Azure Bootcamp 2018

Contents

[Prereqs 1](#_Toc512076087)

[Getting started 1](#_Toc512076088)

[Workshop 1 – Inventory “microservice” on Docker 4](#_Toc512076089)

[Run a SQL Server container 4](#_Toc512076090)

[Get an API going 4](#_Toc512076091)

[Add Docker Support 5](#_Toc512076092)

[Take a copy of the DB and reset 7](#_Toc512076093)

[Add docker-compose support 8](#_Toc512076094)

[Workshop 2 – Push the Inventory microservice to AKS 8](#_Toc512076095)

[Deploy an AKS cluster 8](#_Toc512076096)

[Set up your Docker Hub account 9](#_Toc512076097)

[Make a release build of the microservice API 9](#_Toc512076098)

[Push your images to Docker Hub 10](#_Toc512076099)

[Set up the KubeCtl tool 11](#_Toc512076100)

[Deploy our microservice to AKS 12](#_Toc512076101)

[Workshop 3 – Make your own Container Registry (ACR) 13](#_Toc512076102)

# Prereqs

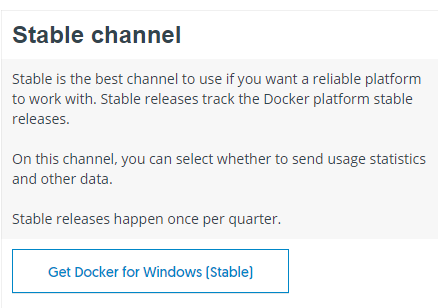
Don’t worry if you don’t have all of these yet!

* Docker for Windows
* SQL Server 2017 Docker image.
* Azure CLI for PowerShell.
* SQL Server Management Studio.
* Postman.

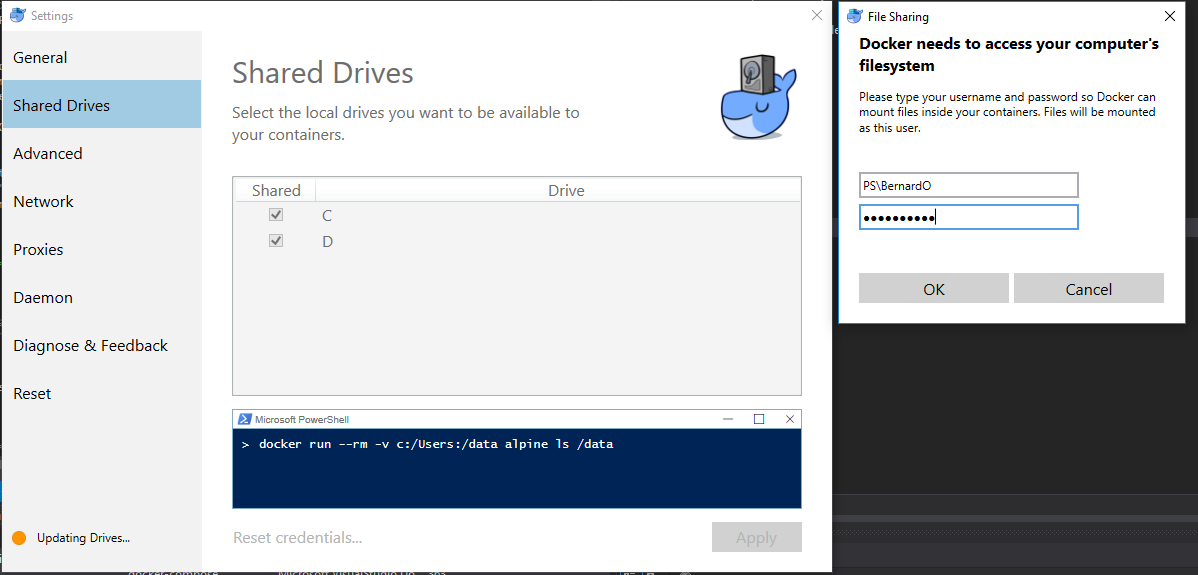
# Getting started

You can do this while Bernard is talking if necessary.

