**Azure Container Services**

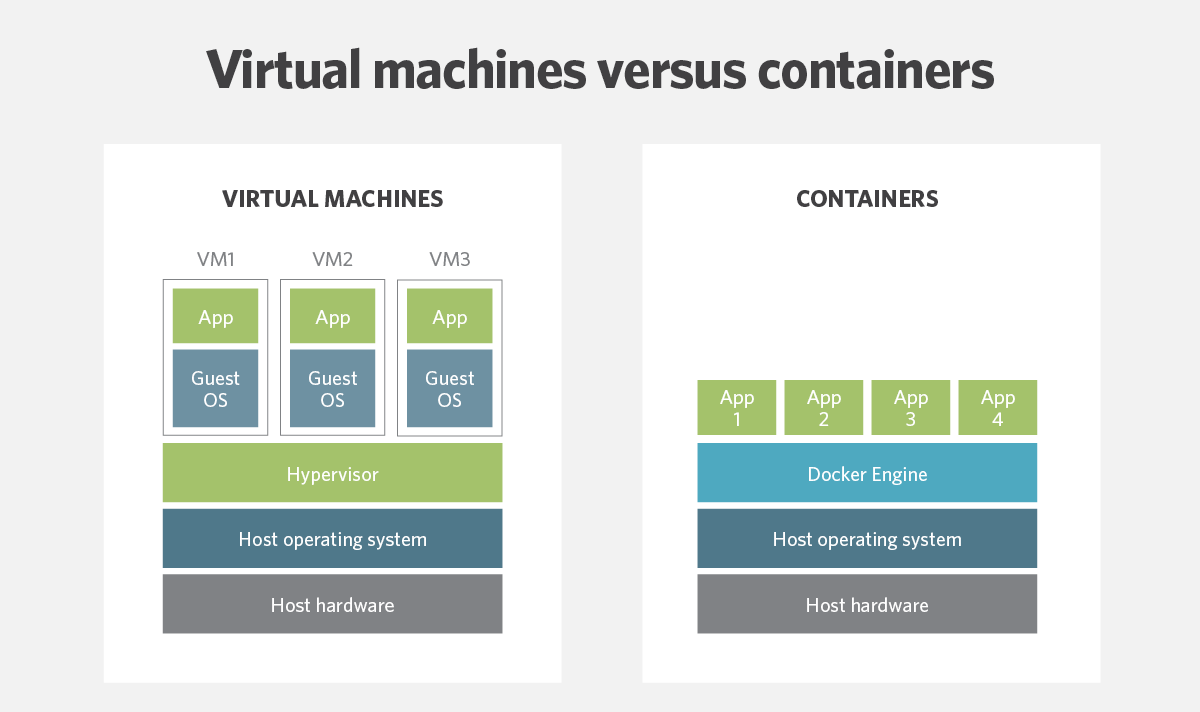
**Containers**

Containerization, also called container-based virtualization and application containerization -- is an OS-level virtualization method for deploying and running distributed applications without launching an entire VM for each application. Instead, multiple isolated systems, called containers, are run on a single control host and access a single kernel.

**VMs Vs Containers**

Containers share the same OS kernel as the host, containers can be more efficient than VMs, which require separate OS instances. Containers hold the components necessary to run the desired software, such as files, environment variables and libraries. The host OS also constrains the container's access to physical resources -- such as CPU and memory -- so a single container cannot consume all of a host's physical resources.

A virtual machine mimics a complete server. In a typical virtualized server, each VM "guest" includes a complete operating system along with any drivers, binaries or libraries, and then the actual application. Each VM then runs atop a hypervisor, which itself runs on a host operating system and in turn operates the physical server hardware.



**What Are the Advantages of Using Containers?**

* The average container size is within the range of tens of MB while VMs can take up several gigabytes. Therefore a server can host significantly more containers than virtual machines.
* Running containers is less resource intensive then running VMs so you can add more computing workload onto the same server.
* Provisioning containers only take a few seconds or less, therefore, the data center can react quickly to a spike in user activity.
* Containers can enable you to easily allocate resources to processes and to run your application in various environments.
* Using containers can decrease the time needed for development, testing, and deployment of applications and services.
* Testing and bug tracking also become less complicated since you there is no difference between running your application locally, on a test server, or in production.
* Containers are a very cost effective solution. They can potentially help you to decrease your operating cost (less servers, less staff) and your development cost (develop for one consistent runtime environment).
* Container-based virtualization are a great option for microservices, DevOps, and continuous deployment.

**What Are the Disadvantages of Using Containers ?**

* One of the main disadvantages of container-based virtualization compared to traditional virtual machines is security. Containers share the kernel, other components of the host operating system, and they have root access. This means that containers are less isolated from each other than virtual machines, and if there is a vulnerability in the kernel it can jeopardize the security of the other containers as well.
* Virtual Machines only share the hypervisor which has less functionality and less prone to attacks than the shared kernels of the containers. The system hardware is presented to the VMs in a virtualized form so intrusions, viruses, and other malicious activities cannot spread over to other VM.
* Less flexibility in operating systems. You need to start a new server to be able to run containers with different operating systems. While virtual machines with any kind of OS can live next to each other on the same server. This might not be a problem for hosting providers, but for complex enterprise application this can be a serious constrain.
* Another challenge is networking. Deploying containers in a sufficiently isolated way while maintaining an adequate network connection can be tricky. There are solutions that are addressing this issue such as Weave Net 1.5 but as it looks like at the moment there is still room to improve.

**Container implementations**

In addition to Docker, CoreOS released a streamlined alternative, called Rocket. And Canonical, developers of the Ubuntu Linux-based OS, announced the LXD containerization engine for Ubuntu, which will also be integrated with OpenStack. Microsoft also partnered with Docker to create Windows Server containers and Hyper-V containers.

Although Docker is the most popular container technology, but there many other container solutions out there:

* LXC
* LXD
* Solaris Zones
* RKT
* BSD Jails
* Windows Server containers and Hyper-V containers

**NonLinux-based Container Solutions**

Microsoft provides two type of container solutions Windows Server Containers and Hyper-V containers. The main difference between two that Windows Server Containers, just like Docker, share the kernel with the container host and the other containers while Hyper-V Containers do not.

**Container Orchestration**

Container orchestration platforms empower users to easily deploy, manage, and scale multi-container based applications in large clusters without having to worry about which server will host a particular container. Container cluster orchestration is also a very competitive space. Some of the most popular vendors are:

* Kubernetes
* Docker Swarm
* Amazon ECS
* Azure Container Service
* Marathon
* CoreOS Fleet
* Open Stack Magnum
* Diego
* Hashicorp Nomad

**Docker**

Docker is the company driving the container movement and the only container platform provider to address every application across the hybrid cloud.

**Docker Images**

A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. Available for both Linux and Windows based apps, containerized software will always run the same, regardless of the environment. Containers isolate software from its surroundings, for example differences between development and staging environments and help reduce conflicts between teams running different software on the same infrastructure.

Images are constructed from filesystem layers and share common files. This minimizes disk usage and image downloads are much faster.

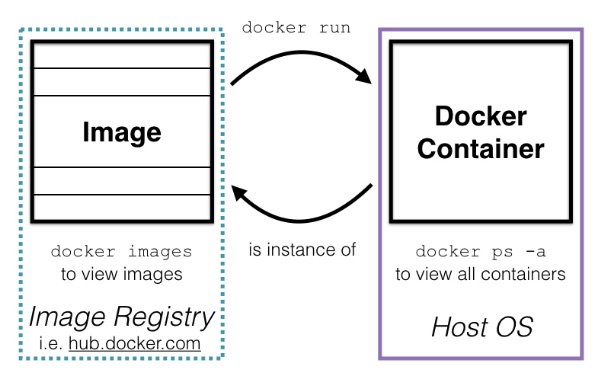
**Docker images VS Containers**

**Images**

* Docker images are the basis of containers.
* An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime.
* Image contains a union of layered filesystems stacked on top of each other.
* Images are read-only.
* Docker images are stored in image registries ( local, docker hub, ACR)

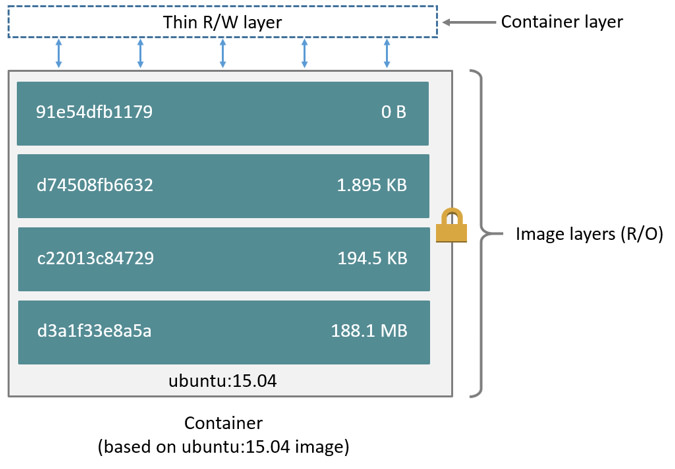
**Containers**

* A container is a runtime instance of a docker image.
* A Docker container consists of
  + A Docker image
  + An execution environment
  + A standard set of instructions



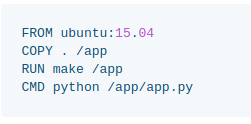
**Docker images and layers**

* Docker image is built up from a series of layers.
* Each layer represents an instruction in the image’s Dockerfile.
* Each layer except the very last one is read-only.



**Dockerfile**

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.



**Dockerfile commands**

**ADD** - ADD command gets two arguments: a source and a destination. It basically copies the files from the source on the host into the container's own filesystem at the set destination. If, however, the source is a URL (e.g. http://github.com/user/file/), then the contents of the URL are downloaded and placed at the destination.

# Usage: ADD [source directory or URL] [destination directory]

ADD /my\_app\_folder /my\_app\_folder

**CMD** - The command CMD, similarly to RUN, can be used for executing a specific command. However, unlike RUN it is not executed during build, but when a container is instantiated using the image being built. Therefore, it should be considered as an initial, default command that gets executed (i.e. run) with the creation of containers based on the image.

# Usage 1: CMD application "argument", "argument", ..

CMD "echo" "Hello docker!"

**ENTRYPOINT** - ENTRYPOINT argument sets the concrete default application that is used every time a container is created using the image. For example, if you have installed a specific application inside an image and you will use this image to only run that application, you can state it with ENTRYPOINT and whenever a container is created from that image, your application will be the target.

If you couple ENTRYPOINT with CMD, you can remove "application" from CMD and just leave "arguments" which will be passed to the ENTRYPOINT.

# Usage: ENTRYPOINT application "argument", "argument", ..

# Remember: arguments are optional. They can be provided by CMD

# or during the creation of a container.

ENTRYPOINT echo

# Usage example with CMD:

# Arguments set with CMD can be overridden during \*run\*

CMD "Hello docker!"

ENTRYPOINT echo

**ENV** - The ENV command is used to set the environment variables (one or more). These variables consist of “key = value” pairs which can be accessed within the container by scripts and applications alike. This functionality of docker offers an enormous amount of flexibility for running programs.

# Usage: ENV key value

ENV SERVER\_WORKS 4

**EXPOSE** - The EXPOSE command is used to associate a specified port to enable networking between the running process inside the container and the outside world (i.e. the host).

# Usage: EXPOSE [port]

EXPOSE 8080

**FROM** - It defines the base image to use to start the build process. It can be any image, including the ones you have created previously. If a FROM image is not found on the host, docker will try to find it (and download) from the docker image index. It needs to be the first command declared inside a Dockerfile.

# Usage: FROM [image name]

FROM ubuntu

**MAINTAINER** - One of the commands that can be set anywhere in the file - although it would be better if it was declared on top - is MAINTAINER. This non-executing command declares the author, hence setting the author field of the images. It should come nonetheless after FROM.

# Usage: MAINTAINER [name]

MAINTAINER authors\_name

**RUN** - The RUN command is the central executing directive for Dockerfiles. It takes a command as its argument and runs it to form the image. Unlike CMD, it actually is used to build the image (forming another layer on top of the previous one which is committed).

# Usage: RUN [command]

RUN aptitude install -y riak

**USER** - The USER directive is used to set the UID (or username) which is to run the container based on the image being built.

# Usage: USER [UID]

USER 751

**VOLUME** - The VOLUME command is used to enable access from your container to a directory on the host machine (i.e. mounting it).

# Usage: VOLUME ["/dir\_1", "/dir\_2" ..]

VOLUME ["/my\_files"]

**WORKDIR** - The WORKDIR directive is used to set where the command defined with CMD is to be executed.

# Usage: WORKDIR /path

WORKDIR ~/

**Example Docker file**

############################################################

# Dockerfile to build MongoDB container images

# Based on Ubuntu

############################################################

# Set the base image to Ubuntu

FROM ubuntu

# File Author / Maintainer

MAINTAINER Example McAuthor

# Update the repository sources list

RUN apt-get update

################## BEGIN INSTALLATION ######################

# Install MongoDB Following the Instructions at MongoDB Docs

# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

# Add the package verification key

RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

# Add MongoDB to the repository sources list

RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | tee /etc/apt/sources.list.d/mongodb.list

# Update the repository sources list once more

RUN apt-get update

# Install MongoDB package (.deb)

RUN apt-get install -y mongodb-10gen

# Create the default data directory

RUN mkdir -p /data/db

##################### INSTALLATION END #####################

# Expose the default port

EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)

CMD ["--port 27017"]

# Set default container command

ENTRYPOINT usr/bin/mongod

**Building Docker images**

Install docker on your machine. For windows you can install Docker for windows. For Linux servers install Docker for Linux. For old platforms which does not meet the requirements for Docker for Windows, you can install Docker toolbox.

**Docker Toolbox**

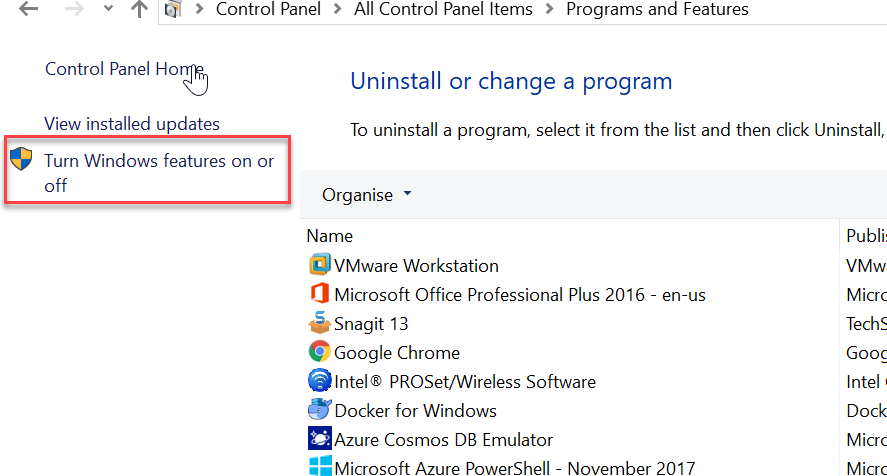
Docker Toolbox is an installer for quick setup and launch of a Docker environment on older Mac and Windows systems that do not meet the requirements of the new Docker for Mac and Docker for Windows apps.

Toolbox includes these Docker tools:

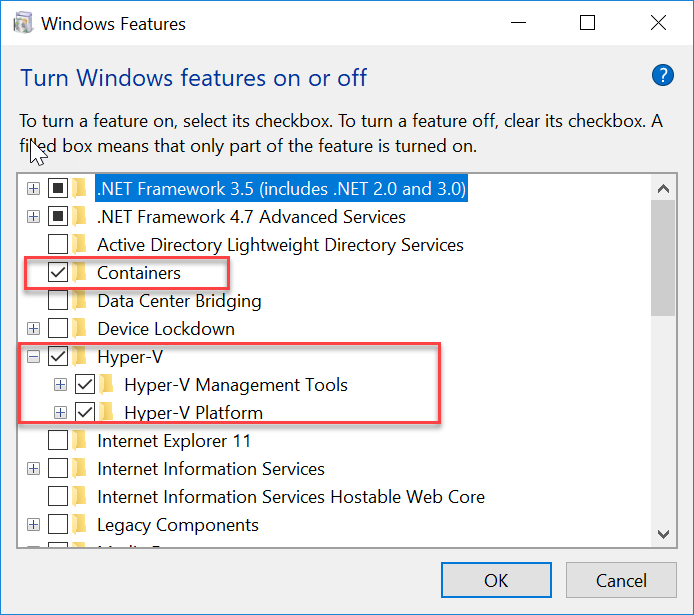
* Docker Machine for running docker-machine commands
* Docker Engine for running the docker commands
* Docker Compose for running the docker-compose commands
* Kitematic, the Docker GUI
* A shell preconfigured for a Docker command-line environment
* Oracle VirtualBox

**Working with Docker for Windows**

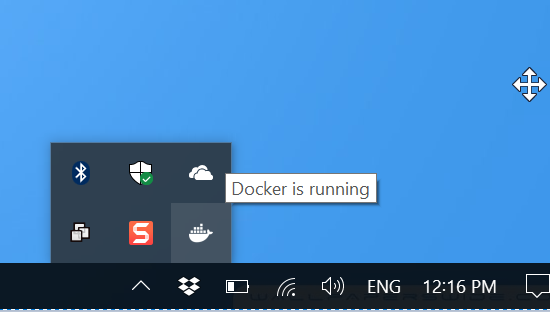
1. Install Windows Server 2016 or Windows 10 Pro/Ent
2. Enable Hyper-V from ‘Programs and Features’ > ‘Turn Windows Features on and off’.



