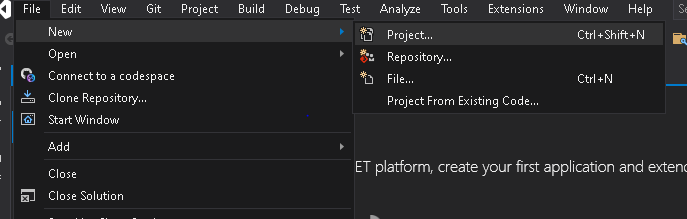
1. **Building ASP.NET Core MVC Application.**

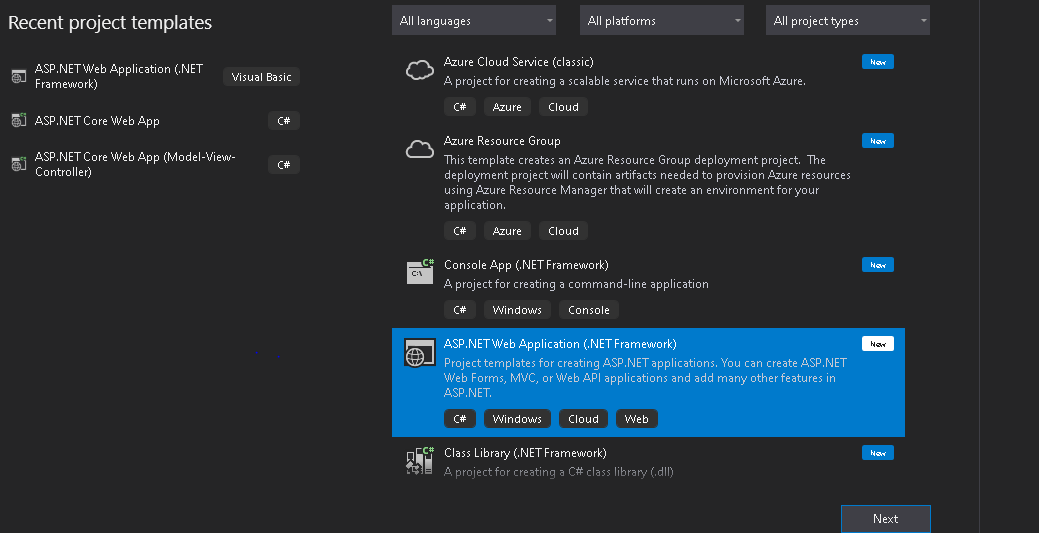
https://www.tutorialsteacher.com/mvc/create-first-asp.net-mvc-application

