to focus on. First though, let's review what we saw in this module. We saw how Docker is a very scriptable platform, building images from Docker files when using Docker Engine on a local machine. We saw how you can provision and manage container clusters using Docker Swarm, and then manage your cluster from the manager node. Docker service is a tool you use to launch the containers of a service, which you can later modify, including scaling the number of containers running in a particular service. We learned how microservice architectures can make development cycles for complex applications far more efficient and robust. And finally, how Elastic Beanstalk can get your clusters up and running within the AWS environment quickly, and largely, painlessly.

The Elastic Beanstalk Toolset

The Elastic Beanstalk Command Line Interface

Still with me? Great. In this module, we're going to learn our way around planet Elastic Beanstalk, with a particular focus on the kind of special care and feeding that Docker containers can get there. We'll start by installing and using Amazon's Elastic Beanstalk Command Line Interface, then we'll work through some of the peculiar Beanstalk terminology, and also unravel some of the confusion that can sometimes hover around the three different ways to play with Docker on Beanstalk, single-container, multi-container, and preconfigured. We'll also look at using Dockerrun.aws.json files, and the differences between versions 1 and 2. Finally, we'll launch an actual application or two. We're going to be busy for the next while, maybe you'd better turn off your phone. See here we are at last, the Elastic Beanstalk Service dashboard. From here, we're just a few clicks away from taking robust environments from idea to production in no time flat. Just don't get too comfortable, as I plan to avoid it as much as humanly possible in favor of AWS's Beanstalk Command Line Interface. Now don't get me wrong, I've got nothing against working from a browser, but believe it or not, I think that Beanstalk, even more than other AWS services, will make more sense to you and get you working productively more quickly from a command line. And besides, making things happen from the command line interface is way more fun. There is one thing you'll still have to do from the browser, generate your first pair of access keys. It's true that you can always run aws iam create-access-key --user-name from the AWS CLI, but that will only work once you've got the CLI going, and that requires keys. It's a bit of a chicken and egg thing. So I'll click on my account name of the top of the screen, and then on My Security Credentials. Once there, I'll expand the Access Keys item and click Create New Access Key. Now I must tell you that I'm doing two very bad things here. Number one, I'm using my root account for day- to-day access to services, something that for security reasons you should really avoid, using instead an account user you can create and then assign appropriate permissions in IAM, and I'm exposing my keys in this video, which normally would be insane. Don't worry though. I'll have deleted these keys long before this course if published. I'm only doing it so you can see how it's done, but you should never do this yourself as part of a production deployment. Make sure you safely copy the keys right away, especially the Secret Access Key, which you'll never be shown again. Now we're ready to head over to the command line. This is a VirtualBox virtual machine running Ubuntu Linux with Docker Engine and Python 3 already installed. I could probably get a lot done using the AWS general purpose CLI, but I'm going to use their special Elastic Beanstalk interface instead, which I'm going to install using the pip method. First, I'll use curl to download a script to install pip, the Python package manager, and then use Python to run the script. I'll check the pip version to confirm that it's installed and active, and then I'm finally ready to use pip to install the awsebcli. Once that's nicely settled, I'll type eb, which is the way I tell my shell that I'd like to use the Beanstalk Interface. Eb alone prints the help page. One thing I'll do now is create a new directory called, say, myapp, change that directory, and then initialize the directory as a base for my application. Here I'll select a region for my application. I'll take my life in my hands and go with us-east-1, despite that region's 2 serious service outages over the past couple of years. Since I'm setting things up for the first time on this machine, I'll then enter my key ID and secret access key. Good thing I saved them. Now I'm asked to give my application a name. It can be the same as the directory. I'll select a platform, Docker is fine for me, and then a Docker version. Having SSH access to my container might later prove useful, so I'll say yes. If there's already an SSH key pair associated with this account, we'll be asked if we want to use it. Otherwise, we can create a new one. I'll show you how to use ebssh a bit later in this module. And we're done. Eb list will print all the environments we've currently got live in the current region. None so far. Eb create will launch a new environment, but we're not ready for that just yet. Next up, we'll learn about the way Beanstalk organizes things.