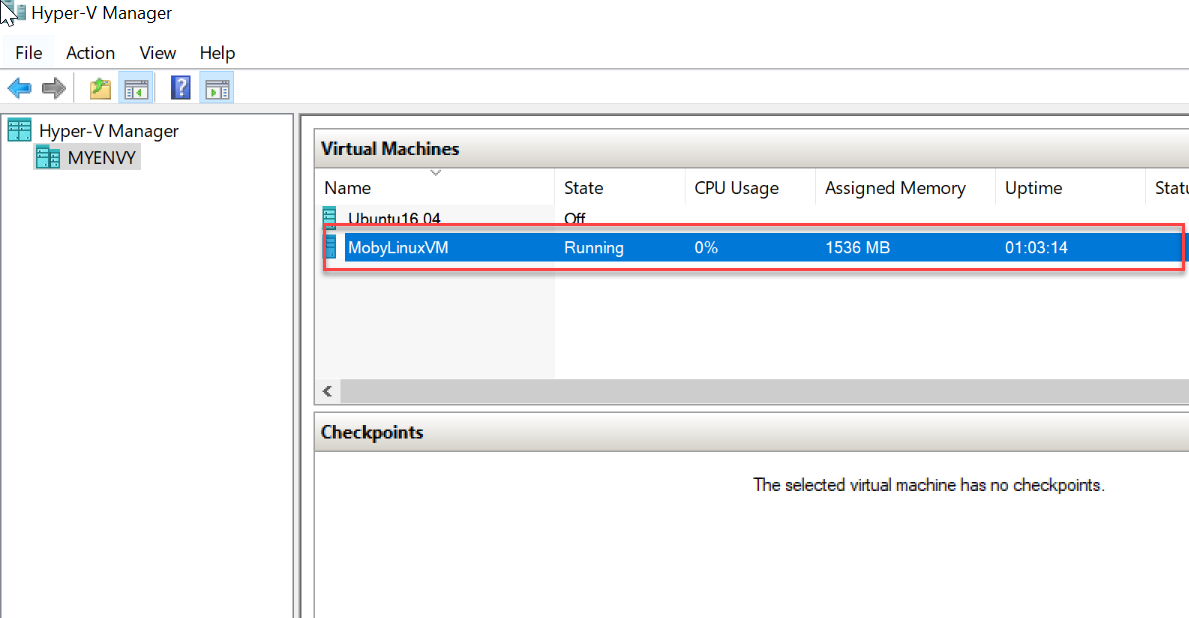
1. Enable ‘Containers’ and ‘Hyper-V’ features



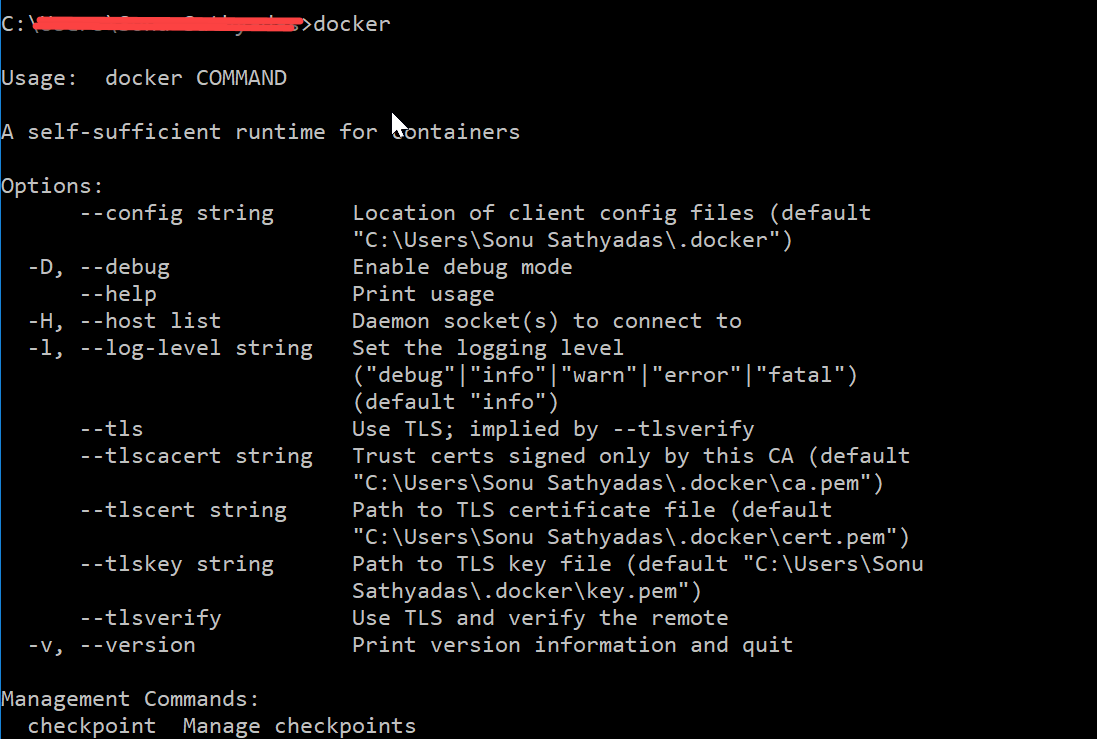
1. Once the installation completed, restart the machine.
2. Now, Install Docker for Windows. Get it from <https://store.docker.com/editions/community/docker-ce-desktop-windows>
3. Once the installation completed Docker icon will be displayed in ‘System Tray’.



1. You can see a VM is running in Hyper-V for docker daemon. Open the Hyper-V Manager.



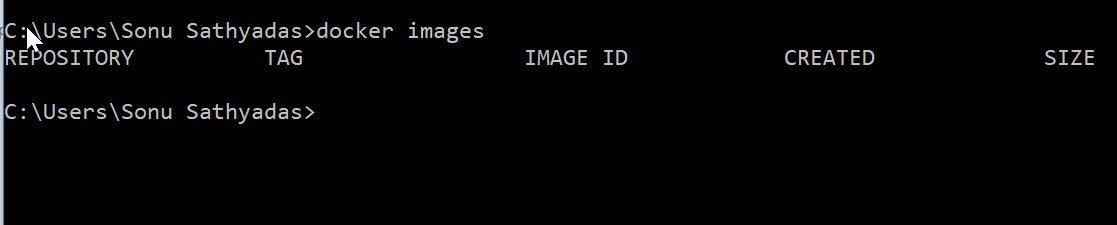
1. Now, You are able to run the docker commands. Open the command prompt and type **docker** and press enter key.

****

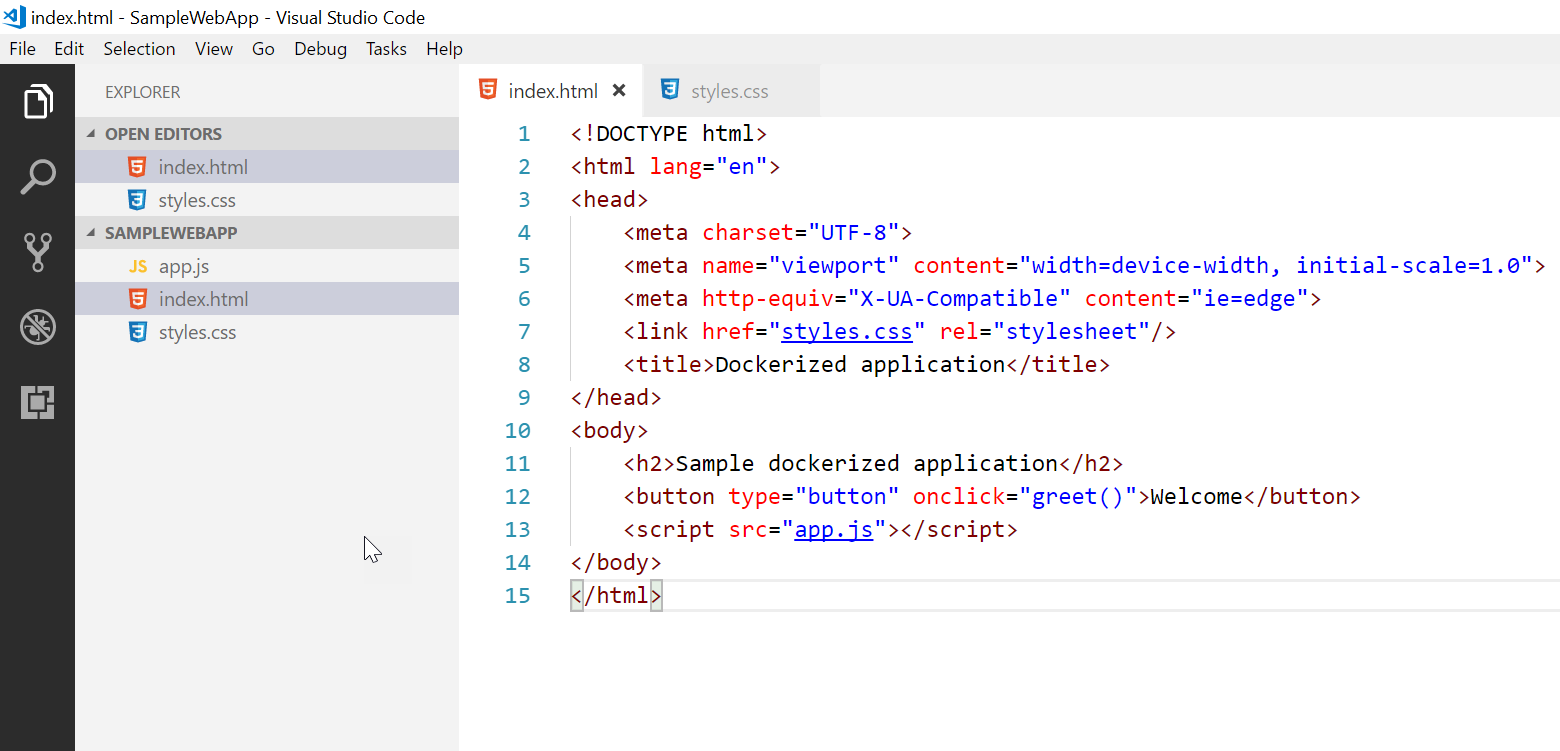
1. You are ready with the docker installation on Windows.

**Creating Docker images**

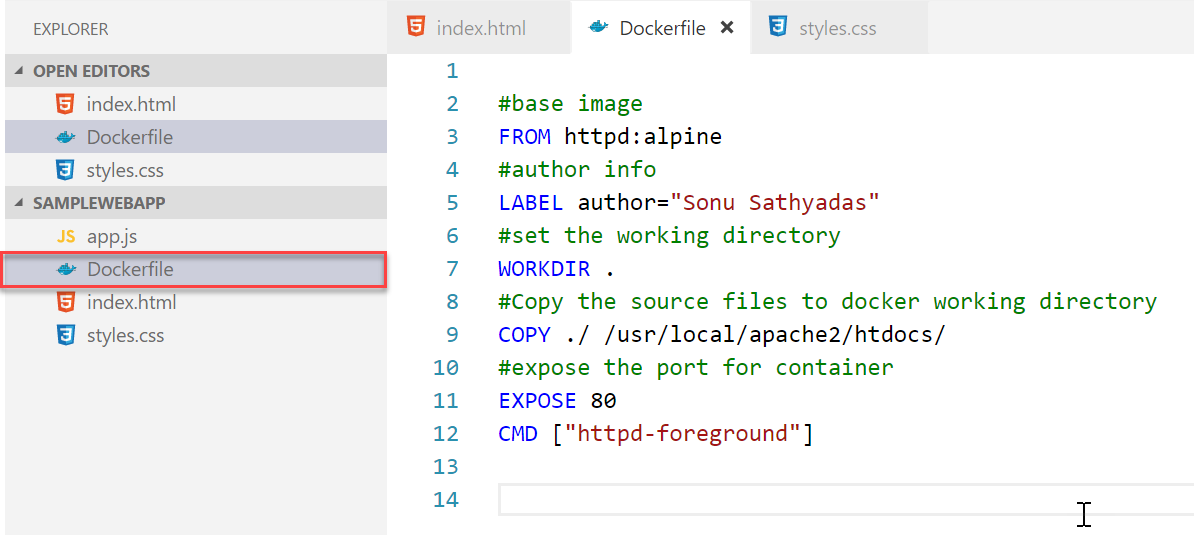
1. Open the command prompt and type ‘**docker images’** to list all the docker commands.



1. It shows an empty list of docker images. You can create a new docker image in you local repository using ‘**docker build’** command.
2. Create a simple web application with one html page, one CSS file and one JS file using Visual Studio Code.



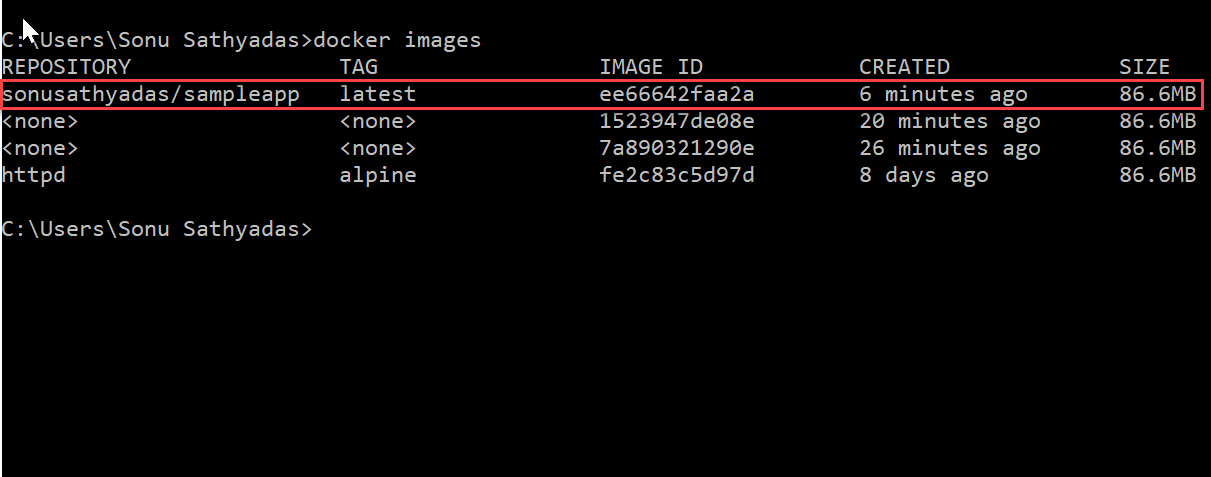
1. Add a docker file to the project. Add the following code in the docker file.



1. Build the docker image with the following command

**docker build -t sonusathyadas/sampleapp:latest .**

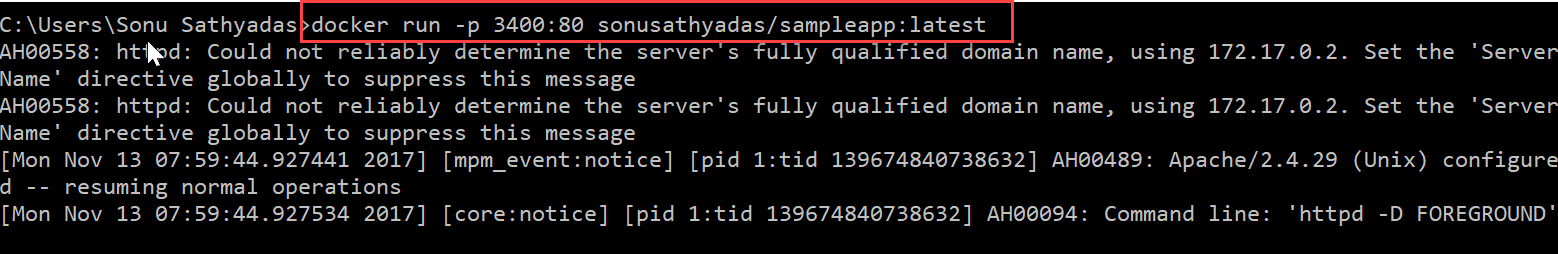
1. It creates the docker images in your local repo. Run the ‘**docker images’** command to see the list of all images