Steps:-

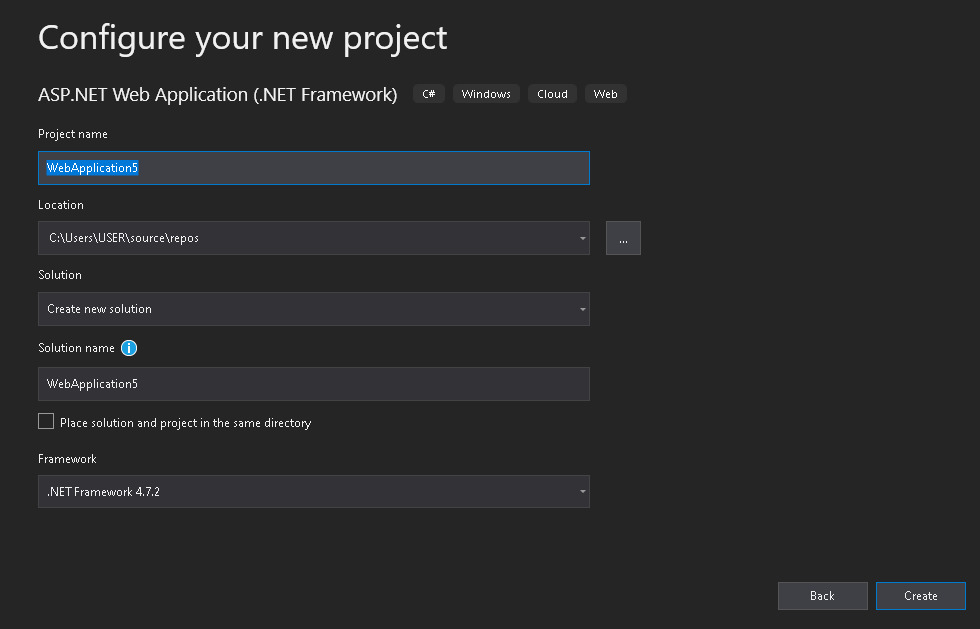
1. Go to file🡪 New🡪 Project



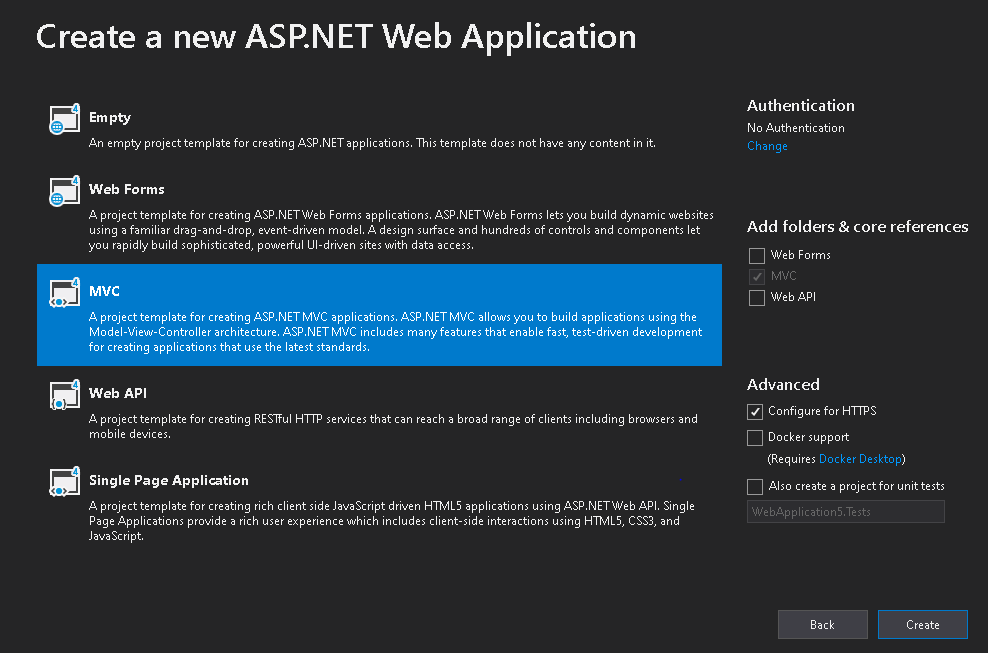
1. Select ASP.NET web application(.Net framework)



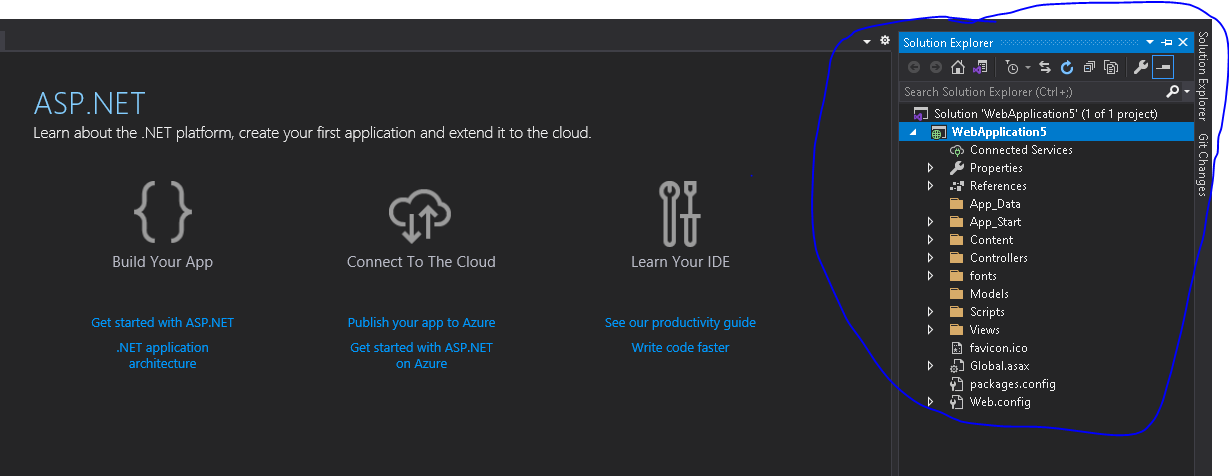
1. Hit next on the next screen you have to set the project name, destination location else everything you can keep it a default and hit the create button.



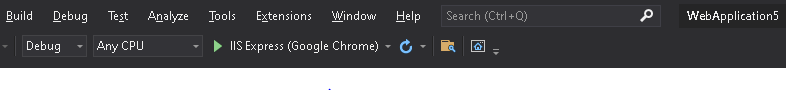
1. On the next screen do select the MVC then press create button.