Working with Docker in a Beanstalk World

Let's take a bit of a step backwards to give ourselves a bigger view of the playing field. When you initialize a directory on your local PC, the way I just did to create a new application, and then use create to push a new environment to AWS, Beanstalk will build an application framework and an environment. The framework is really just a space for storing your application files. The environment, by contrast, is the infrastructure Beanstalk assembles to support your application. That means things like at least one EC2 instance, a load balancer, auto scaling rules, CloudWatch alarms, and security groups. I think it's fair to say that EB environments are roughly translatable to the services we saw earlier in Docker Swarm. That is, both contain the configuration information and ability to spawn and manage as many containers as necessary to do the work at hand. The environment you'll get depends on how you answer the questions that accompany the eb init command when you ran it in the new directory. Eb platform show will display some of the platform and environment values that will be used for any subsequent create operations. You'll remember how the platform choices we were offered included both Docker and multi-container Docker. The difference between the two is practical. Single containers don't require load balancers and auto scaling to manage traffic between resources the way multiple containers do. What these single containers do need, however, is some way to communicate with the outside world. That, in Beanstalk, is provided by an NGINX reverse proxy. The most significant difference between the way you prepare applications for those two platform options, is in which of the two versions of the Dockerrun.aws.json file you'll create. Version 1 is used for single container environments, and as you can probably figure out for yourselves, you use version 2 for multiple containers, an architecture that might be attractive if you're trying to pull together an application using a fleet of independent microservices. I'll talk more about those two formats in the next clip. Now, besides those two choices, there's also something called a preconfigured option. AWS makes Amazon machine images available that are preloaded with fully configured versions of popular deployment profiles on top of Amazon Linux. Practically, that means rather than having to start from scratch with your Dockerfile or Dockerrun files, you're able to use a shortcut by selecting a baseline environment in which to build further. As you can see here, there are preconfigured AMIs available for Docker with Glassfish, Go, or Python already installed. If you want to take advantage of, say, the Python option, but from the CLI rather than through the browser, and that is your first choice, right, then you would simply point the FROM line in your Dockerfile to this aws-eb-python image. You can get the image information for all pre-configuration options from AWS's documentation, docs.aws.amazon.com/elasticbeanstalk/latest/dg/concepts.platforms in this case. I'll quickly illustrate how all this works from the myapp directory I just created and initialized. Let me tell you a bit of a sad story first. My early attempts to launch Docker containers in Beanstalk were not particularly successful. Oh, most of the 10-minute process seemed to go alright each time, but at the end, the application itself would invariably fail with error log messages claiming that there was no Dockerfile present. No Dockerfile? But of course I'd included a Dockerfile, often nothing else but a Dockerfile. After some time and research, it finally hit me that it was Docker Engine's fault. Years ago when I first started using Docker, I had read that the Dockerfile file name should be spelled with an uppercase D, but at some point Docker Engine stopped caring, and I got into the lazy habit of spelling it with all lowercase. Well it turns out that Beanstalk isn't quite so generous as Docker Engine, and simply didn't see any Dockerfile, no matter how much I was sure it was there. The moral of the story? Spelling counts. Use uppercase. So with that in mind, I'll open a new file named Dockerfile with nano, my favorite text editor, and paste in this example which actually comes to us courtesy of some AWS documentation. As we've already seen, the FROM line specifies the preconfigured Python AMI. The EXPOSE line opens port 8080 between the Python running on the container this Dockerfile will build and the reverse proxy on the host. This will allow communication with the internet. You could open other ports, but reverse proxy will only use the first on the list. Finally, the RUN line installs the postgresql database engine. What that final command does is force delete the contents of the var/lib/apt directory. On Debian systems, this directory can fill up pretty quickly with package installation-related data. Apparently, individual layers of Docker images can bloat quite quickly, unless you keep a lid on the stuff that's kept here. So consider this a best practice. Now, there's nothing left but to pull the trigger with eb create, and any name I'd like to give the environment. The whole process can easily take 10 minutes before it's done, but you will see event logs displayed in the terminal, and in the browser console. Just make sure you're in the right region. I'm also going to show you some of the stuff that's happening behind the scenes. If you move over to the EC2 dashboard, and then the instances page, we can see a new instance loading, along with details of the security group it's using, the Beanstalk AMI, and the instance's public IP address, although that won't get you anywhere in this particular case.