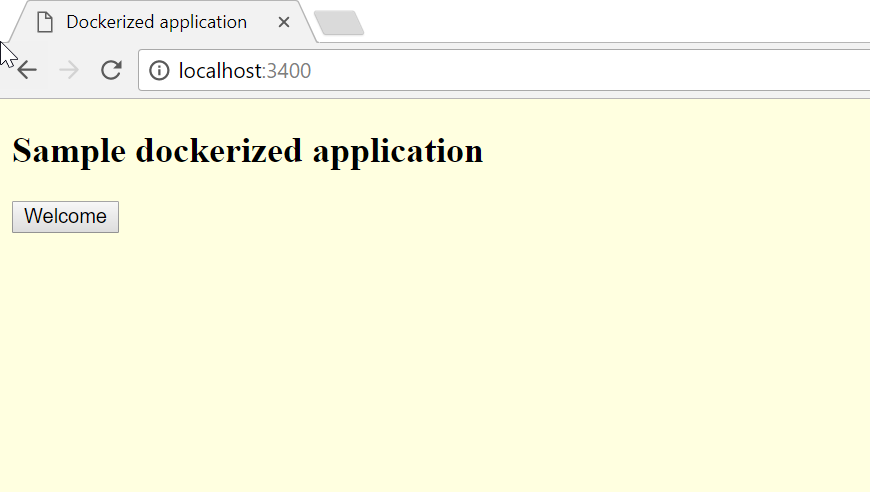


1. Run the docker image by running the following command.

**docker run -p 3400:80 sonusathyadas/sampleapp:latest**

1. The application start running on port number 3400.



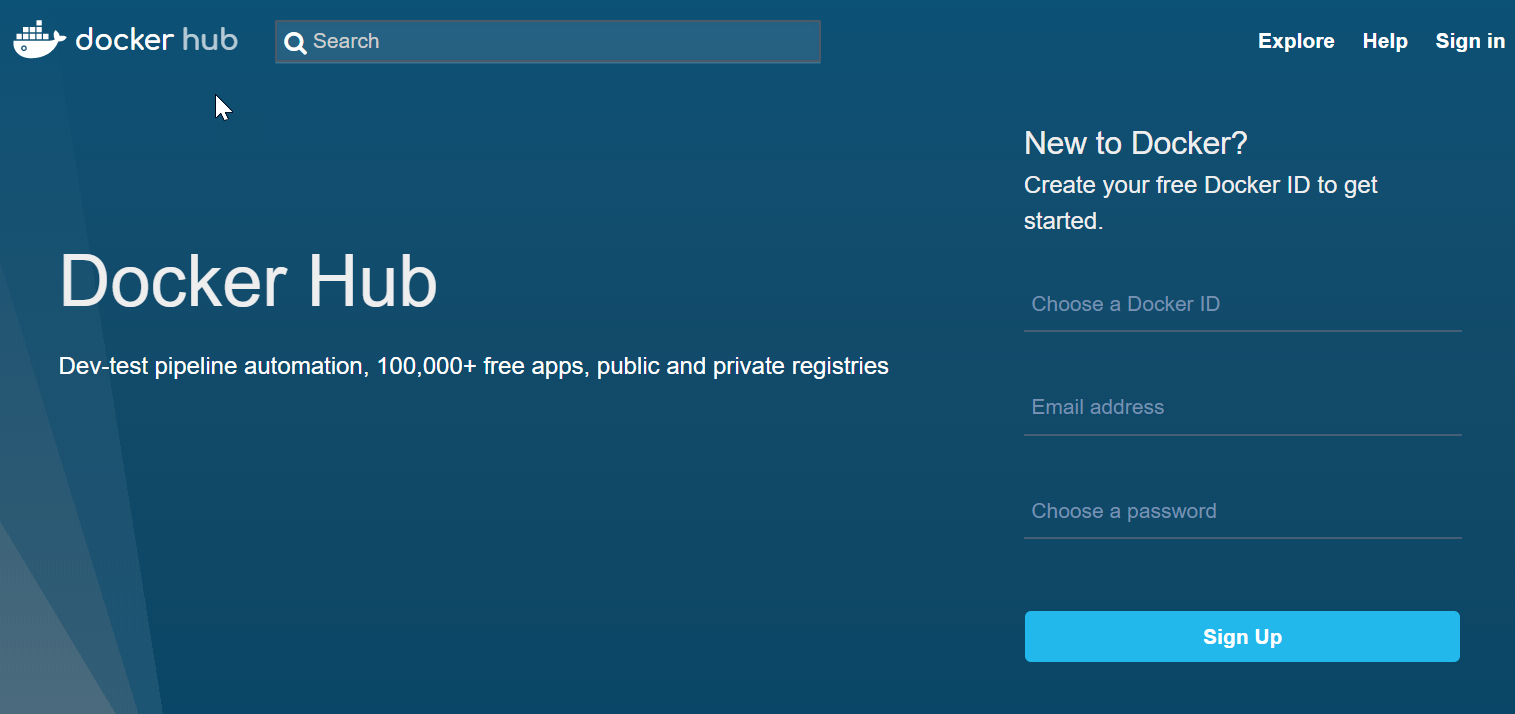
1. Open browser and navigate to <http://localhost:3400> 

**Docker Hub**

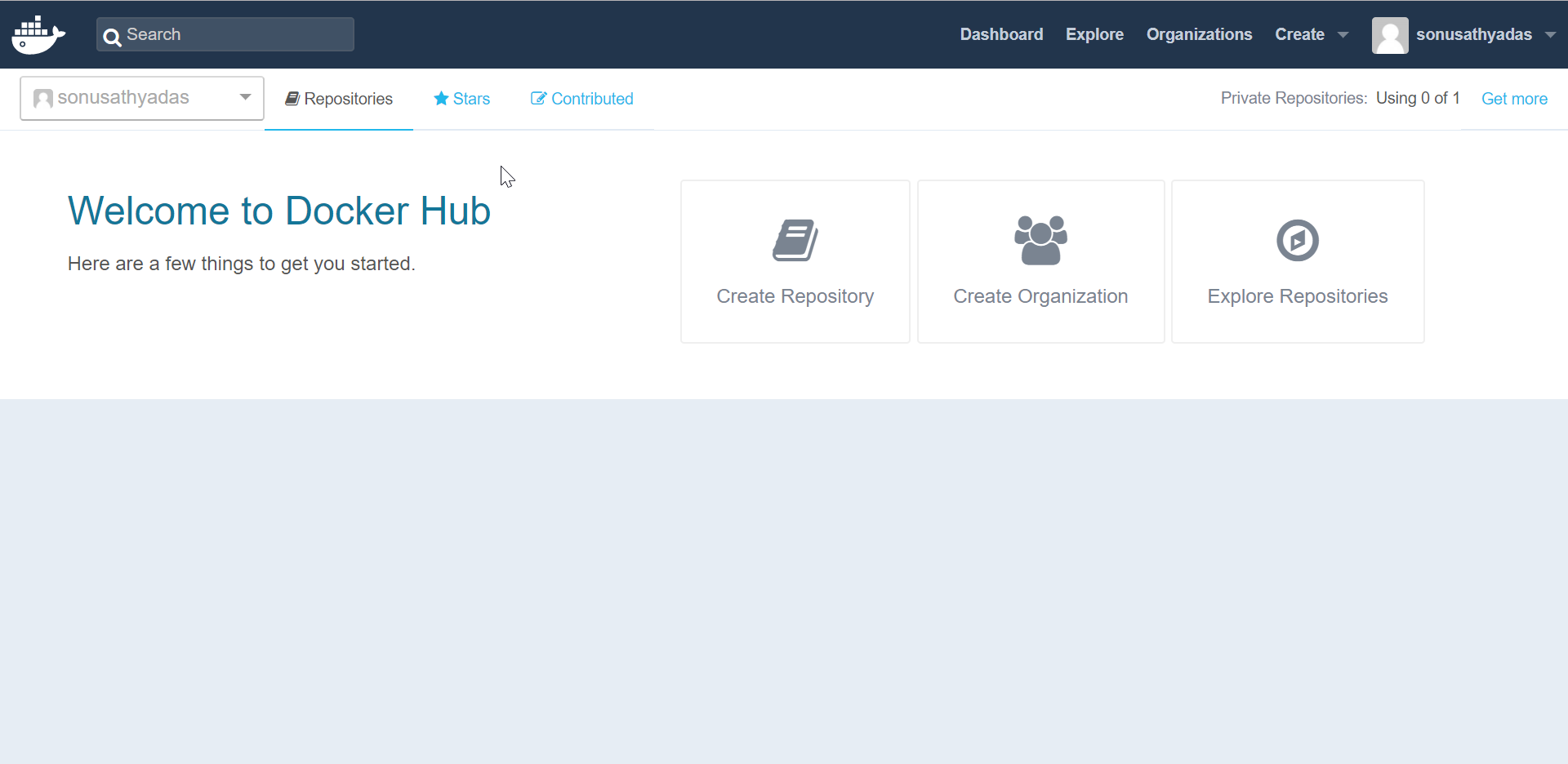
Docker hub is a repository for docker images. You can create Public or private repositories. You can also create Organization in Docker Cloud that contains Teams, and each Team contains users. Once the account is created you can Push images to Docker hub using **docker push** command. It is also Pull the image using **‘docker pull’** command.

**Push the image to docker hub**

1. Create a Docker Hub account, if you don’t have an account.
2. Navigate to <https://hub.docker.com> and fill the details for the sign up page.



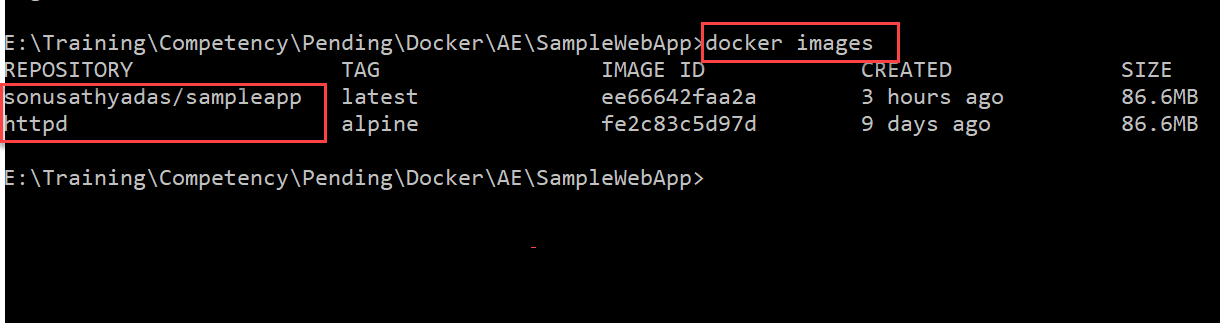
1. Once the account is created, you can login to the docker hub account. Once logged in you will be redirected to the Dashboard.



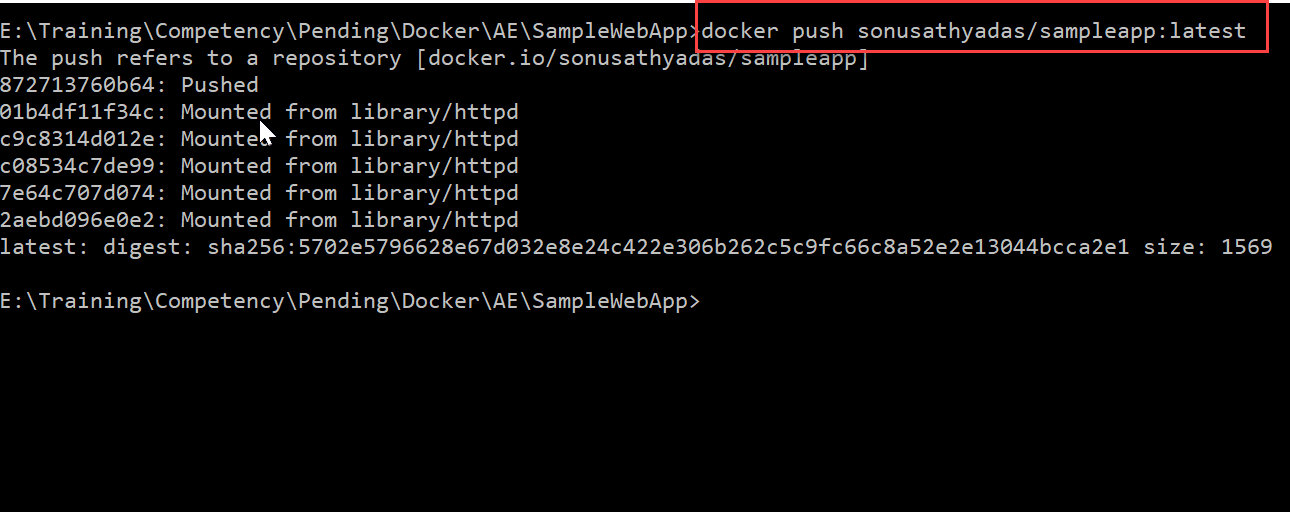
1. Now, you can connect to docker hub from your windows command prompt. Open the command prompt and type the ‘docker login’ command. Enter the username and password for docker hub and you will successfully login to the docker hub.



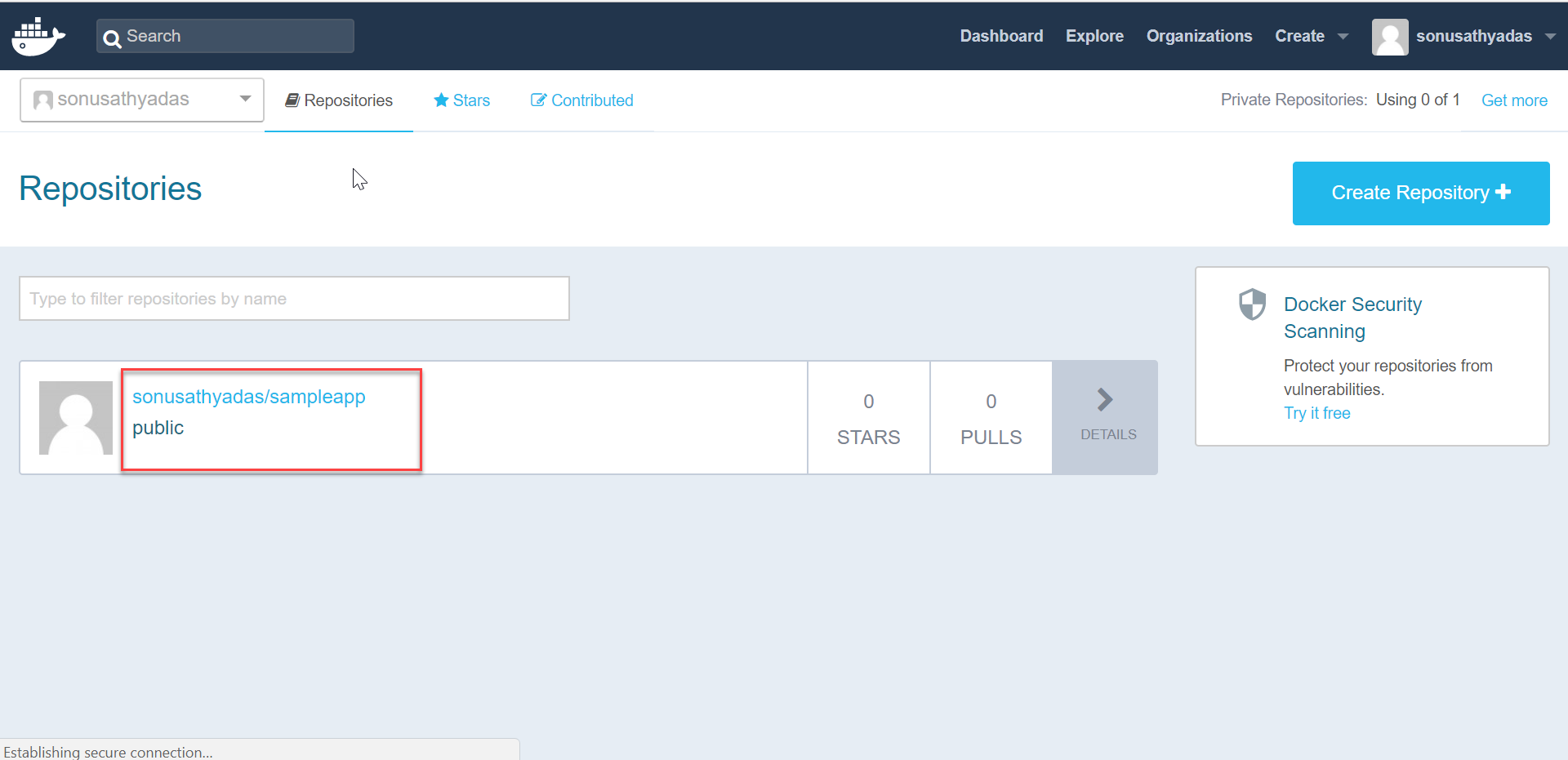
1. List all the images using ‘docker images’ command



1. Upload the image which you want using docker

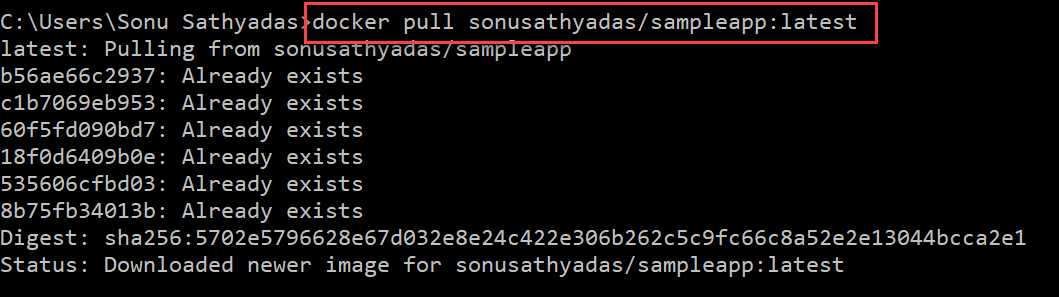


1. Now you can see the uploaded image in the docker hub portal.

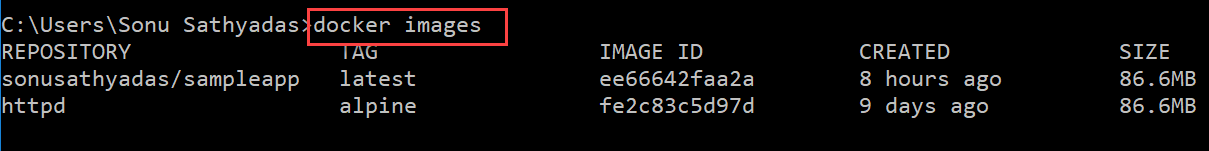


**Pull images from Docker Hub**

1. Open Command prompt. Run the ***docker pull*** command to pull the image from docker hub.



1. List the images using ***docker images*** command.



**Pricing of Docker Hub**



**Azure Container Registry.**

Azure Container Registry is a private registry for hosting container images. Using the Azure Container Registry, you can store Docker-formatted images for all types of container deployments. Azure Container Registry integrates well with orchestrators hosted in Azure Container Service, including Docker Swarm, DC/OS, and Kubernetes.