* $ git clone https://github.com/bernardoleary/Azure-Bootcamp-2018.git
* Install Docker for Windows (D4W) – shouldn’t take too long: <https://docs.docker.com/docker-for-windows/install/>

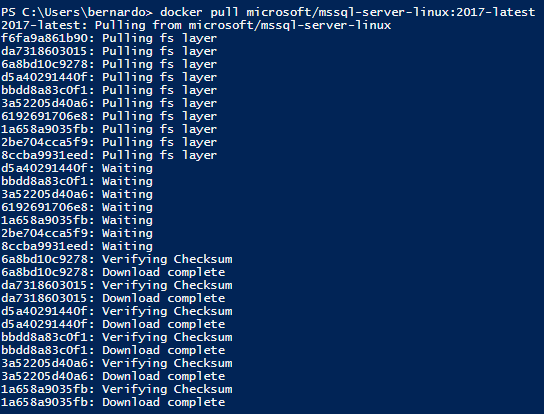


Make sure that you have enabled shared access across your drives:



* Get the SQL Server 2017 image – might take a little while:

$ docker pull microsoft/mssql-server-linux:2017-latest

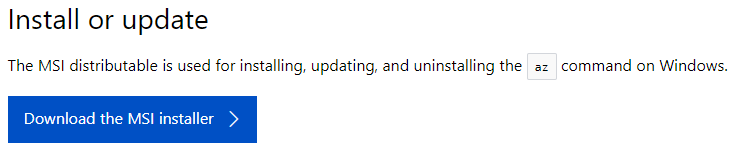


* Confirm download:

$ docker images



* Install the Azure CLI: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli-windows?view=azure-cli-latest>



* Make sure you have Visual Studio Tool for Docker installed: <https://docs.microsoft.com/en-us/dotnet/standard/containerized-lifecycle-architecture/design-develop-containerized-apps/visual-studio-tools-for-docker>
* Make sure you have SSMS installed: <https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017>
* Make sure you have Postman (or similar) installed: <https://www.getpostman.com/>

# Workshop 1 – Inventory “microservice” on Docker

We’re going to build an extremely simple microservice (hence the quotation marks) using Docker for Windows, Docker Compose and SQL Server 2017 on Docker. Then we’re going to populate it using Postman.

## Run a SQL Server container

1. Run the SQL image – create a container (change the port if you have a default SQL Server instance running on your machine already):

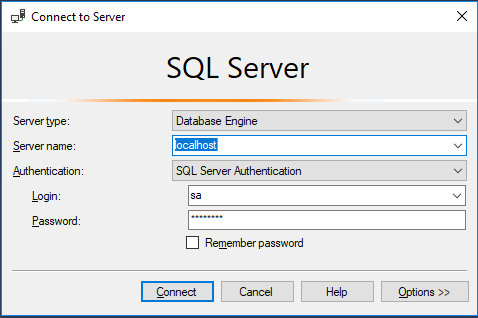
$ docker run -e "ACCEPT\_EULA=Y" -e "MSSQL\_SA\_PASSWORD=P@ssw0rd" -p 1433:1433 --name sql1 -d microsoft/mssql-server-linux:2017-latest

1. Verify the container is running:

$ docker ps

$ docker container ls

1. Log in to the DB server:



1. Run the SQL script to create the InventoryDB: <https://gist.github.com/bernardoleary/faf1e515d40f38db7fcf2aeb29bb4a3b>

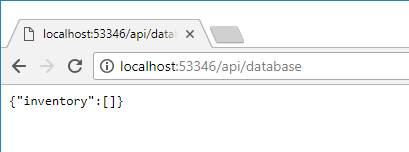
Populate the InventoryDB with some stuff manually if you like.

## Get an API going

1. Get the code for the demo solution:

$ git clone

1. Take a look at Startup.cs – especially if you have not used dotnet core much in the past, note the baked-in IoC management (IServiceCollection). Also note the database connection details.
2. Run the solution and browse to the DB controller:

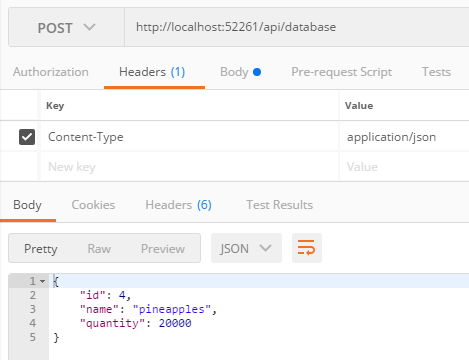


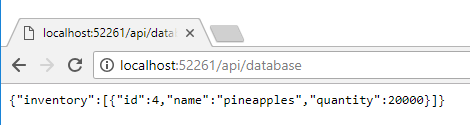
1. Now we can populate the DB using Postman:

{"id":4,"name":"pineapples","quantity":20000}

<http://localhost:53346/api/database>

Content-Type application/json

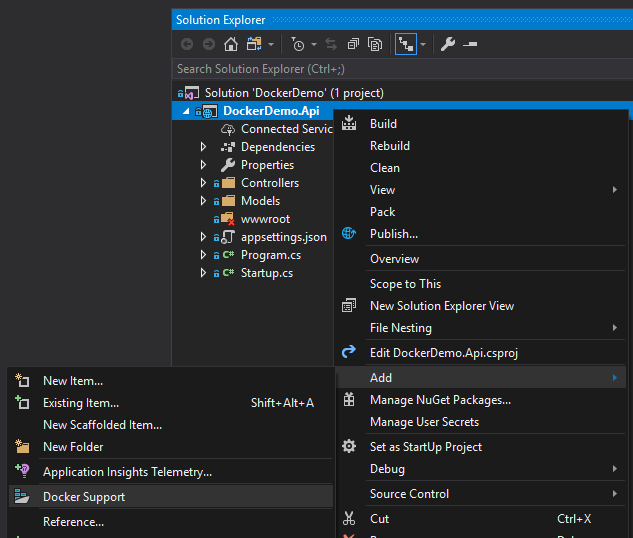




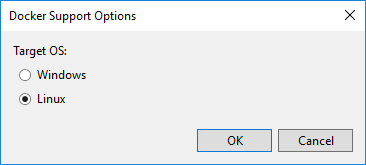
1. Note that this solution is not running as a container – it is on IIS Express – but the database is on a container.

## Add Docker Support

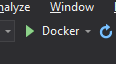
1. Now add “Docker Support” to the dotnet core project:



Target Linux (because our SQL Server container is running on Linux also):



This will add another project to your solution called “docker-compose”. Note that your debug options have changed to “Docker” only (“IISExpress” is gone). Also a file called “Dockerfile” has been added to your dotnet core project.



1. Run the Docker-ised solution in debug mode.

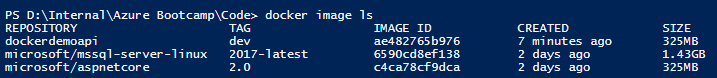
What happens when you try to reach [http://localhost:<port>/api/values](http://localhost:%3cport%3e/api/values)?

What happens when you try to reach [http://localhost: <port>/api/database](http://localhost:32769/api/database)?

1. Go to your PowerShell prompt and run:

$ docker image ls

This will list all images – note that there are two new images there now – one for your application and one for microsoft/aspnetcore:



Docker has downloaded the microsoft/aspnetcore image because it is required to run our dotnet core web application – this is specified in the Dockerfile (take a look).

1. Now list your running containers:

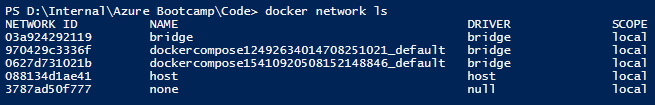
$ docker image ls

You should see there are two containers running – one is out SQL DB, the other is our dotnet core web application:



Why can the web-app not see the DB? Remember that Docker is like a miniature datacentre running on your PC – complete with networks. You can inspect the networks and see what containers are running on them:

$ docker network ls



$ docker network inspect <network id>

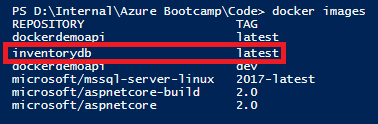
Unless it is told to, Docker will not put containers on the same network – hence why our containers aren’t able to see each other.

## Take a copy of the DB and reset

1. Because Docker images are stateless, when we create a container from one it will spawn from scratch (a blank DB server) – so to avoid having to recreate the DB, we take a copy of the running container and commit it as an image – like this (get the container ID by running a “$ docker ps”):



$ docker image ls



1. Stop debugging and clear out all containers:

$ docker stop $(docker ps -a -q)

$ docker rm $(docker ps -a -q)

Check that no containers are running:

$ docker ps

## Add docker-compose support

1. Open the file named docker-compose.yml. Note that only our web-app’s container is listed. We need to start the inventory container at the same time using docker-compose so that they containers are on the same network. Add the highlighted lines to the docker file:

version: '3'

services:

dockerdemo.api:

image: dockerdemoapi

build:

context: .

dockerfile: DockerDemo.Api/Dockerfile

dockerdemodb:

image: inventorydb

1. Open Startup.cs and change the following line of code:

Environment.GetEnvironmentVariable("SQLSERVER\_HOST") ?? " dockerdemodb";

1. Run the solution in debug mode again and try to reach the “/api/database” endpoint. Run a “$ docker ps” to see that you have your two containers running. The API container can now look up the hostname dockerdemodb and get back that container’s IP address. Same goes for any other container we run as part of this solution using docker-compose.