Working with Dockerrun.aws.json v.1

In this demo, we're going to create a Dockerrun.aws.json file for a single container web server deployment. To do that, I'll make and initialize a new directory, write a simple index.html file that will populate the web root directory of the container, write my Dockerrun file, create the environment, and then, just to show you exactly what's happening, SSH into the EC2 host machine. Let's go. Back on my local Linux virtual machine with Docker Engine and the Beanstalk CLI installed, I created a new directory called mysingle, short for my single container application. I guess I'm not very likely to win any literature prizes for creativity. I'll change into my new directory and initialize it, once again, choosing the single Docker container platform. Now I'll use my text editor to add a short inspirational message to an index.html file that will have to act the role of a web application. Bear in mind that I'm saving this file to the same directory where the Dockerrun file will go, which coincidentally, is my next order of business. This file is the centerpiece of the whole thing, so we'll go through it one section at a time. The first line within the curly braces will tell Beanstalk that this config file is written in Dockerrun version 1. The first section, Image, points to the Docker image that we want to use as a base for the container. In my case, I'm going to use the Apache web server whose official image is called httpd. Since the httpd image is an official image hosted on Docker Hub, it doesn't require any more of an identifying address. Beanstalk would need a bit more explanation for images hosted elsewhere. If you want to use a private image, you'll also need to include authentication information. This isn't an issue in our case, but is usually accomplished by saving your authentication information to an S3 bucket, and then exposing the bucket as part of an authentication section in the Dockerrun file. We'll see this happen in the final module in this course. Back in our Dockerrun file, since this is going to be a web server, we'll expose port 80 in the Ports section, and down at the bottom, tell Beanstalk where we'd like our logs. The Volumes section will require a bit more context before it makes sense. This is how we pass files and data along a two-step journey from our computers, to the host EC2 instance, and then to the container we're building. When you include a file among your application data, like the index.html I created for example, it is automatically copied to a directory on the EC2 host instance called var/app/current. That's called the HostDirectory. I'll show it to you a bit later. The ContainerDirectory value tells Beanstalk to mount any files in current to, in our case, usr/local/apache2/htdocs. Why there? Because that happens to be the default web root directory on Alpine Linux instances, or in other words, that's where Apache will look for any files it can use to populate its website. I'll save the file and exit nano. Now it's a good idea to do a test run of our environment, especially since it can take so much time for Beanstalk to put the whole thing together. So I would use eb local run to run it locally. If there were any serious problems, they would probably show up in output we'd be shown. Unfortunately, the most recent Docker release of last week seems to have broken this feature in the eb cli for now, so I'll have to take my chances with the real thing. I'll type eb create, giving the environment a name. After the first short while, we're told that it's safe to Ctrl+C out of the shell, so at least we get our command line back. If you want to check on the build process, we can always hit eb status, although it may still be a bit early to get anything useful from that. We could also return to the AWS dashboard, where updates are displayed as they come in. This is the first place you should come to troubleshoot if anything should go wrong. Once the EC2 host instance is running, we can actually SSH in from back in the terminal using eb ssh and the environment name, without the need of regular login tools. This is possible and safe because you've already got the key pairs set up from before. I'll go to var/app/current to list the contents. Now this is interesting when you consider that there were only two files in our initialized directory, but they seem to have had a baby. Where did this Dockerfile come from? If we take a look inside, we'll see that Beanstalk had taken the images and port sections from the Dockerrun.aws.json file, and added them to the Dockerfile, converting them to the proper syntax in the process. I guess this makes it easier to integrate all this with the Docker system. At any rate, we'll return once more to my terminal and run eb status. This time, everything appears to be running properly. I'll copy the instance endpoint and use curl to pay it a visit. And just look, our web application. Last step, cleanup. When you're done admiring your work, you can shut it all down using eb terminate. This very important, as Beanstalk launched all kinds of services and resources as part of this environment. If you don't shut it down the right way, some could be left running, and costing you more money every hour.