Use Azure Container Registry to:

* Store and manage container images across all types of Azure deployments
* Use familiar, open-source Docker command line interface (CLI) tools
* Keep container images near deployments to reduce latency and costs
* Simplify registry access management with Azure Active Directory
* Maintain Windows and Linux container images in a single Docker registry

There are three SKU available for Azure Container Registry

1. Basic
2. Standard
3. Premium

| SKU | Description |
| --- | --- |
| Basic | A cost-optimized entry point for developers learning about Azure Container Registry. Basic registries have the same programmatic capabilities as Standard and Premium (Azure Active Directory authentication integration, image deletion, and web hooks), however, there are size and usage constraints. |
| Standard | The Standard registry offers the same capabilities as Basic, but with increased storage limits and image throughput. Standard registries should satisfy the needs of most production scenarios. |
| Premium | Premium registries have higher limits on constraints, such as storage and concurrent operations, including enhanced storage capabilities to support high-volume scenarios. In addition to higher image throughput capacity, Premium adds features like geo-replication for managing a single registry across multiple regions, maintaining a network-close registry to each deployment. |

**Create an Azure Container Registry using Azure CLI**

1. Login to Azure using Azure CLI

**az login**

1. Register module for ACR

az provider register –n Microsoft.ContainerRegistry

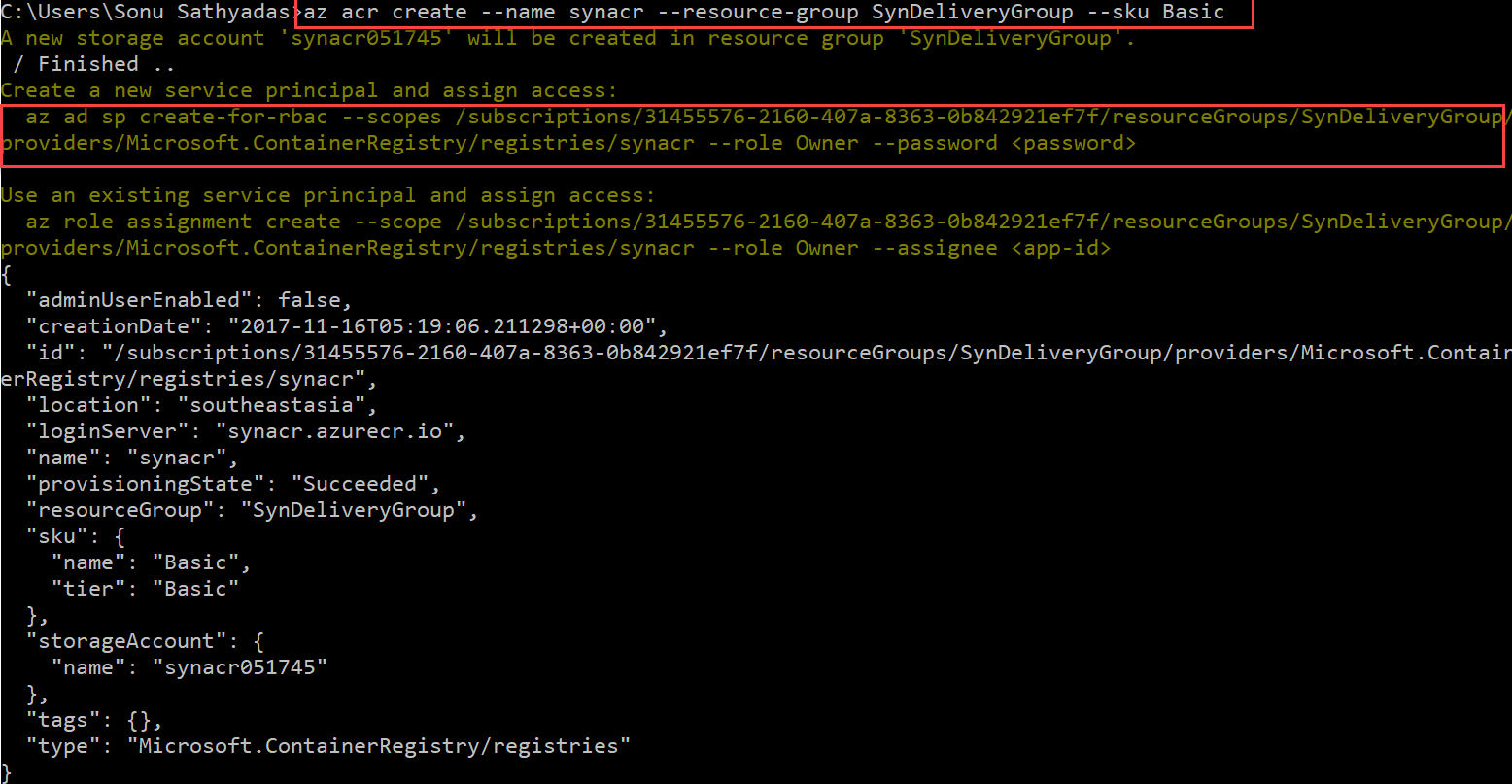
1. You can create a resource group, if already exists you can use the existing resource group. Create the group in ‘EastUs’,’WestUS’,’Westeurope’. Because **az continer create** command can create containers in those regions. Currently only those regions supports that service.

**az group create --name “<res\_Grp>” --location “<location>”**

1. Create a ACR using the following command.

**az acr create --name “<acr\_name>” --resource-group “<res\_grp>” --sku Basic/Standard/Premium**

**Note**: Container Registry name should not container space or hyphen or any other special characters. Lower case or upper-case letters with numbers allowed.



1. Once the ACR is created, you need to enable the admin login for the acr.

**az acr update -n “<acr\_name>” --admin-enabled true**

1. You can now login to Azure Container Registry to explore “az acr” commands.

**az acr login --name “<acr\_name>”**

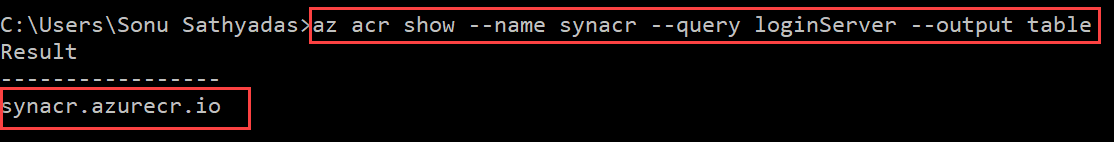
**Note:**  Make sure your docker daemon (Docker for Windows) is running.

1. Try the following command to list the image in ACR.

**az acr repository list -n <acrname> -o table**

1. Now, you can push the images to ACR. But before doing that you need to tag the images with the ACR login name. To know the ACR login name run the following command.

**az acr show --name “<acrname>” --query loginServer --output table**



1. Tag the image with the acr login name ie: synacr.azurecr.io in this case.

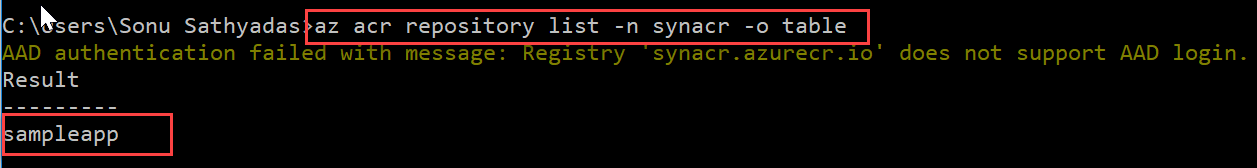
**docker tag sonusathyadas/sampleapp synacr.azurecr.io/sampleapp:1.0**

1. Upload the docker image to Azure CR using the docker push command.

**docker push synacr.azurecr.io/sampleapp:1.0**

1. List the images in ACR using the command.

**az acr repository list -n <acrname> -o table**



**Deploying Containers using Azure Container Instances Service**

Azure Container Instances offers the fastest and simplest way to run a container in Azure, without having to provision any virtual machines and without having to adopt a higher-level service. Azure Container Instances is a great solution for any scenario that can operate in isolated containers, including simple applications, task automation, and build jobs. For scenarios where you need full container orchestration, including service discovery across multiple containers, automatic scaling, and coordinated application upgrades, we recommend the Azure Container Service.

1. Login to the azure cli

**az login**

1. Get the Azure Container Registry credentials

**az acr credential show --name <synacr> --query "passwords[0].value"**

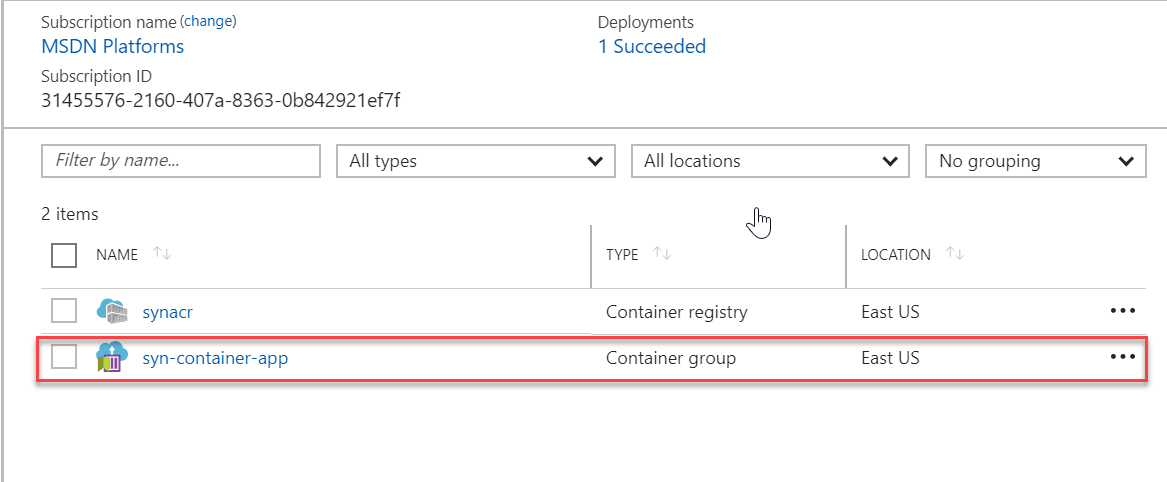
1. Create a new container instance service by running the command. The images are coming from the Azure Container Registry

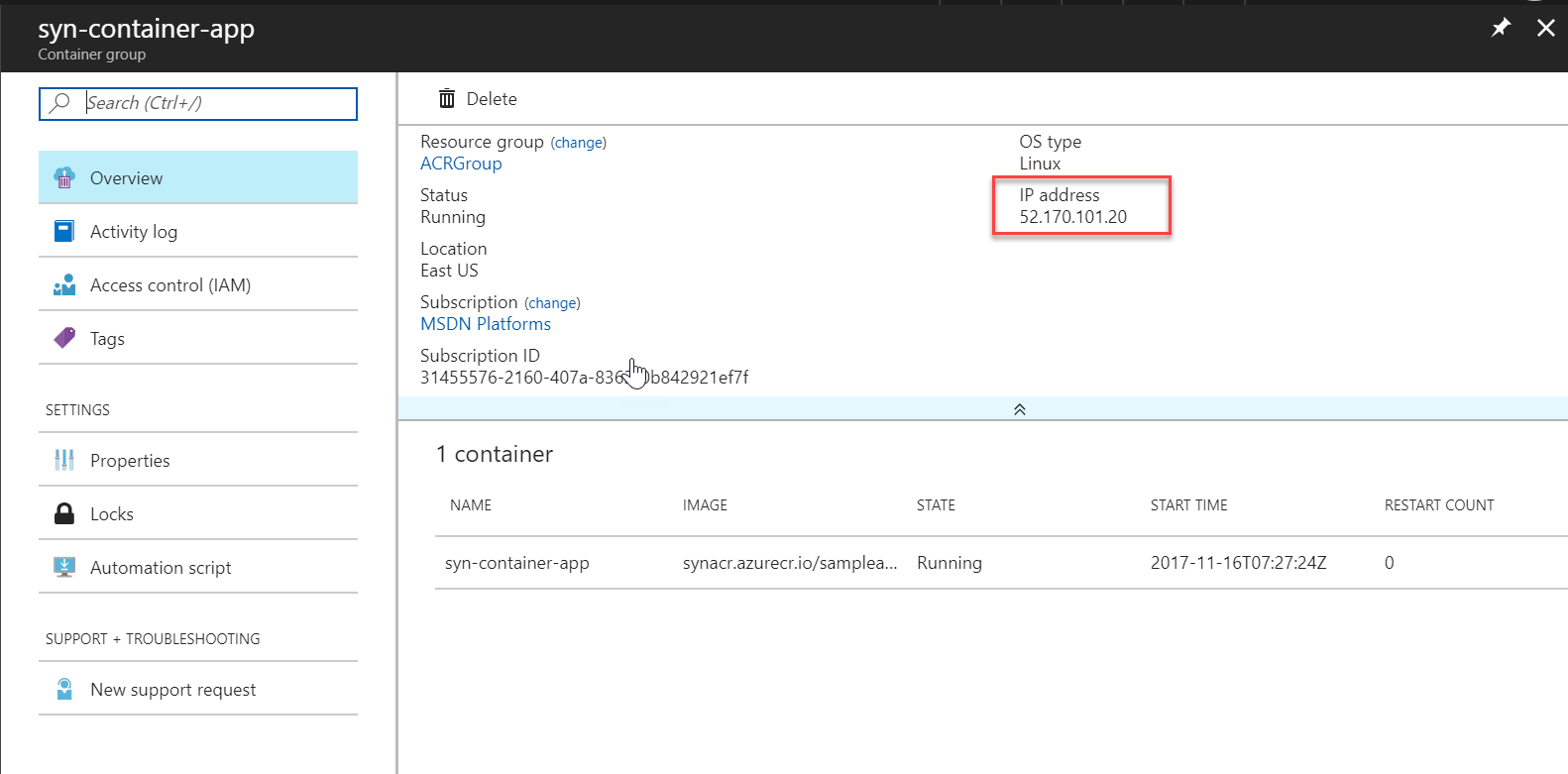
**az container create --name “<container\_app\_name>” --image “<imagename>” --cpu 1 --memory 1 --registry-password “<acr\_password> --ip-address public -g “<res\_grp>” –ports “<ports\_to\_open>”**

**example:**

**az container create --name syn\_container\_app --image synacr.azurecr.io/sampleapp:1.0 --cpu 1 --memory 1 --registry-password 8SvXOO3s=ZM2Z1gUSLXbn73FC8TC4SMB --ip-address public -g ACRGroup –ports 80**

1. Once the Container Instance creation is completed, you can see the service in the portal.

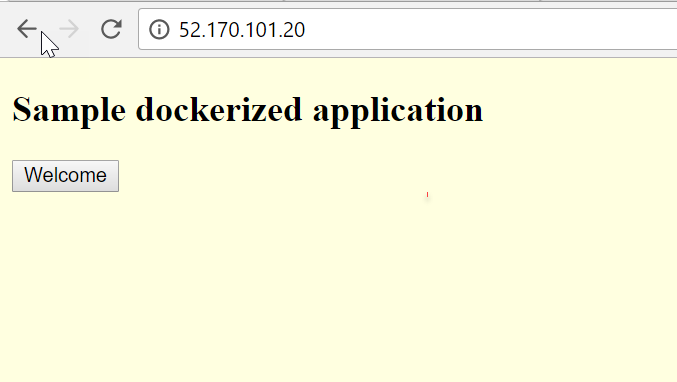




1. Check the status of the service created.

**az container show --name syn-container-app --resource-group ACRGroup --query instanceView.state**

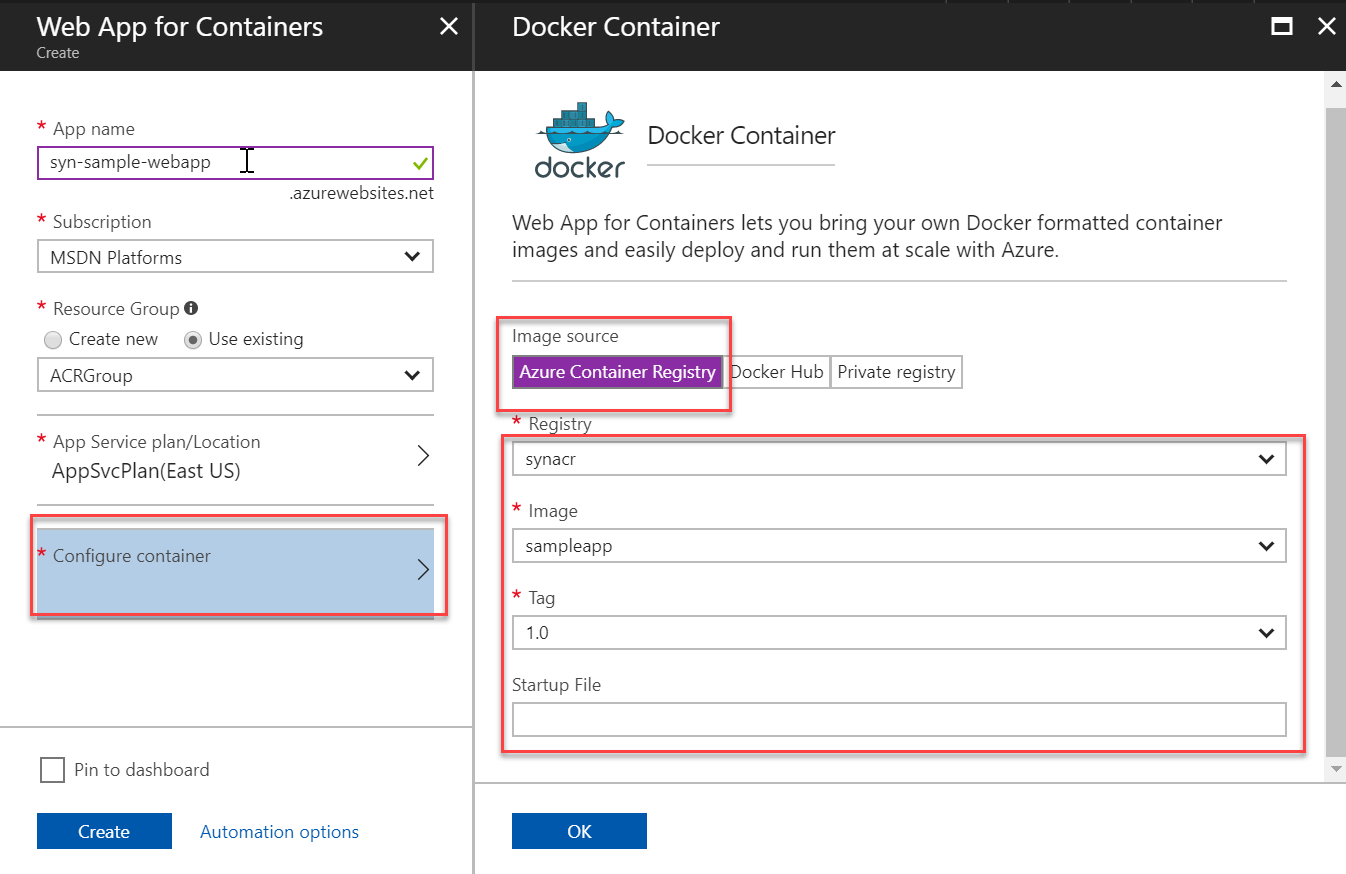
1. You can now navigate to the public IP address of the instance