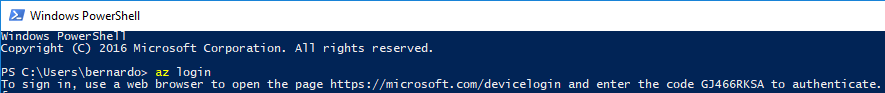
# Workshop 2 – Push the Inventory microservice to AKS

We’re going to make a Docker Hub account, push our Docker images to Docker Hub and finally we’ll launch them to managed Kubernetes on Azure (AKS).

## Deploy an AKS cluster

1. Open PowerShell and login to Azure:

$ az login



Set our subscription ID:

$ az account set --subscription “<subscription id>”

1. Create a new Resource Group for this workshop, which we’ll put the K8s (K8s is short for Kubernetes) cluster on – note, the Resource Group must be in eastus or another region that support the AKS preview:

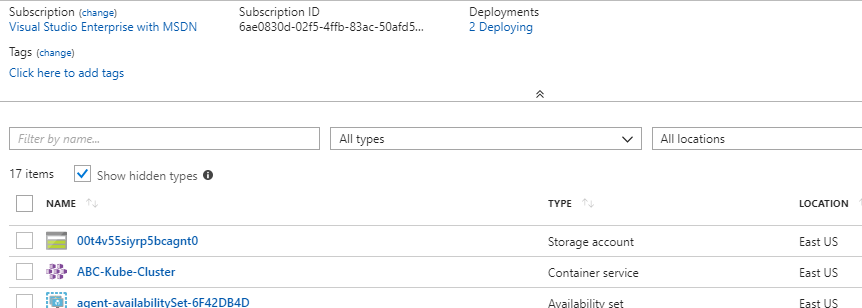
$ az group create -l eastus -n ABC- K8s

1. Create our K8s cluster – this will take about 10-to-15 minutes:

$ az acs create -n ABC-Kube-Cluster -d ABC-Kube -g ABC-K8s --generate-ssh-keys --orchestrator-type kubernetes --agent-count 1 --agent-vm-size Standard\_D1\_v2



Once you see the “Running ..” prompt you should be able to see the cluster deploying on the Azure portal:



## Set up your Docker Hub account

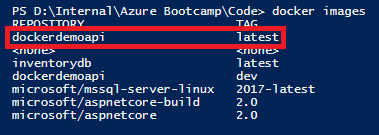
1. Go to <https://hub.docker.com> and sign-up – too easy!

## Make a release build of the microservice API

1. Go back to VS and change your build mode to “Release” and re-run our microservice:



Note that once the build/run is completed that there is a new tag in our dockerdemoapi image tagged “latest”. Previously we only had one image in the repo, with a “dev” tag:



1. Test that the microservice works by changing our docker-compose.yml and docker-compose.override.yml file subtly to match our new images and running “$ docker-compose up” and/or running the release build in VS:

**docker-compose.override.yml**

version: '3'

services:

dockerdemoapi:

environment:

- ASPNETCORE\_ENVIRONMENT=Development

ports:

- "80"

**docker-compose.yml**

version: '3'

services:

dockerdemoapi:

image: <your docker hub namespace>/dockerdemoapi

build:

context: .

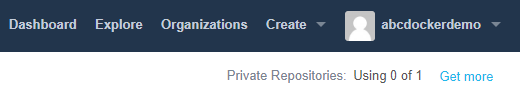
dockerfile: DockerDemo.Api/Dockerfile

dockerdemodb:

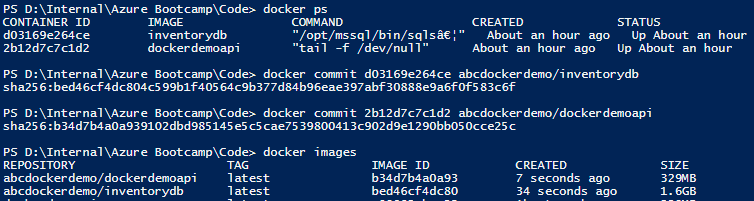
image: <your docker hub namespace>/inventorydb

1. Using the Docker “ps” and “commit” commands, make copies of your images that are prefixed with the namespace that you have created for your Docker Hub profile.