Working with Dockerrun.aws.json v.2

That covers us nicely for Dockerrun version 1 files and single-container deployments. If you want to create an infrastructure with multiple containers working together, perhaps you're designing an application using microservices, then you'll need a version 2 file, and you've come to the right place. Here's a sample file that comes to us courtesy of Amazon's documentation. Right away, you can see that it's built quite differently than the version 1 we worked on earlier. Of course, the first line spells out the version explicitly, but the volumes section is a whole different design. For one thing, each volume has a name, which will be referenced later in the file in a separate mount section. At this point, we're only providing a sourcePath, which is the same as the host directory in version 1. That means it refers to files of the var/app/current directory on the EC2 host instance that were passed from within the application directories in our local PC. This first volume is called php-app and will be taken from any files that are present within the directory php-app in my application root directory on my PC. If I like, I can populate it with an index.php file. The second volume will contain nginx configuration files to make our web proxy work. The next section contains container definitions. The first of two containers this file will create happens to be called php-app, and will use the Docker php:fpm image. Fpm means that it's optimized for fast CGI. The container environment, we'll use the name PHP. The true value for essential means that if this container goes down, the whole application will stop. The container will be given 128 MB of RAM, and its internal mountPoints, as defined below, will be the web root directory of var/www/html. Finally, this true tells the system that the files in this volume will have read only permissions. The second volume, which will be used to populate the NGINX container, follows a similar pattern. The links section establishes communication between containers, in this case, allowing connections with the php-app container. The last section is mountPoints. Here volumes we've already defined above are referenced by name and then their container path, which works the same as container directory back in version 1, and permissions are set. The source for this third volume will be generated automatically by AWS, and will live in the var/log containers directory in the EC2 host. As with version 1, you'll need to include all the files and directories you want in your container within the application root directory. You'll also need to specify multi-container Docker rather than just plain old Docker when you initialize that directory. Let's do some review. You'll remember how we used Python pip to install the Elastic Beanstalk CLI to a virtual machine running Linux. We saw how a directory can be initialized with a particular profile to make it available for creating Beanstalk environments. We learned about three kinds of Beanstalk Docker platforms, preconfigured, where you can used a specially prepared EC2 AMI that comes with a specified environment like Python preinstalled, single-container, and multi-container. We dug into the syntax of Dockerrun.aws.json files, both version 1 for single-container deployments, and version 2 for multiple containers, and we learned about the differences between syntax used by the 2 versions.