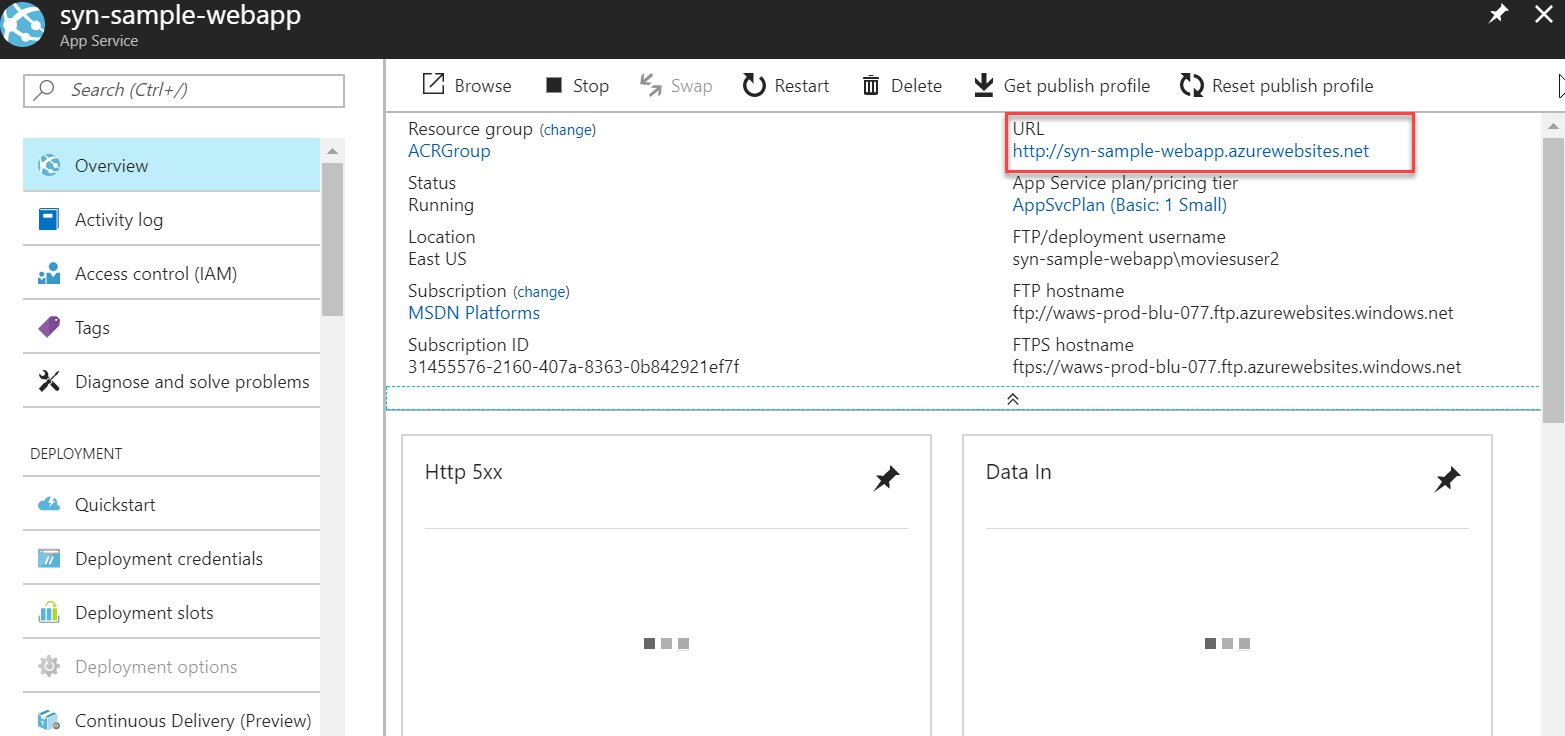
**Deploying Containers from ACR using Web App for containers Service**

You can deploy your image which is stored in the ACR using Azure Web app for Containers.

1. Create a new ‘Web app for Containers’ service from Azure Portal
2. In the blade click on ‘Configure Container’ and it opens the settings blade.



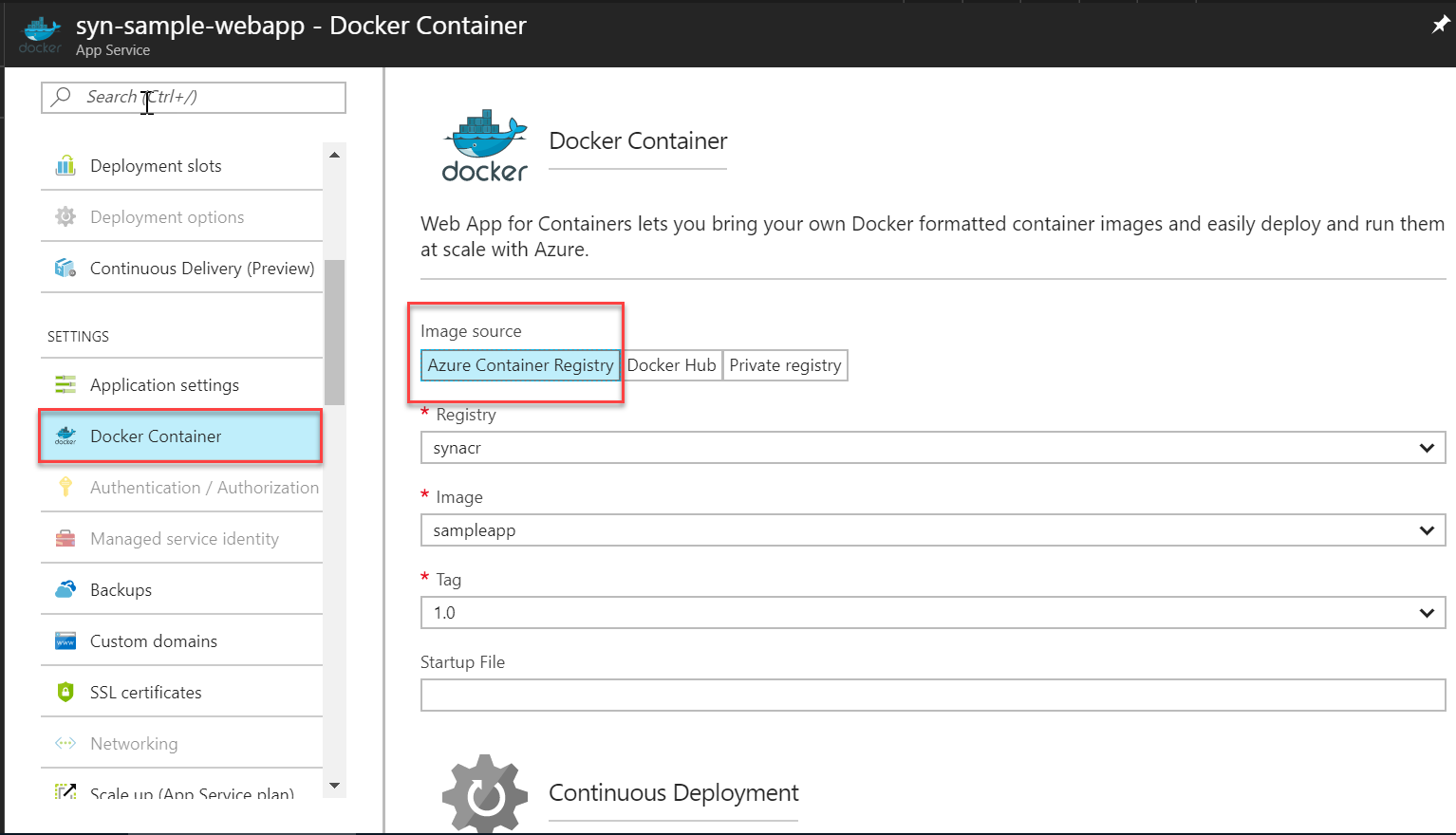
1. Specify the ACR name, image name, tag for Azure Container Registry option.
2. Click OK and Create. It starts deploying the service.
3. Once the deployment is completed, you can click on the deployment url to navigate to the application.



1. It opens the application in browser.



1. You can update the image name, or tag by selecting the ‘Docker container’ option.



**Container management using Kubernetes**

**What is Kubernetes?**

Kubernetes came from Google and donated to CNCF in 2014. It is open-source. It is written in Go/Golang. Kubernetes is used to manage the containers. It is derived from Google’s proprietary container management tools called Borg and Omega. But kubernetes is Open-source. The work Kubernetes came from the Greek work which means for ‘Helmnsman’ – the person who steers the ship. K8s stands for Kubernetes.

**What and Why?**

Containers bring scalability challenges. There could be thousands of containers in the server. You can manage the containers using Kubernetes. Developers need not to bother about on which server their codes run on. It can be managed by Kubernetes. DataCenter OS (DC OS), or Docker Swarm can also manage it. Kubernetes is a production-grade, open-source platform that orchestrates the placement (scheduling) and execution of application containers within and across computer clusters.

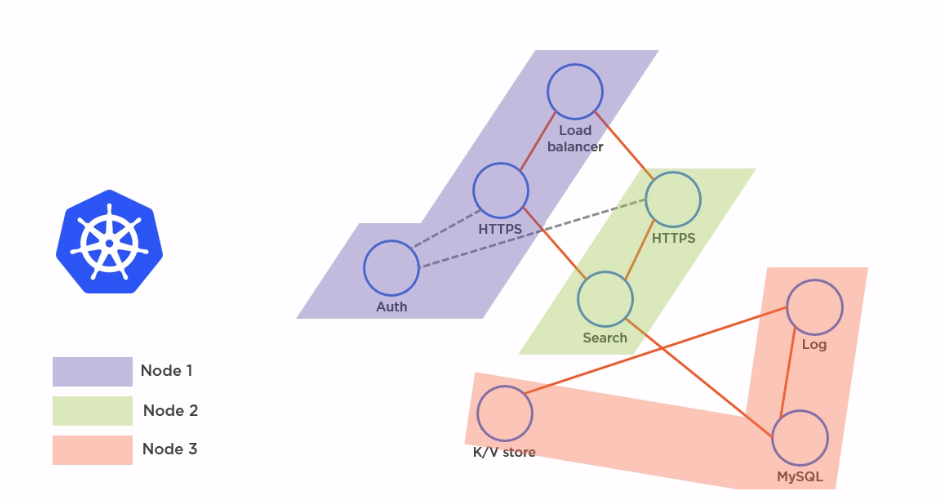
It is similar to a courier service. We package the item and label it with proper manifest. Remaining all tasks are managed by Courier services. Which flight ships it, Where to deliver, how it cares all these. Kubernetes also does the same for us. We need to package the application in containers with proper manifest. Kubernetes will manage remaining.

**Kubernetes Architecture**

* Masters
* Nodes
* Podes
* Services
* Deployments

**The Big Picture**

Kubernetes is an orchestrator for microservice apps. Microservices are individually scalable small piece of application. Each microservice can be called as a **Pod**. Kubernets manages the pods and ensure that they run in right network, right secrets. An app is collection of interrelated pods.



**Kubernetes Cluster**

Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit. The abstractions in Kubernetes allow you to deploy containerized applications to a cluster without tying them specifically to individual machines. To make use of this new model of deployment, applications need to be packaged in a way that decouples them from individual hosts: they need to be containerized. Containerized applications are more flexible and available than in past deployment models, where applications were installed directly onto specific machines as packages deeply integrated into the host. Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way.

A Kubernetes cluster consists of two types of resources:

* Master: The Master coordinates the cluster

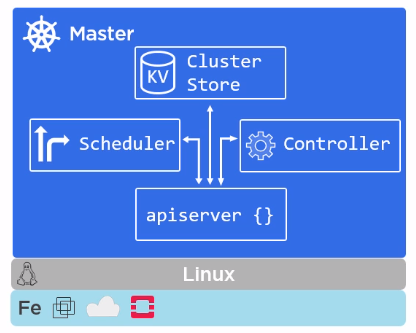
The Master is responsible for managing the cluster. The master coordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.

* Nodes: Nodes are the workers that run applications

A node is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster. Each node has a Kubelet, which is an agent for managing the node and communicating with the Kubernetes master. The node should also have tools for handling container operations, such as Docker or rkt. A Kubernetes cluster that handles production traffic should have a minimum of three nodes. The nodes communicate with the master using the Kubernetes API.

**Masters**

Master is used to manage the Nodes in a cluster. There could be one or more masters. For HA it is better to keep multiple masters in the cluster. The Kubernetes API server validates and configures data for the api objects which include pods, services, replicationcontrollers, and others. The API Server services REST operations and provides the frontend to the cluster’s shared state through which all other components interact.



* Kube-apiserver

Masters are the front-end for the control plane. They expose the REST API to connect with the cluster. Masters are only exposed to the public not the nodes. kube-apiserver exposes the Kubernetes API. It is the front-end for the Kubernetes control plane. It is designed to scale horizontally – that is, it scales by deploying more instances. No other master components expose the endpoint except apiserver. We directly interact with apiserver only.

* Cluster Store (etcd)

etcd is used as Kubernetes’ backing store. All cluster data is stored here. Always have a backup plan for etcd’s data for your Kubernetes cluster. Cluster state and configuration is stored here.

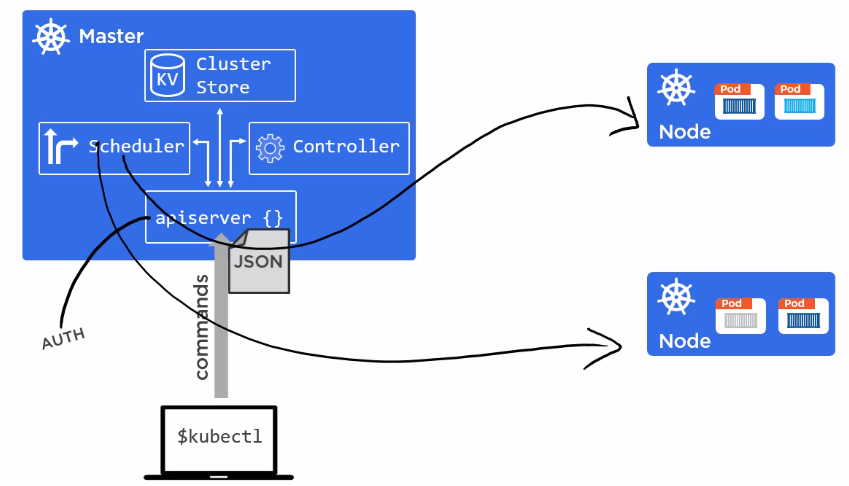
* Kube-controller-manager

It is the controller of controllers. kube-controller-manager runs controllers, which are the background threads that handle routine tasks in the cluster. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

These controllers include:

* + Node Controller: Responsible for noticing and responding when nodes go down.
  + Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
  + Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods).
  + Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces.
* Kube-scheduler

Watches apiserver for new pods. Assign works to nodes – affinity/anti-affinity, constraints, resources etc. kube-scheduler watches newly created pods that have no node assigned, and selects a node for them to run on.



**Nodes**

A node is a worker machine in Kubernetes, previously known as a minion. A node may be a VM or physical machine, depending on the cluster. Each node has the services necessary to run pods and is managed by the master components. The services on a node include Docker, kubelet and kube-proxy.

Each node contains kubelet, container runtime and the proxy.

* Kubelet

The kubelet is the primary “node agent” that runs on each node. The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the apiserver) and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn’t manage containers which were not created by Kubernetes.

Tasks done by kubelet:

* + Register node with cluster
  + Watches apiserver
  + Instantiates pods
  + Reports back to master when pod goes down
  + Exposes endpoint on port 10255
* Container engine

Does container management

* + Pulling images
  + Start/stop containers

They are pluggable: Mostly Docker or rkt

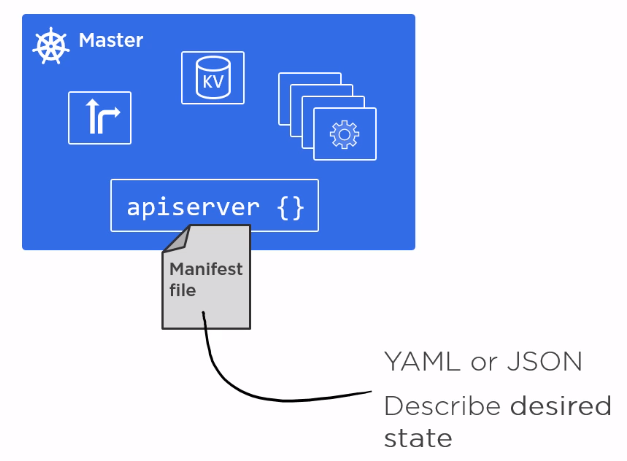
* Kube-proxy

Kubernets networking : POD IP Addresses – all containers in a pod share a single IP

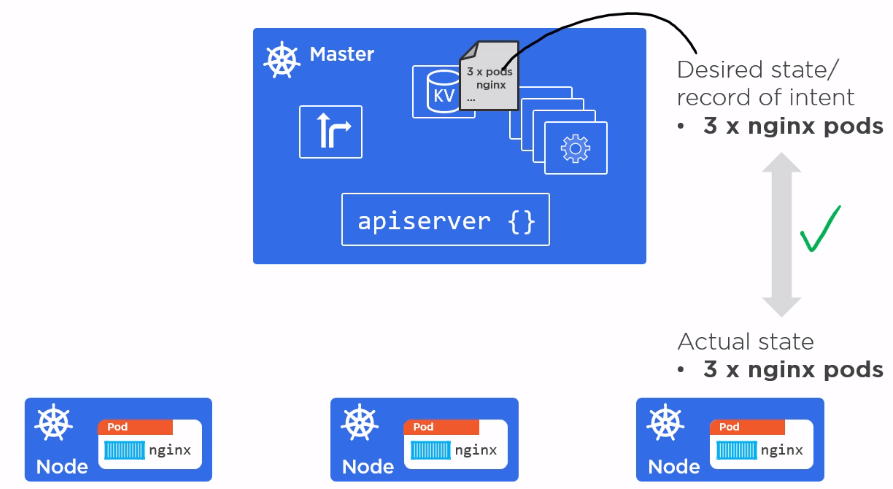
Load balances across all pods in a service.

