Introduction to Kubernetes

In this lesson you will learn about the need for container orchestration and how Kubernetes can help you deal with the multiple challenges of running containers in a Production environment.

Why container orchestration?

Let's bring back our production box, which from here on we will refer to as just a node. This node is running the Docker engine, which allows it to run Docker containers, like our Catalog container and our Inventory container.

Now, we know that the node is capable of running containers, but there are several questions that we have not answered yet:

- To start with, who pulls the containers into the node? Do we want to manually pull images into this node any time we release a new version? It's probably going to be a time consuming and error prone task

- Next, how to run each of these containers properly? As you know, you many need to provide multiple parameters to the docker run command. Do we perhaps create scripts to run each of our containers properly?

- Then, how to tell if we have enough container instances to handle our load? What if we need two or four more containers for each microservice? Do we want to manually run these additional containers when needed?

- How do we decide which is the best node for each container? You will likely have not just one but multiple nodes available. How to balance our containers properly?

- What if one of the containers crashes due to a very unexpected bug. Who is going to be in charge of noticing this issue and bring the missing container back?

- How do we provide all the configuration values and secrets required to run the microservices?

- Also, let's say the Inventory container wants to talk to the Catalog container, perhaps to get a list of items. How would Inventory know how to locate the Catalog container? Consider that it could live in the same node or in any other node.

- What if one of your client applications want to reach the container from the outside? How to provide a public IP address to the node?

- And, once the external request reaches the node, how to route it to the appropriate container instance?

These are some of the reasons why you need some form of container orchestrator, and this is exactly what Kubernetes was created for.

Enter Kubernetes

So, let's say you have a couple of nodes and you would like to run your Docker containers there. How can Kubernetes help you with this?

Well, the main job of Kubernetes is to place your containers into pods that can run in your nodes. A pod is the smallest deployable unit of computing in Kubernetes and is a group of one or more containers that share storage and network resources.

So, how to get the container into a pod in a node? Well, one of the main benefits of Kubernetes is that it can turn desired state into actual state. You can do this by creating a deployment, where you specify the name of the image to deploy, how many replicas of the pod you would like to create, and many other details.

You provide this deployment to the Kubernetes control plane, which contains the services needed to run pods. The control plane will go ahead and pull down the specified Docker image from the corresponding container registry into the node and then it will use it to create the requested pod.

Kubernetes can also handle the selection of nodes for your pods. So, if you want to add a second replica of your Catalog pod, all you have to do is modify the number of replicas in your deployment and Kubernetes will spin up a second pod in whichever node has free resources available.

This should also give you an idea of how easy it is to scale your microservices in Kubernetes. If you would like a more automated approach, you can also enable auto scaling. Using an horizontal pod auto scaler, or HPA, you can define resource thresholds that can trigger the creation of new pods automatically.

For instance, you can define that none of the pods should use more than 50% of CPU, and if some pod reaches that usage, a new pod should be automatically created to deal with the excess, keeping a minimum of 1 pod and growing up to a total of 10 pods in this example.

Another key feature of Kubernetes is its self-healing capability. If one of your pods crashes, for whatever reason, Kubernetes will immediately notice that the requested number of replicas does not match the number of existing pods, so it will provision a new pod right away, without you having to intervene in any way.

Kubernetes can also store both configuration data and secret values and can make them available to your pods as environment variables.

Now, how about service discovery? Let's say that the Inventory pod wants to make a call to the Catalog pod. To enable this, you can create a Kubernetes service. This service gets assigned a name and an IP address discoverable by any pod running in the nodes managed by Kubernetes.

Here, Inventory can make a call to the service configured for Catalog, and the service will route it to one of the Catalog pods. This also highlights the load balancing capability of Kubernetes, because when you

have multiple pods running your microservice, the service will decide how to load balance between them, without you having to worry about this at all.

What if you wanted an external client to access your Catalog pods say from the Internet? All you have to do is change your Kubernetes service to be of the load balancer type, and it will acquire an external IP that clients will be able to reach, assuming you are running Kubernetes in a public cloud provider.

Another great feature of Kubernetes is how it can gradually rollout new container versions with zero downtime. So, image you have modified your Inventory deployment to use the Inventory V2 image.

Instead of just destroying the existing inventory pod and create a new one, Kubernetes will first spin up a new inventory pod, with the new container version, it will wait until it can confirm this new pod is healthy and ready to receive requests, and only then it will go ahead and delete the old pod.

The inventory clients should never notice this version upgrade, provided it is backwards compatible, of course.

So, everything you see here, except for the client and the container registry, is what is known as a Kubernetes cluster. You can create a cluster either in your dev box or in a public cloud provider like Microsoft Azure, Google Cloud or Amazon Web Services.

In the next lesson you will create your first Kubernetes cluster using the Microsoft Azure Kubernetes Service.