Integrating Beanstalk and Docker with Your AWS Resources

ECS and Elastic Beanstalk Integration

Even though, as I said a while back, Elastic Beanstalk is an abstraction designed to hide a lot of the complexity of large cluster deployments, it's worthwhile spending at least a little time learning about some of the elements that make it work. In fact, some Elastic Beanstalk administration tools will appear incomprehensible if you don't have some basic understanding of what's going on under the hood. So in this module, I'm going to take you on a simple tour through half a dozen or so of the major AWS services being used within a Beanstalk environment. We'll make our way through many of the elements launched silently by Beanstalk in support of our application. We'll quickly look at our EC2 instance, the load balancer coordinating the flow of user traffic, and the Auto Scaling group that maintains the appropriate numbers of containers. We'll see IAM and the policies that define resource access, S3, where key data files can be stored, and CloudWatch, where events are tracked. But our first stop will be the EC2 container service, AWS's framework for defining and deploying clusters of Docker containers. I'm definitely not going to dive too deeply into ECS here. For one thing, its design presents just a level of complexity that Beanstalk is supposed to avoid, but also because I've already got a Pluralsight course that covers that space, Using Docker on AWS. Instead, I'll just offer you a quick structural overview. The containers you launch into ECS all run on one or more instances of a special EC2 server built on the Amazon ECS-optimized Amazon Linux AMI. That Amazon machine image, once launched as an instance, comes out of the box with an Amazon ECS agent Docker container already running. The agent is the point of contact with the ECS manager, and with other containers. An ECS configuration itself is made up of clusters, tasks, container definitions, and services. A cluster, the way ECS uses the term at least, is really just a space used to contain and organize all the other bits. Within your cluster, a task will allow you to define some of the working environment your containers will have, primarily through the container definition options that it exposes. The service is the orchestrator that launches, monitors, and controls all the containers as they go about their duties. It's the service that manages each of the running tasks. I'm not sure you've noticed this, but what I find a bit nasty about the container business as a whole, is that each platform, whether it's Docker itself or other providers like AWS, Ansible, and Kubernetes, will use terms like service and cluster in different ways, often just close enough to each other to be totally confusing. If I ruled the universe, things would work very differently, let me tell you. Well as long as I'm not yet ruler of the universe, we'll continue our tour in the next clip.

Elastic Beanstalk Integration with AWS Infrastructure

Let's see how things look over at the dashboard for Elastic Beanstalk itself. Currently, I have only one application running in this region. The reason it's tinted an ominous red is because it's currently in a failed state. That shouldn't bother us though. I'll click on the environment, and I'm taken to its dashboard. Notice the endpoint URL that's shown at the top. Recent events are displayed below. I'll scroll down just a bit to show you. The Configuration tab displays tiles that nicely group configuration items by topic. I'll click on the Instances gear icon, and we can see how simple it is to update any particular configuration setting. The next stop on our tour will be the EC2 instances dashboard. You can see that the only instance currently running right now is using the AWS Elastic Beanstalk AMI, and the Security group permits SSH traffic on port 22 from anywhere, and browser traffic through port 80 from a second security group. You can tell this is a security group ID because it begins with the letters sg. I think it's worth looking a bit deeper into this second group. If I click through to the security group itself, I can see that the other group whose inbound traffic we permitted is being used by an Elastic Beanstalk load balancer. In fact, the way Beanstalk designs multi-container environments, users will only reach their application by way of load balancers, which are capable of organizing what would otherwise quickly become chaotic. The load balancer page itself provides plenty of troubleshooting tools. Here's another clue to the failed state problem being experienced by our application. We'll look at one more tool while we're still within the EC2 pages, Auto Scaling Groups. Here too, the launch configuration that our group is using has an AWS Elastic Beanstalk ID showing us that it was created, and is managed, entirely by Beanstalk. This should also remind us that shutting down a Beanstalk environment, once you're done with it, should definitely be done through Beanstalk itself. Otherwise, there's a very strong change that you're going to leave something running for no reason. Because internal coordination between AWS resources is so critical to making a container infrastructure work, especially when you're building an application based on microservices, effectively assigning permissions to all your bits and pieces is really important. In the AWS world, most of that will be handled through IAM rules. If you'd try to put these pieces together manually, you'd have to create and assign each of the policies on your own. It can be done, but it can also get really complicated. To see how it's handled by a pro, let's take a look at how Beanstalk did it for our application. There are currently two active roles, both associated with Elastic Beanstalk. I'll click on the EC2 role, and we can see that three policies have been attached. Reading through the WebTier policy, we see that an EC2 instance with this role attached will be able to Get, List, and PutObjects in any S3 bucket whose name contains the elasticbeanstalk prefix. This is critical, because many related configuration and data files will be stored in S3. XRay is a new AWS service that helps you analyze and debug distributed applications. At this point, it's still in preview, but I guess the IAM policy has already been updated in anticipation. I can tell you that I certainly would never have thought to include it on my own. Finally, the policy gives EC2 instances the ability to create and store a CloudWatch log file, so you can, if necessary, consume them later. The MulticontainerDocker policy gives EC2 instances the ability to interact with ECS administration tools. The asterisk, next to Resource, tells us that this applies to any associated resource. And the WorkerTier policy, permits access to the various resourced needed to use a worker environment to decouple an application front end from long-running processes. I'm afraid that getting to the bottom of that set of concepts would take us quite a way beyond the scope of this course. The Access Advisor tab takes us to data that can tell us the last time each of those IAM permissions was actually granted. Knowing how, and how often resources are consumed within our environment, can sometimes be helpful in understanding how things are actually working out. Finally, I'll browse over to CloudWatch, Amazon's resource and application monitoring service, to see what's been set up for us there. There are two alarms, one to alert Beanstalk when demand suggests it's time to scale up the available resources, and one to tell us when it's time to scale down. Overall, I'd say that's an impressive range of resources and infrastructure management tools to be the result of a single terminal command. Let's review what we've seen in this module. Among the services launched with at least some Elastic Beanstalk applications, is an ECS optimized EC2 instance that comes with a live Docker container running as an agent on behalf of EC2 container service administration tools. ECS, which runs a lot of the Elastic Beanstalk service behind the scenes, is made up of clusters that contain tasks, container definitions, and services, and customized and fully-integrated infrastructure elements are automatically added to a Beanstalk application within EC2, IAM, S3, and CloudWatch. In the final module of this course, we'll put together a full, multi-container application and see what happens.