**Declarative Model and Desired State**

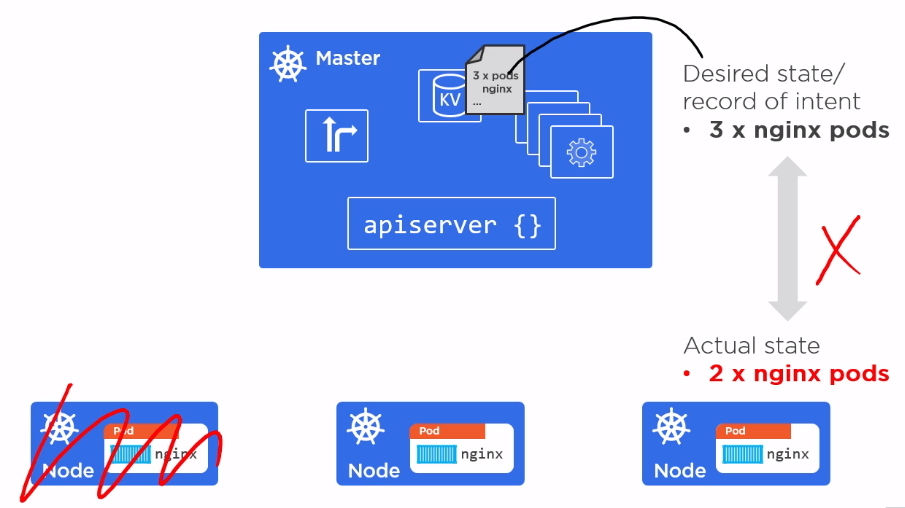
Kubernetes objects can be created, updated, and deleted by storing multiple object configuration files in a directory and using kubectl apply to recursively create and update those objects as needed.

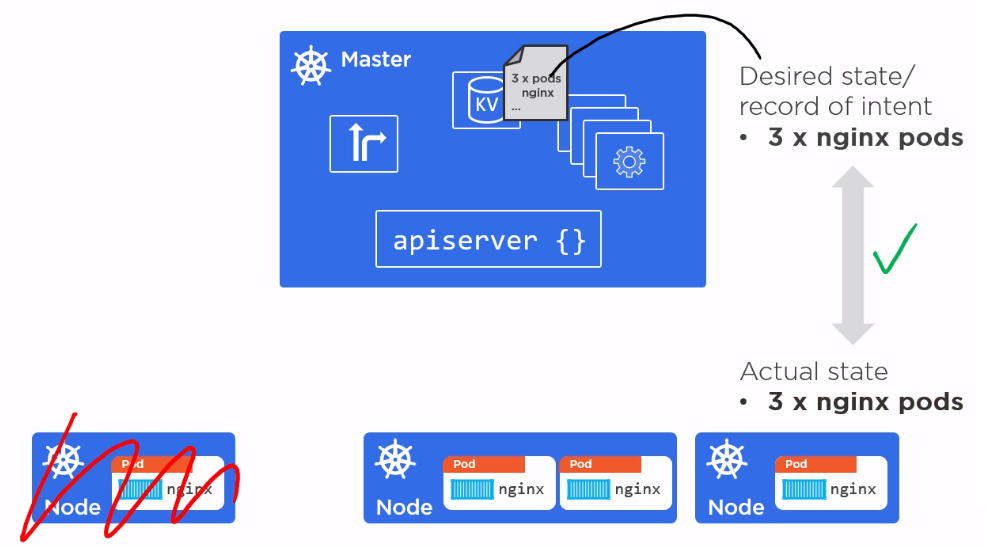


For example, When we deploy the pods we can set the desired number of pods to 3. That means always it requires 3 instances.



Whenever one node goes down it creates a new pod in any one of the healthy nodes.

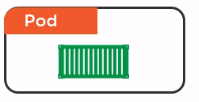
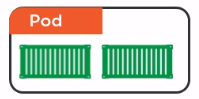




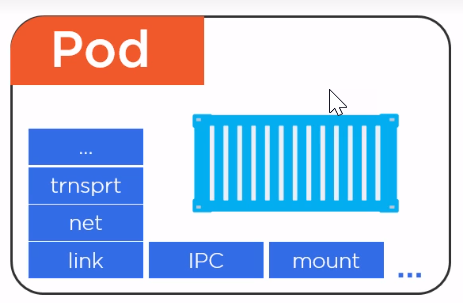
**Pods**

A Pod is the basic building block of Kubernetes–the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents a running process on your cluster. A Pod encapsulates an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run. A Pod represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources. Docker is the most common container runtime used in a Kubernetes Pod, but Pods support other container runtimes as well.

Containers cannot run directly inside the kubernets node. It runs within a pod. A single pod can have one or more containers.

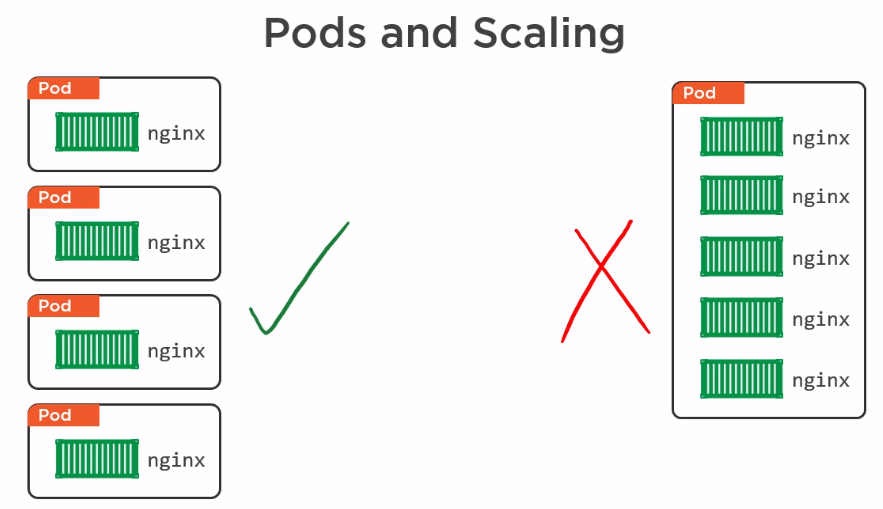
 

All the containers share the pod environment. The pod environment contains network stack, kernel namespaces and IP.



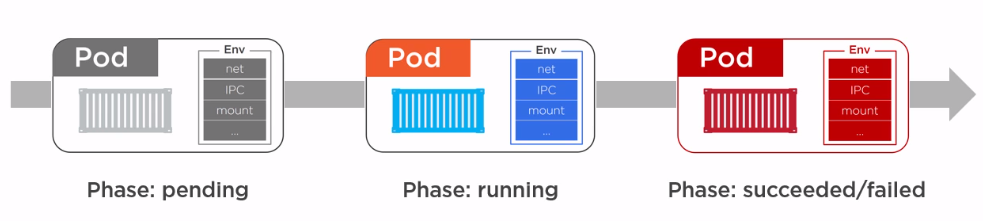
Scaling pods

You can add or remove pods according to the requirement for scaling.



Pod Lifecycle

Pods are mortal. They starts runs and dies.



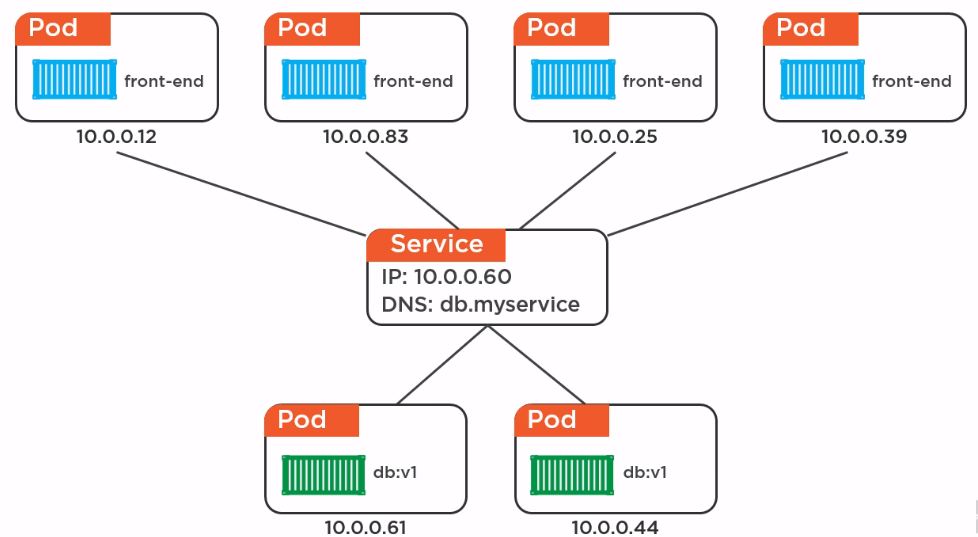
Deploying pods

We need to submit the manifest file to the apiserver in the form of YAML or JSON file (preferably YAML). The apiserver validates it and deploy them to one of the node which satisfies the requirements. The replication Controller (ReplicaSet) ensures that desired number of pods running in cluster.

**Services**

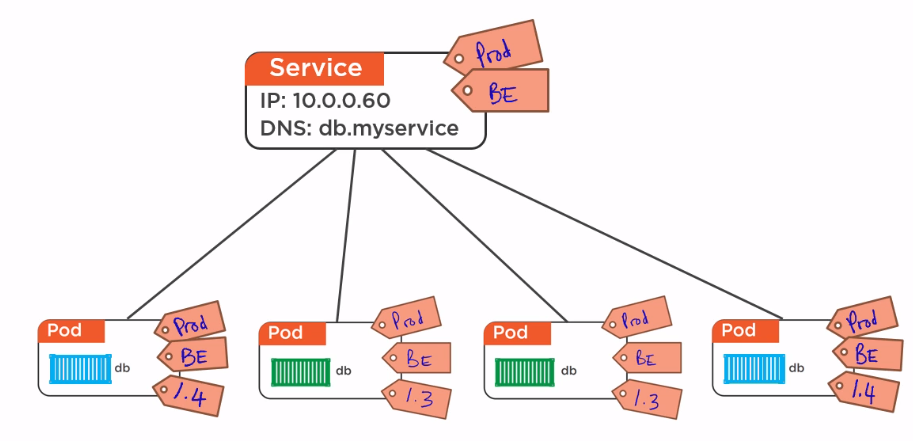
A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service. The set of Pods targeted by a Service is (usually) determined by a Label Selector.

Every pod will have some internal IP. Pods can communicate using those internal IP. But it is not reliable. For example, a Pod that runs a front-end web app need to talk to a pod that contains a db. If the pods are communicating using the internal IP it is not reliable, because the pods can go down any time and when they are recreated they gets a new IP. Also, when we do scaling pods, the number of pods will increase or decrease continually. So it is not possible to rely on pods IPs. So kuberbets uses services as an intermediate between pods. When some pods wants to talk to another pod, instead of communicating directly, they communicate through the services. Service instances will always have constant IP and DNS. So when we do scaling of pods it will not affect the communication system of pods.

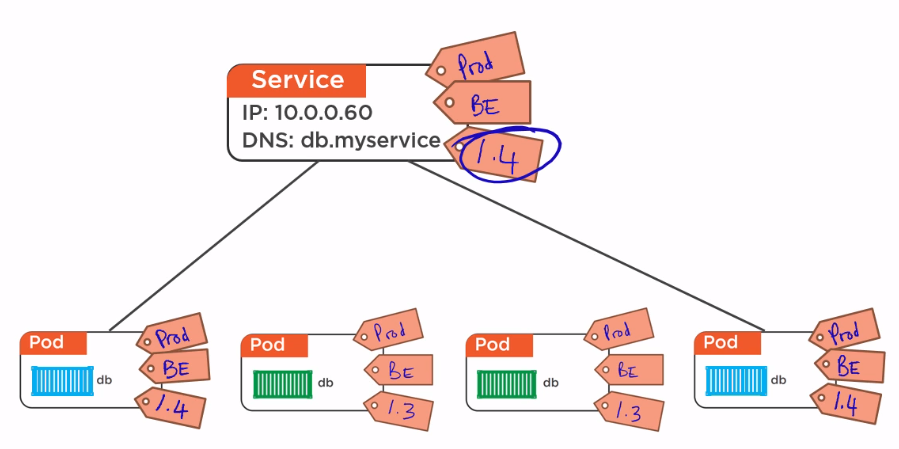


Label selector for grouping pods

Every pod can have a set of one or more labels. The same set of label is applied for services also. Any pods which satisfy the set of labels assigned to service is Load balanced under the service object. Example if a service object has two labels – BE (backend) and Prod(production). Any pods that have these labels will be load balanced by the service. Services always do the health check for pods and send traffic only to healthy pods. It performs random load balancing, but you can configure it for session affinity.



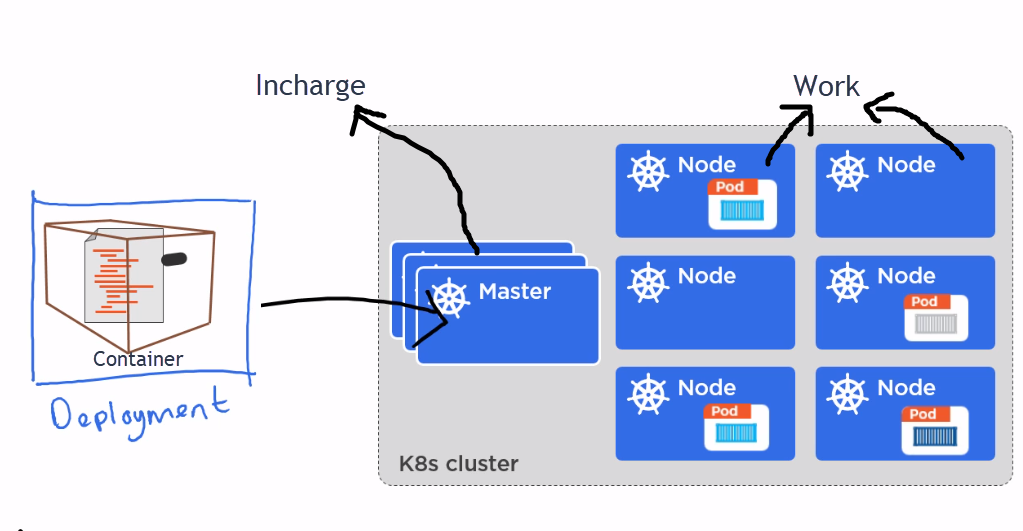
When a new label is added to the service, it selects only those pods which satisfy those pods which have the same set of labels. It drops other pods.



**Deployments**

A Deployment controller provides declarative updates for Pods and ReplicaSets. We might create a Deployment to tell Kubernetes to manage a set of replicas of that Pod — literally, a ReplicaSet — to make sure that a certain number of them are always available. You describe a desired state in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments. We define the deployments in a yaml file.

Note: Before deployments, there were replication controllers, which managed pods and ensured a certain number of them were running. Now with deployments we move to replica sets, which are basically the next-generation of replication controllers.



**Installing Kubernets**

Kubernets can be installed and configured to manage clusters that runs on local machine, Cloud or on-premise.

**Minikube**: Minikube is a tool that makes it easy to run Kubernetes locally. Minikube runs a single-node Kubernetes cluster inside a VM on your laptop for users looking to try out Kubernetes or develop with it day-to-day.

Minikube supports Kubernetes features such as:

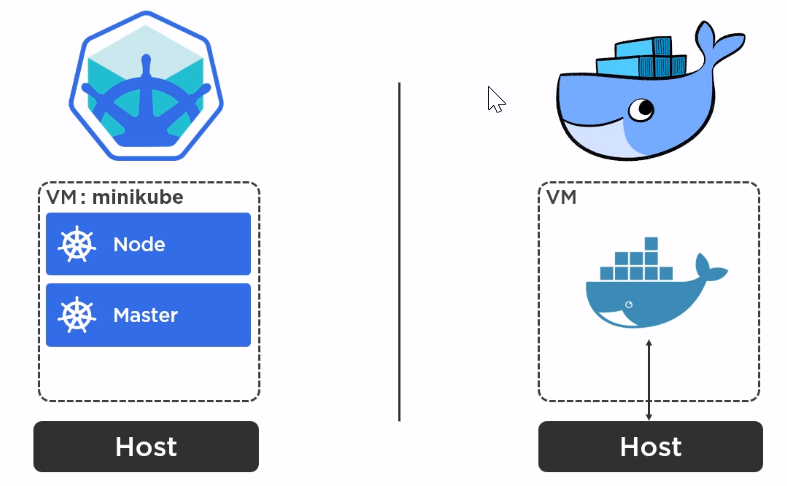
* DNS
* NodePorts
* ConfigMaps and Secrets
* Dashboards
* Container Runtime: Docker, and rkt
* Enabling CNI (Container Network Interface)
* Ingress

**Cloud Platform**: Google Container Engine (GKE), Azure Container Services (ACS)

**Minikube vs Docker For Windows/Docker for Mac**

Minikube is for Kubernets and Docker for Windows/Mac is for Docker.