Docker Hub namespace – my one is “abcdockerdemo”:



Commit command to make images that are prefixed with your Docker Hub namespace:



## Push your images to Docker Hub

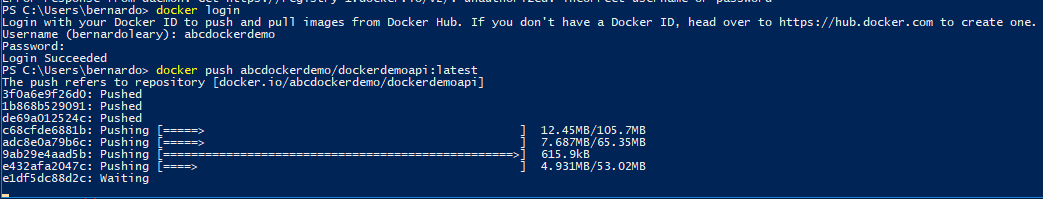
1. Login to Docker:

$ docker login

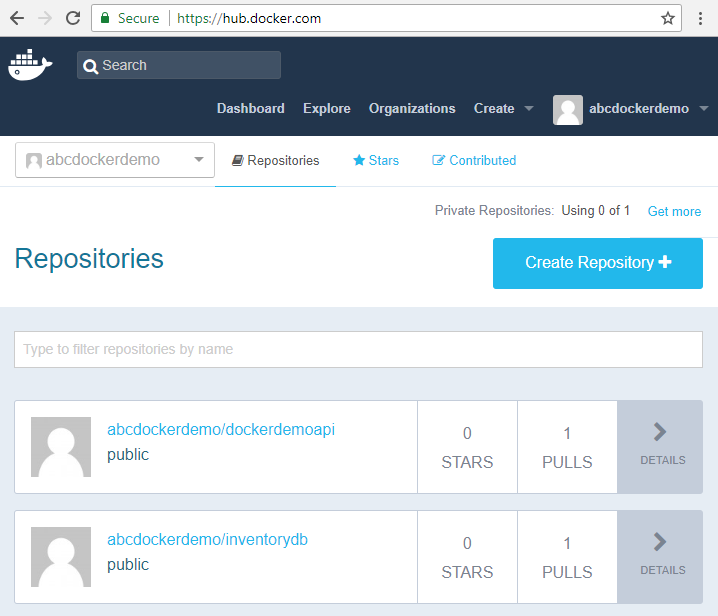
1. Push your images to Docker Hub:

$ docker push <your docker hub namespace>/dockerdemoapi:latest

$ docker push <your docker hub namespace>/inventorydb:latest



You should be able to see your new repos on Docker Hub when you have finished:

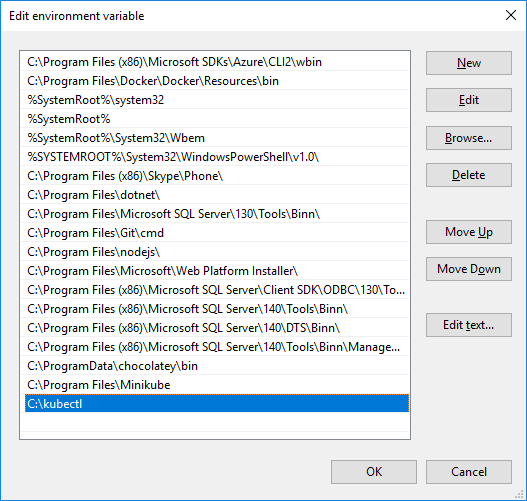


## Set up the KubeCtl tool

1. Make a folder called “kubectl” under your “C:\” then run:

$ az acs kubernetes install-cli --install-location=C:\kubectl\kubectl.exe

1. Put kubectl on your PATH:



1. Get the key that will enable you to interact directly with AKS from the command line:

$ az acs kubernetes get-credentials --resource-group=ABC-K8s --name=ABC-Kube-Cluster

You should see the output as follows:

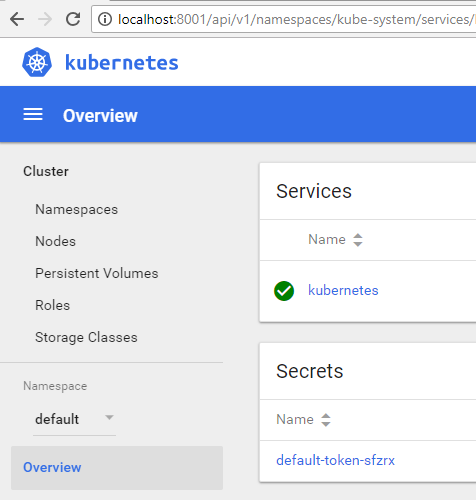
Merged "k8s-kubemgmt" as current context in C:\Users\bernardo\.kube\config

1. Start the kubectl proxy:

$ kubectl proxy

You should see output as follows: “Starting to serve on 127.0.0.1:8001”

Browse to <http://localhost:8001/ui>



## Deploy our microservice to AKS

1. Add a deployment file that we will use to upload to Kubernetes – I called mine dockerdemo.yml:

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: abcdockerdemo-deployment

spec:

replicas: 3

template:

metadata:

labels:

app: abcdockerdemo

spec:

containers:

- name: dockerdemoapi

image: <your docker hub namespace>/dockerdemoapi:latest

ports:

- containerPort: 80

- name: inventorydb

image: <your docker hub namespace>/inventorydb:latest

ports:

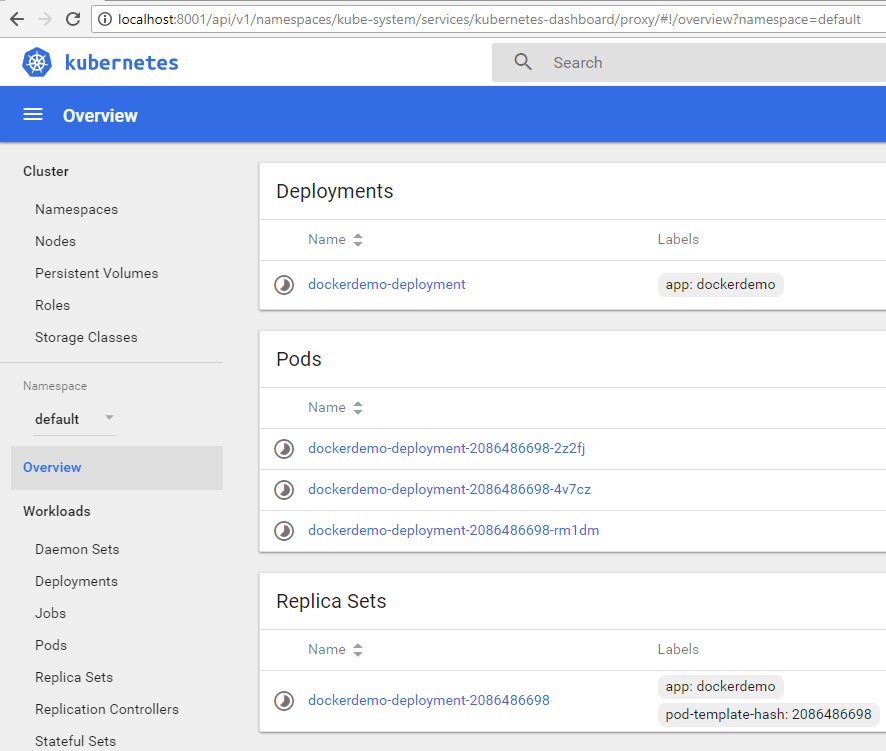
- containerPort: 1433

1. Deploy to AKS:

$ kubectrl apply -f dockerdemo.yml

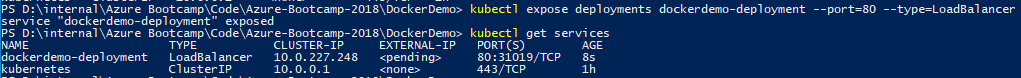


You should be able to see the Kubernetes pods, etc, deploying via the UI – the images are being pulled from Docker Hub:



1. Request a load balancer setup so that we can expose the API on the internet:

$ kubectl



Get the list of running services, as shown above – you should see that there is an IP address being applied to the “dockerdemo-deployment” (as above). This takes a little while to apply.

$ kubectl get services

1. All going well you should be able to see a result in our “/api/values” and “/api/database” endpoints.

# Workshop 3 – Make your own Container Registry (ACR)

We’re going to make our own container registry on ACR, push our images to it and connect our AKS instance to it…