Deploy a Full-sized Docker/Elastic Beanstalk Project

Deploy a Multi-container Application Locally

Okay, so it's time to use everything we've learned up to this point to build a working, multi-container Docker environment on Elastic Beanstalk. The application I've got in mind is a two-tiered WordPress deployment. Two tiers in this case, means that the WordPress installation will live in a separate container from the database that drives it. This also means that we'll need to do more to get both containers running. We'll also have to make sure that they're fully aware of each other, so data can be read and written back and forth. Effectively, all we'll need to do is write a single, relatively brief, Dockerrun.aws.json file identifying all the parts and their environment settings, and Beanstalk will do the rest. But before we do that, I thought it would be useful to first see how you can do exactly the same thing locally using Docker Engine. I'm going to run two containers, the first using the official image of the mariadb database, and the second using the official WordPress image. They've both already been pulled to my local machine, so things should move along really quickly. The run command I'm using for mariadb is pretty much straight forward, except for one volume and two environment arguments, which I guess means that it's not so simple after all. First off though, I'm naming the container wordpressdb, and telling it to use the latest mariadb image available. The first environment value that follows the -e flag, passes in the database root password. That's right, I'm using password for my password. Naturally, I'm only going with such a weak password because you're watching, and I'm not quite 100% certain that you won't share one of my real passwords. The second environment setting creates a new database called wordpress. By default, that's the name WordPress itself will look for as it sets itself up. The -v flag will create a new volume on the Docker host machine, where persistent database data can be stored outside the container. If you really want to know, that volume will be located on your host deep inside the file system beneath the var/lib/docker/aufs/mnt directory. I'll let the mariadb container loose, and turn my attention to WordPress. This docker run command tells WordPress about the database password I set just before. Without that of course, there'll be no way for us to edit configurations or create blog posts. The --link argument is what makes this whole setup work, and arguably is our most important takeaway from this process. Link tells WordPress to looks for its database on the wordpressdb container. The environment flag allows that to happen by sharing the database root password. The -p flag allows incoming traffic through port 80 by way of the host machine's IP address. Once that's running, I can move over to a browser, navigate to the host IP address, and get to work configuring my WordPress site. Just to emphasize, however, the main point of this little exercise was to show you how cluster environment values are passed to containers at launch time, and how containers can be connected to each other through the link argument. I should note, by the way, that the use of the --link argument in Docker networking has been officially deprecated by Docker, and may disappear altogether at some time in the future. I used it here to illustrate how the similar, but not identical, links directive will work with Elastic Beanstalk scripts. That, you'll see in the next clips.