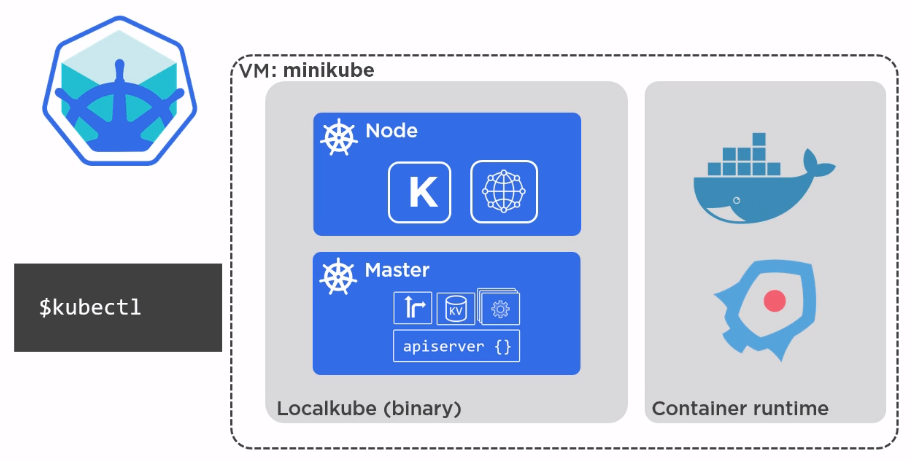
Minikube spins up a single node/master cluster inside the VM or local machine.



**Minikube architecture**

There are two main constructs in the Minikube VM. Loalkube library(binary) and Container Runtime. The localkube library is used for Master and Node. The container runtime is default to Docker and supports rkt also.

Outside the VM, we have a kubernets client called ‘kubectl’. It is used to talk to ,minikube.



**Connecting to Azure Container Services**

1. Login to Azure using Azure CLI

**az login**

1. Create azure resource group

**az group create –name “<res\_grp\_name>” –location “<location>”**

1. Create azure container services

ACS: **acs create --name="<acs\_name>" --orchestrator-type="<kubernetes>"**

**--resource-group="<res\_grp\_name>"**

**--master-count=1 --generate-ssh-keys**

AKS**:**

**az aks create --resource-group “<group-name>” --name “<aks-name>” --node-count 1 --generate-ssh-keys**

**Simple example:**

**az aks create --resource-group=samplegroup --name=bstcluster --dns-name-prefix=bytestream --generate-ssh-keys --node-count=4**

1. Show cluster details

az aks show --name myK8sCluster --resource-group myResourceGroup --output table

1. Install kubectl

**az aks install-cli**

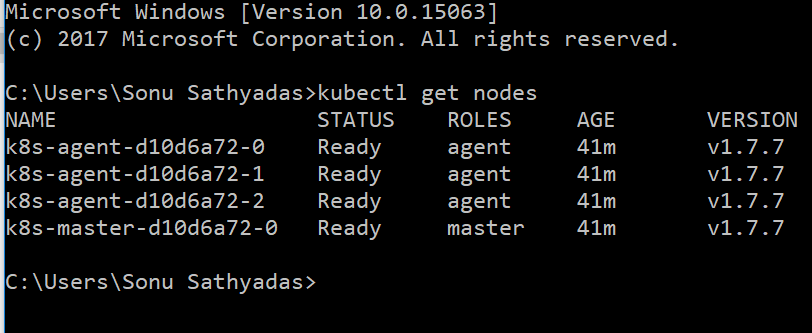
1. Once the deployment is completed. Navigate to the location where kubectl.exe is located. Download the credentials for kubectl

**ACS: az acs kubernetes get-credentials --resource-group=”<grp\_name>” --name=”<acs\_name>”**

**AKS: az aks get-credentials --resource-group=myResourceGroup --name=myK8sCluster**

1. Test whether kubctl.exe is not able to connect to Azure Container Services.

**kubectl get nodes**



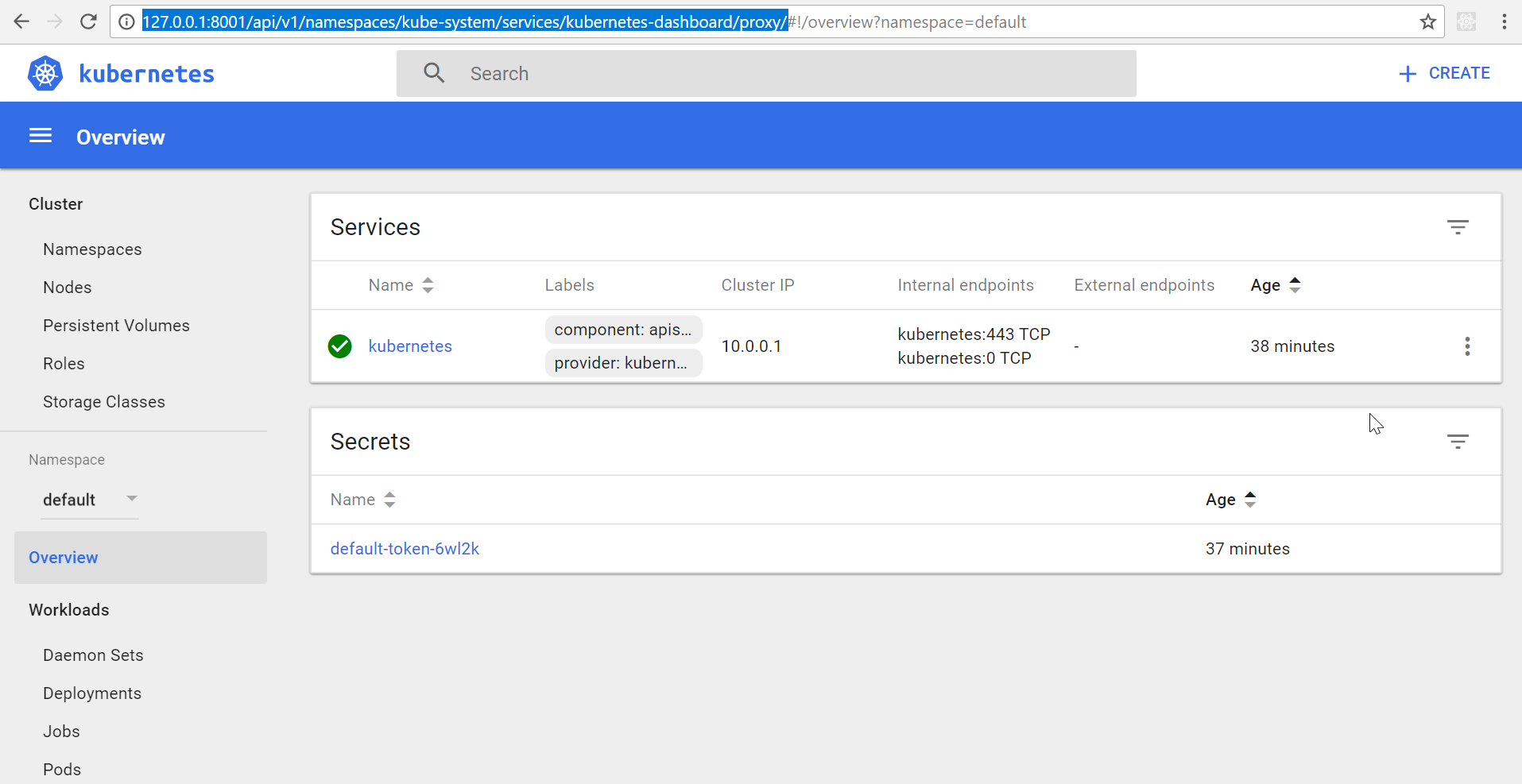
1. Open the Kubernetes Web UI.

**For ACS: az acs kubernetes browse -g “<res\_grp-name>” -n “<acs\_name>”**

**For AKS: az aks browse -g DemoGroup -n bst-kubernetescluster**

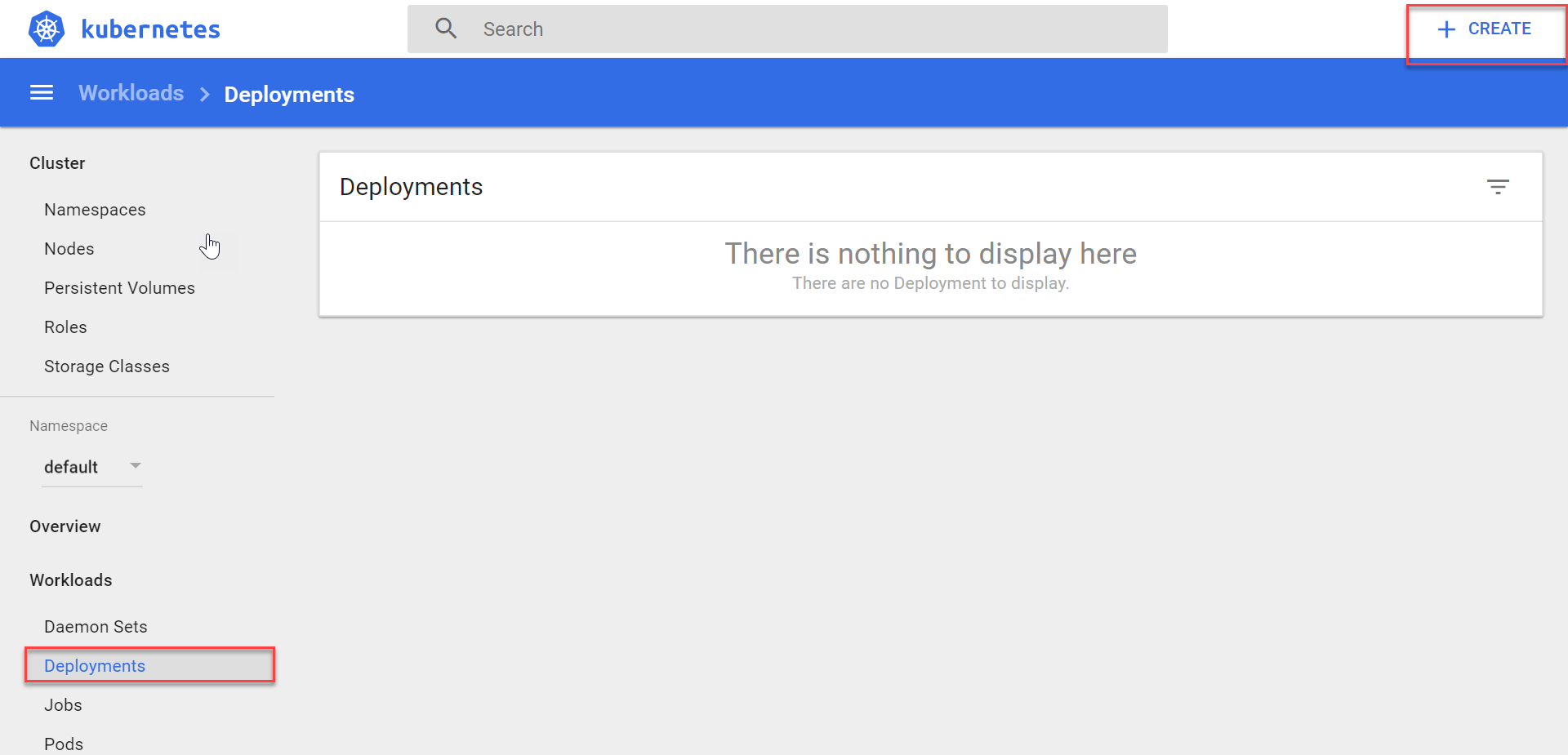
**Note**: It open the browser and navigate to the Web UI page. If the Web UI is not loading properly ie: if only white page is loading, you can add a ‘/’ at the end of the URL. Ie:

<http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/kubernetes-dashboard/proxy/>

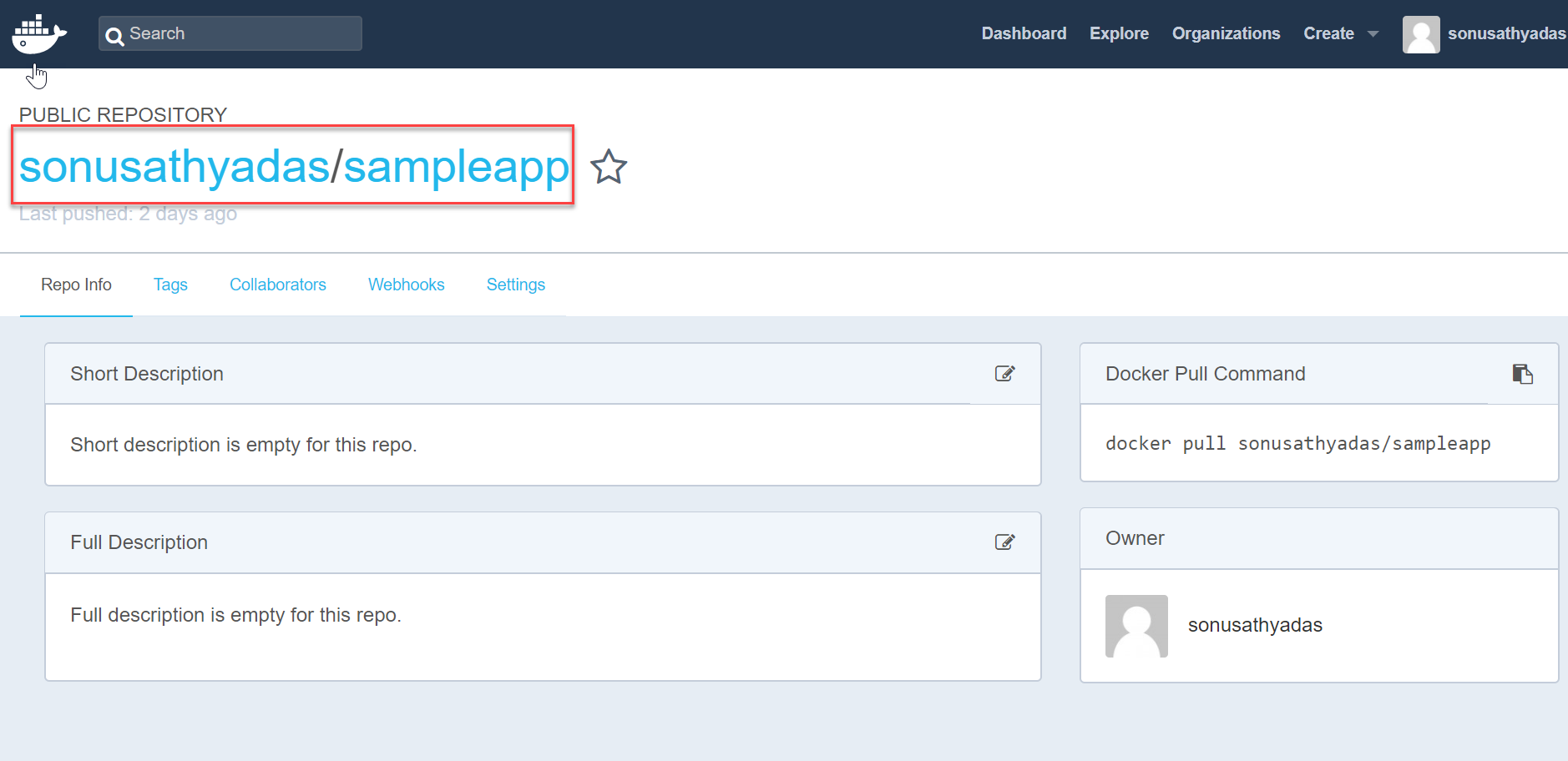


**Deploy images to Azure Container Registry from Docker Hub**

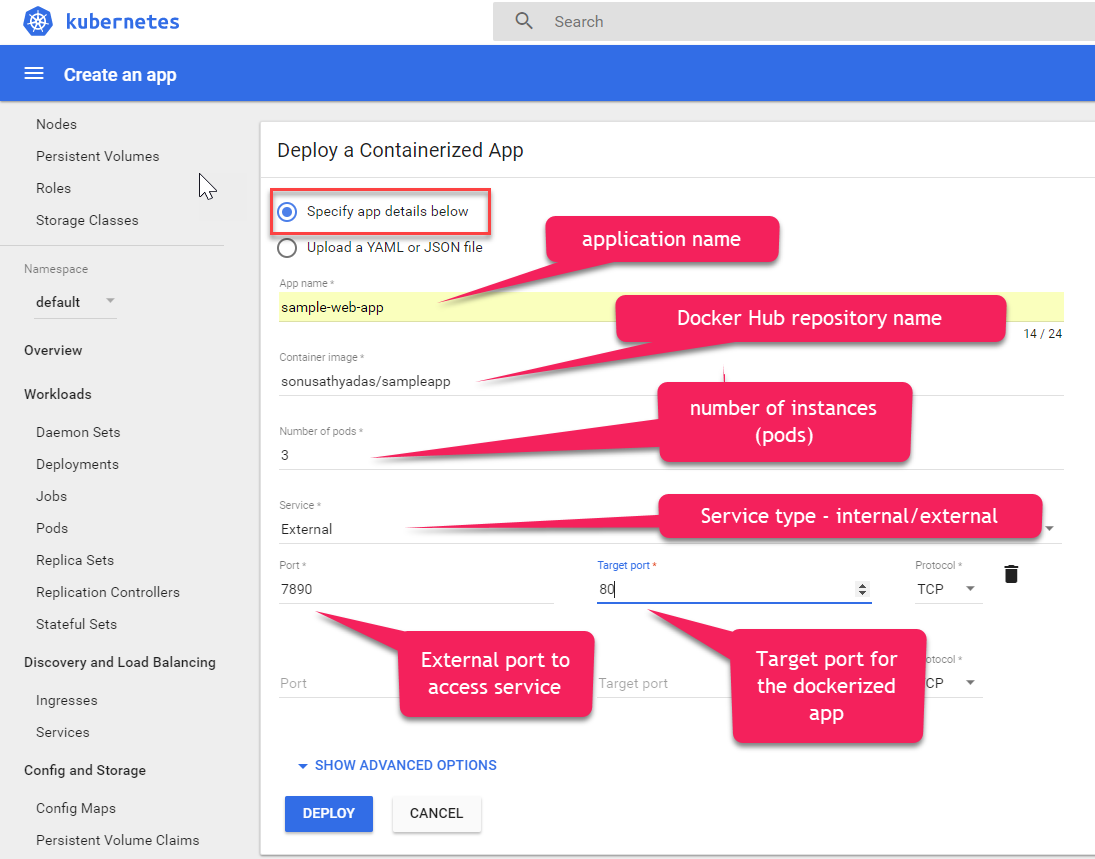
1. You can click on the Deployments link from the ACS Web UI. Then click on the Create button to start a new deployment.



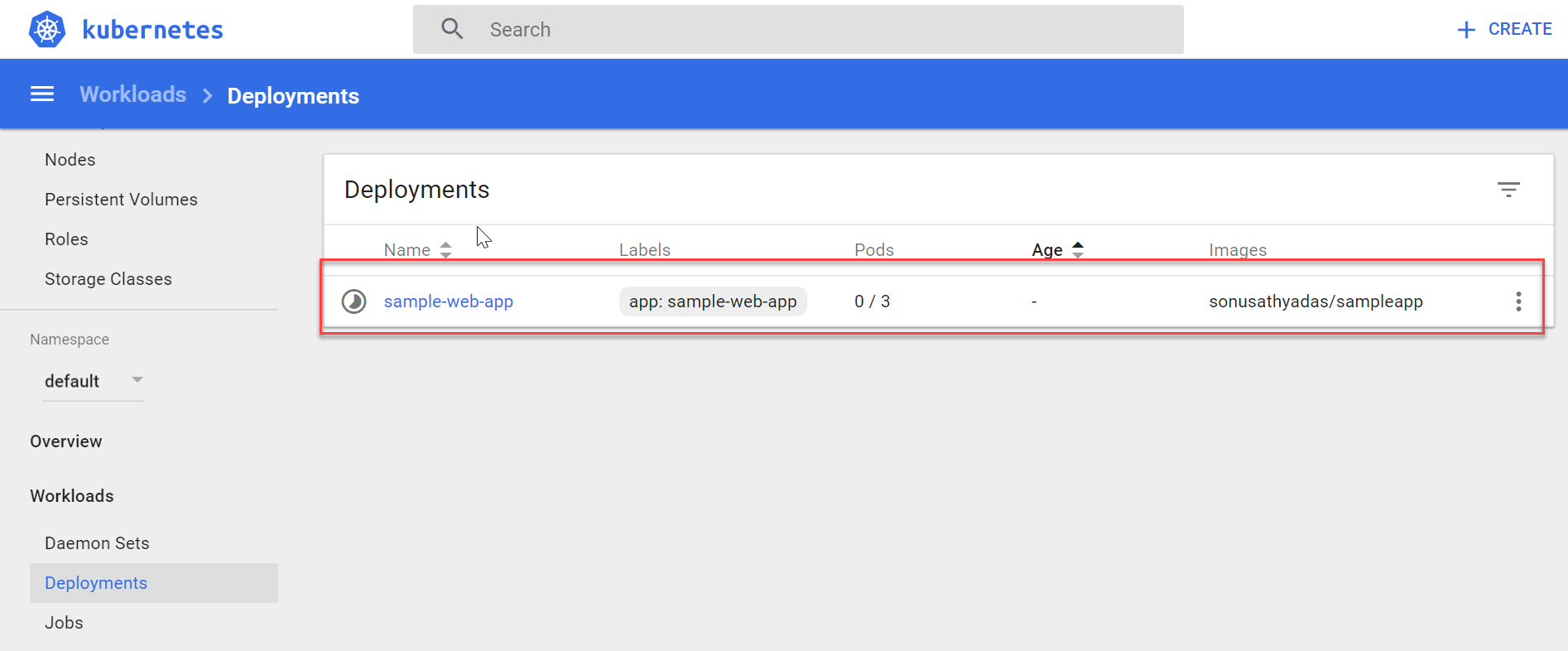
1. You can deploy the image from the Docker Hub public repository.



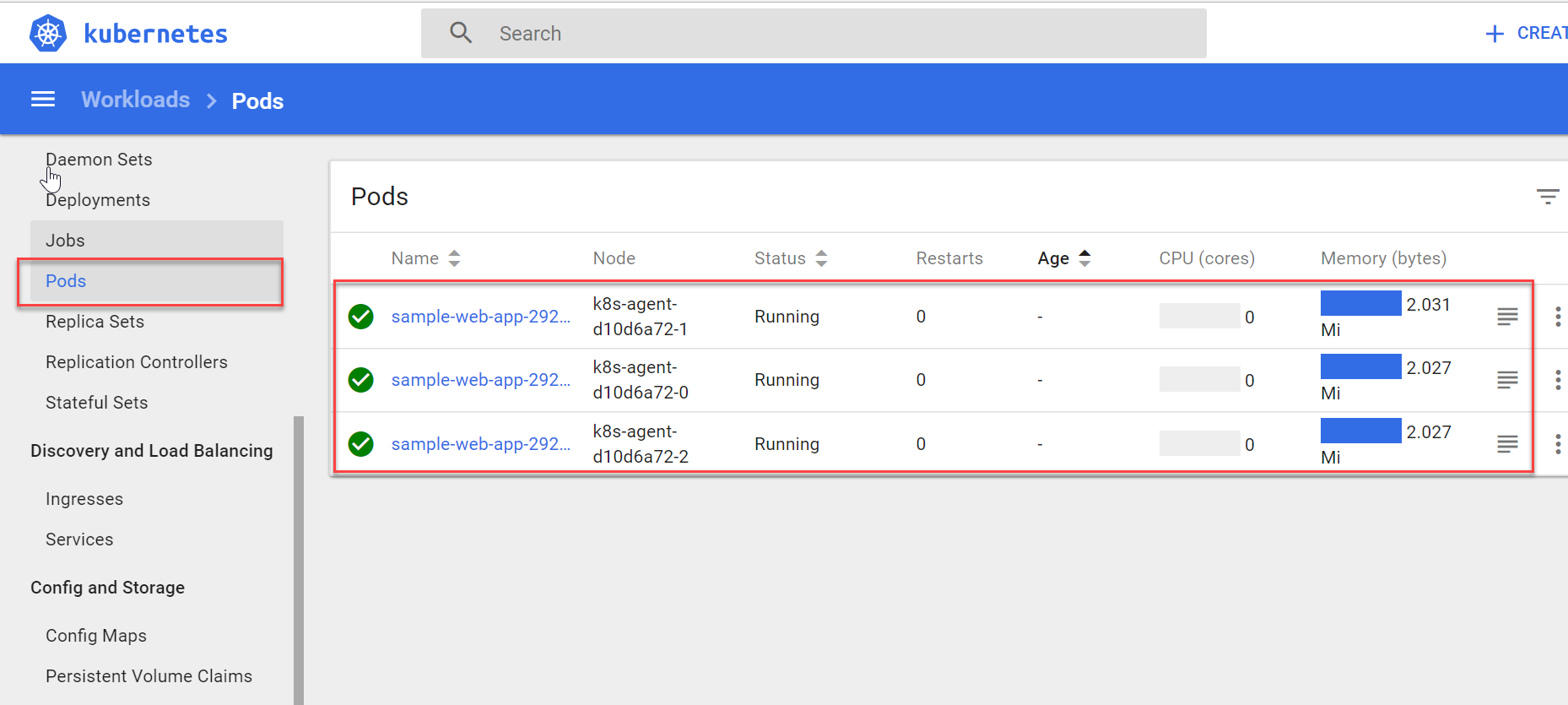
1. You can deploy directly using UI by specifying the details or using a YAML or JSON file. To start deploying an image from Docker Hub, select ‘Specify App Details below’ radio button and enter the app and image details.



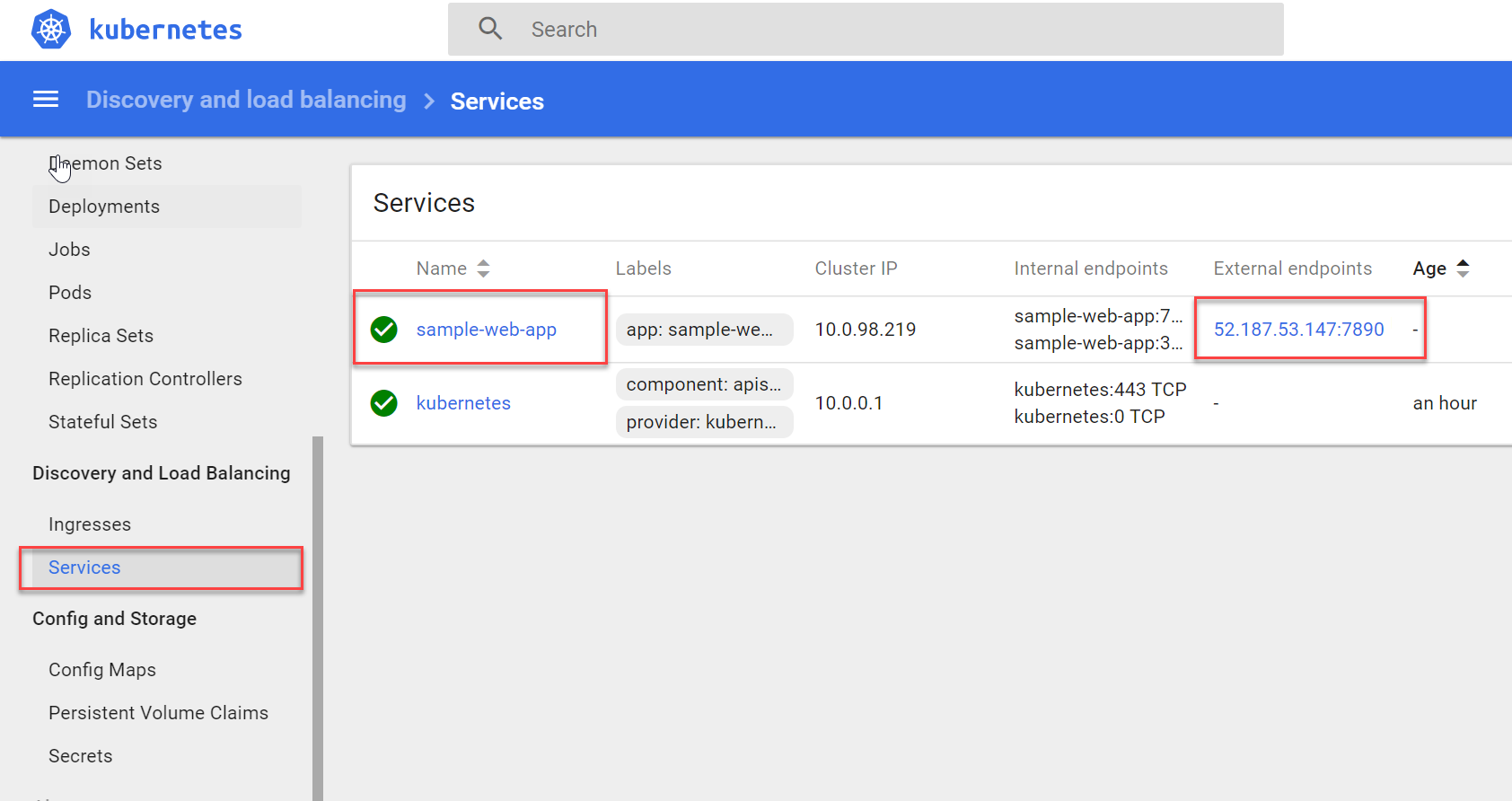
1. Click the Deploy button to complete the deployment. Once the deployment is completed you can see the deployment information in the deployments page.



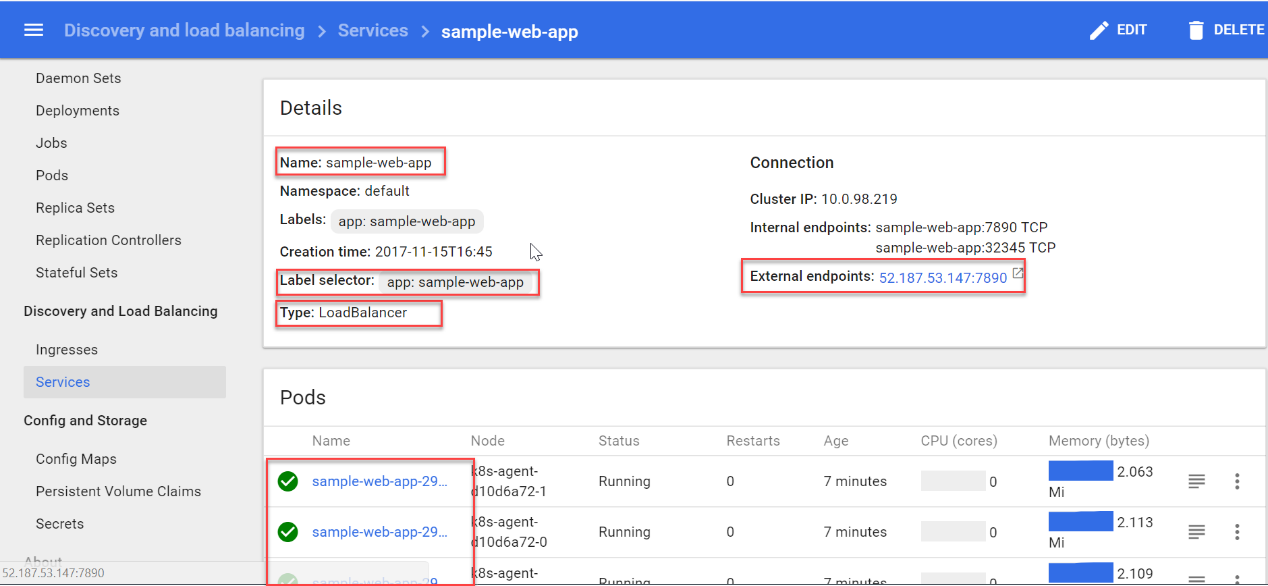
1. Select the ‘Pods’ to see the list of Pods for the application.



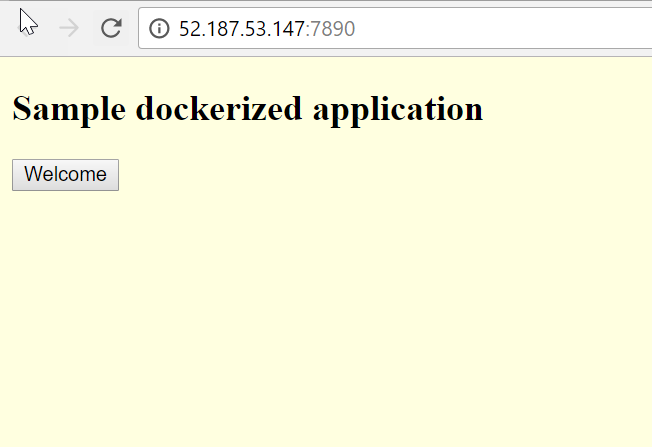
1. Click on the ‘Services’ to see the list of services. You can see the new deployed application instance in the list with an external endpoint IP.



1. Click on the service name to see the details.



1. Click on the external endpoint IP address. You can see the application running in web browser.



**Azure Container Services (AKS)**

Azure Container Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline.

AKS reduces the complexity and operational overhead of managing a Kubernetes cluster by offloading much of that responsibility to Azure. As a hosted Kubernetes service, Azure handles critical tasks like health monitoring and maintenance for you. you pay only for the agent nodes within your clusters, not for the masters. As a managed Kubernetes service, AKS provides:

* Automated Kubernetes version upgrades and patching
* Easy cluster scaling
* Self-healing hosted control plane (masters)
* Cost savings - pay only for running agent pool nodes

Azure manages the nodes in the cluster, so we don’t need to perform any tasks manually like cluster upgrades. Because azure provides these critical maintenance tasks for you AKS does not provides direct access to Cluster (via SSH) to you.

**Creating AKS using Azure CLI**

Enabling AKS preview for your Azure subscription

az provider register -n Microsoft.ContainerService

Creates Kubernetes cluster

az aks create --resource-group myResourceGroup --name myK8sCluster --node-count 1 --generate-ssh-keys

Connect to the cluster

az aks install-cli

az aks get-credentials --resource-group myResourceGroup --name myK8sCluster

kubectl get nodes

Deleting cluster

az group delete --name myResourceGroup --yes --no-wait

**Scaling AKS nodes**

az aks scale --resource-group=DemoGroup --name=bst-kubernetescluster --node-count 5

**Scaling Pods**

**Manually Scaling Pods**

kubectl scale --replicas=5 deployment/products-rest

**Autoscaling pods**

Kubernetes supports horizontal pod autoscaling to adjust the number of pods in a deployment depending on CPU utilization or other select metrics. To use the autoscaler, your pods must have CPU requests and limits defined.

The following example uses the kubectl autoscale command to autoscale the number of pods in the **products-rest** deployment. Here, if CPU utilization exceeds 50%, the autoscaler increases the pods to a maximum of 10.

**kubectl autoscale deployment azure-vote-front --cpu-percent=50 --min=3 --max=10**

To see the status of the autoscaler, run the following command

**kubectl get hpa**

**Update application for existing deployment**

You can update the image of the existing application to update the new version of the application.

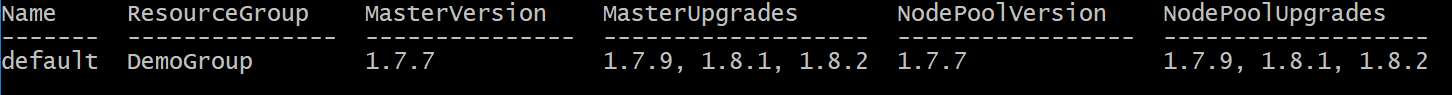
**kubectl set image deployment products-rest azure-vote-front=bstacr.azurecr.io/products-api:1.5**

**Upgrade Kubernetes in Azure Container Service (AKS)**

**Get cluster versions**

az aks get-versions --name <aks-name> \

--resource-group <group-name> --output table



**Upgrade cluster**

az aks upgrade --name <aks-name> --resource-group <group-name> --kubernetes-version <desired version>

Eg: az aks upgrade --name bst-cluster --resource-group DemoGroup --kubernetes-version 1.8.2