Build a Beanstalk Environment for a Multi-container Application

Despite the relative complexity of a WordPress deployment, our configuration will be rather simple. For instance, because this particular application won't require any standalone data volumes, we're not going to need to define any mount points the way we did in previous examples. And because we're using only official publicly available images, we won't have to create an S3 bucket with an authentication file to get us into a private image repository. The problem with that, is that I did already promise I would show you how it's done. So here's how it would go. I'll log into Docker Hub from my local command line, which will generate an authentication config file. I'll use its contents to create a dockerconfig file, and install the AWS CLI, which I'll then use to create a new S3 bucket, into which I will then copy my config file. Assuming that the private image your Beanstalk infrastructure will want to access is on Docker Hub, log in to your account from the command line on a machine running Docker Engine. This will create a hidden directory in your home directory called .docker. If you change to that directory, and then list its contents, you'll see a file called config.json, which as you see when you peek inside, contains an authentication string. I'll copy the file, given it a new name, and open the new version in my text editor. Here, I'll need to remove the outside auths block, including the outside curly braces, and save the file. Moving the new file back to my home directory will make things a little bit easier at the next stage. Now I'll install the AWS Command Line Interface. Don't confuse this CLI with the Elastic Beanstalk CLI I installed earlier. That one contains full functionality that's specific to Beanstalk, while the one I'll show you this time is a more general purpose tool for basic administration across the full range of AWS services, including Beanstalk. Since I've already got Python's packages manager of pip installed on this machine from our installation of the Beanstalk CLI, I'll get the AWS CLI going using the pip method, all in one command. Aws configure will take us through the setup process, during which I'll paste the security credentials I've been keeping on hand for just this purpose. When it comes to S3, the CLI works a lot like your own local command line. Typing aws s3 ls will list all the buckets I've already got, and aws s3 mb will make a new bucket. I'll call it mycreds, and then add a few random numbers since S3 bucket names must be globally unique. Uploading the authentication file is also dead easy, aws s3 cp, the name of the file, and the name of the target bucket. We're all set. The only other piece of information you'll need is the way you'll call the config file from within a Dockerrun.aws.json file. And that, as I showed you earlier, will look a lot like this. So at least I made good on my promise. At the time of course creation at least, whenever you launch a multi-container environment from the Beanstalk CLI, a rather abrupt warning will be displayed, telling you that such environments require additional ECS permissions, and that you'd better go about adding said permissions to the AWS Elastic Beanstalk EC2 role, or else. The warning points us to an AWS documentation page that no longer provides the information that we'd need to get this done. I'm sure that AWS will get around to fixing this soon enough, but there's a pretty good guide here on this page. However, I've found that at least the deployments I've recently launched have been working just fine without any extra permissions policies. In case your mileage varies, you can easily create a new policy and attach it to the Beanstalk Roles that you'll find on the IAM Roles pages, once you've started up at least one Beanstalk environment, or by creating your own and then searching through the Policies menu, selecting the ones that meet your needs, and then attaching them to a appropriate role. You can also create your own policies and then attach them to your role. But as I said, that doesn't concern us right now. Now, we're just about ready to launch our WordPress environment.

Understand and Launch a Two-tiered (WordPress) Docker/Beanstalk App

Let's go over the contents of our Dockerrun.aws.json file. The first container definition is for our database, using the most recent image of mariadb. Mariadb, by the way, is a very popular, fully compatible break-off fork of the MySQL database and uses all the exact same MySQL commands. Once again, since their application requires both containers to be running, I'm setting essential to true. The memory allotted to the container is 128 MB. This, like everything in the file, is critical. I can tell you from bitter personal experience of leaving the memory setting out will kill the application. I also set communication with the container to the MySQL default port of 3306. These environment settings determine the database root password and the name of new WordPress database that will be created on startup. Get those exactly right, they make a big difference. Moving down to the WordPress container, note the port mapping value is set to 80, which is the HTTP default that your users will use to get to your site. The links setting points to the name of our mariadb container and the MYSQL\_ROOT\_PASSWORD; spelling counts. As I mentioned earlier, the local run CLI feature lets you try out your configuration locally without having to wait 10 or 15 minutes for Beanstalk to pull everything together on AWS. As I also mentioned, that feature happens to be broken right now. But I'm confident that by the time you see this, everything will be working once again. Just run eb local run, and you'll see plenty of log messages telling you how things are going. One thing we can do to check our work, is paste the contents of our Dockerrun.aws.json file into an online JSON format tool like this one at jsonformatter.org. Getting JSON right can be really fiddly, and a second opinion can be really helpful. It looks like my version is fine. By the way, the full text of this and all other files I've used through this course, can be seen at my course page, bootstrap-it.com/docker-beanstalk. Now we're finally ready to go. Typing eb create, and a name you'd like to give your environment, is all it takes. I'll pause the video and start it up again in a few minutes once the setup is complete. Everything seems ready for action now. I'll click on the endpoint URL, select a language for the WordPress installation, and enter the information it will need to access the database. The user is root, the password is password, but don't tell anyone, and the host in this case is not local host, but the private IP address of the mariadb container. This will almost certainly be 172.17.0.2, as it is in this case. If you're not sure, you can check it out manually by SSH'ing into the EC2 host from the terminal, and running docker network inspect bridge. Uh oh, you may face this problem. It's just a permissions thing. You can either use sudo, or instead, using the more secure approach, add the EC2 user to the Docker group through gpasswd, and then logging out and in again. Either way will work. Just to save time, I'll use sudo. Inspect shows us two running containers. The IP of this first one is 172.17.0.2, and the second is 0.3. Since the mariadb container was loaded first in the Dockerrun file, it stands to reason that it got 172.17.0.2. This second container is named ecs-awseb-wordpress, where eb stands for Elastic Beanstalk, while the first one will be identified with mariadb. So we're up and running. If you've successfully followed along on your own up until this point, give yourself a hardy slap on the back. If you haven't, I would still congratulate yourself for having hung on listening to me droning for an hour or so. Before we go, however, there's one more thing I'd like to show you. If we visit the EC2 container service, which if you'll recall the way I described it earlier, is the engine invisibly driving a lot of Beanstalk's magic, we should see some signs of what's really happening. I'll click on Task Definitions, and then on the link to our task. By expanding the mariadb and wordpress sections, we can see our settings neatly displayed. Below those, are references to our live volumes, and finally, back up at the top, is the JSON tab, which contains a variation of our own Dockerrun file. Well let's review what we've seen in this module. We learned how we can set environment values with Docker containers, and how they can communicate with each other within a cluster. Even though we didn't need it for our current project, we then learned how to create a Docker config file from the config.json file that's automatically created when we log in to a remote repository. We then installed the general AWS CLI and copied our new config file to an S3 bucket, then we worked through our Dockerrun file and saw how to set port mappings and various environment variables. Finally, we launched our project, created WordPress databases for it, and inspected its environment.

Course Review

Let's take a quick look back at everything we've learned in this course. We began by describing how clusters of containers can be used to deliver microservices, containers cooperatively providing discrete services as parts of a much larger deployment. We discussed the built-in script ability of Docker containers, along with their incredible speed and light weight, and how those qualities make them natural candidates for this kind of compute model. We then got ourselves introduced to the basics of Docker clustering through Docker Swarm mode, and then at the basic structure of AWS's Elastic Beanstalk, how it abstracts the complexity of cluster management, while harnessing the power of a very wide range of AWS resources, including EC2, IAM, S3, elastic load balancing, and of course, the EC2 container service. We took a close look at single-container environments on Beanstalk using Dockerrun.aws.json version 1 files, and multiple-container environments using Dockerrun-aws.json version 2. We also talked about preconfigured AMIs that provide various out-of-the-box Docker platforms. Finally, we put all the pieces together to properly understand the design and administration of a full, multi-container environment by building a two-tiered WordPress application. So that's that. I hope you've been enriched and entertained by this course, and that you feel you're now ready to tackle some serious infrastructure construction on your own. Hope to see you again soon.