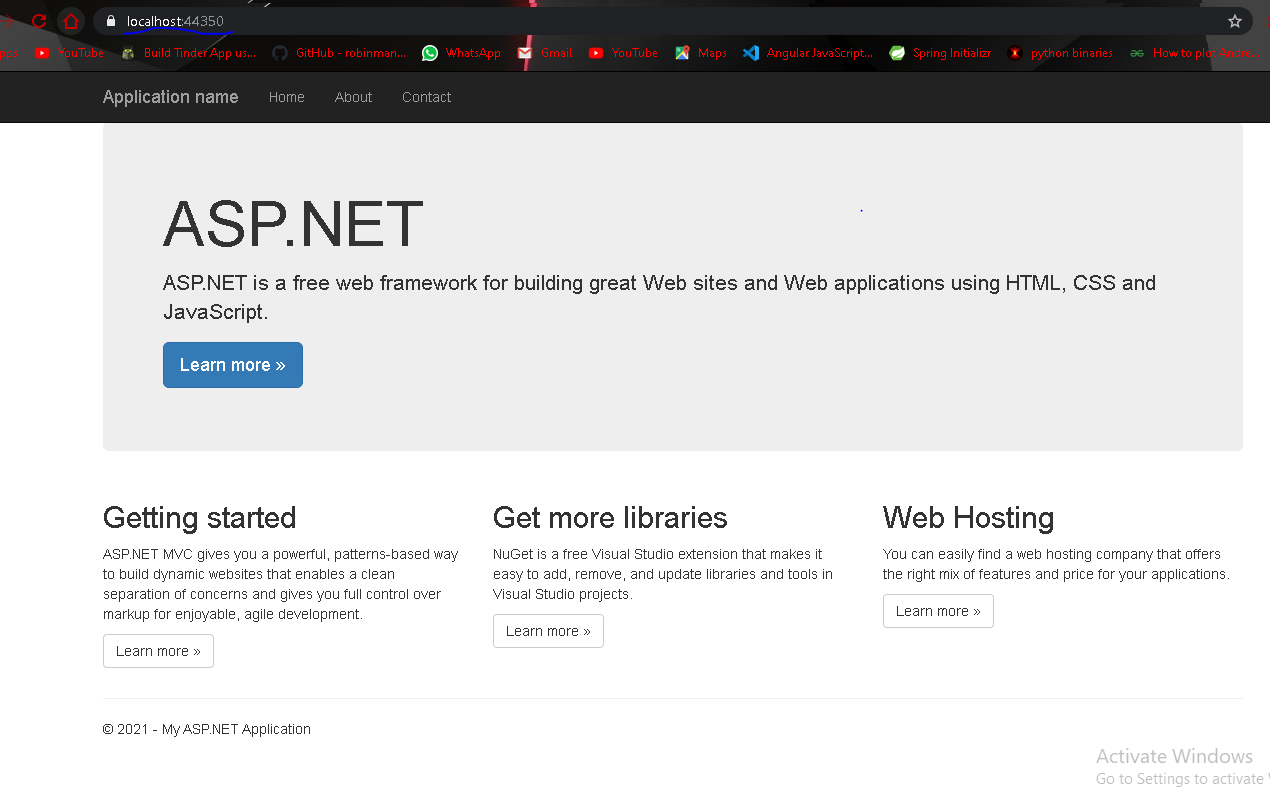
1. Once the project creation completes below screen will appear. On the right hand side of the screen there is solution explorer panel available. Over there you can implement your code and logic.



1. Once you done with all the changes. It is time to rum our code for that we have to click the IIS Express button to run our code.



1. If there is no errors in your code. It will successfully build your app and launch your browser. Over there you can see that your app is up and running on the localhost. For me it’s running on https://localhost:44350/



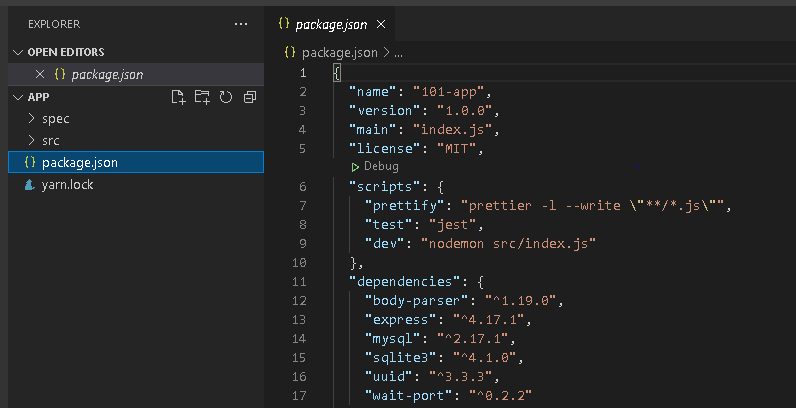
You can add different controllers to your code and perform other activities. Also, you can enhance the look and feel of your website with CSS and bootstrap

1. Working with Docker, Docker Commands, Docker Images and Containers

Docker images and container:-

<https://docs.docker.com/get-started/02_our_app/>

1. [Download the App contents](https://github.com/docker/getting-started/tree/master/app). You can either pull the entire project or download it as a zip and extract the app folder out to get started with
2. Once extracted, use your favorite code editor to open the project. If you’re in need of an editor, you can use [Visual Studio Code](https://code.visualstudio.com/). You should see the package.json and two subdirectories (src and spec).



Build the app’s container image

In order to build the application, we need to use a Dockerfile. A Dockerfile is simply a text-based script of instructions that is used to create a container image. If you’ve created Dockerfiles before, you might see a few flaws in the Dockerfile below. But, don’t worry. We’ll go over them.

1. Create a file named Dockerfile in the same folder as the file package.json with the following contents

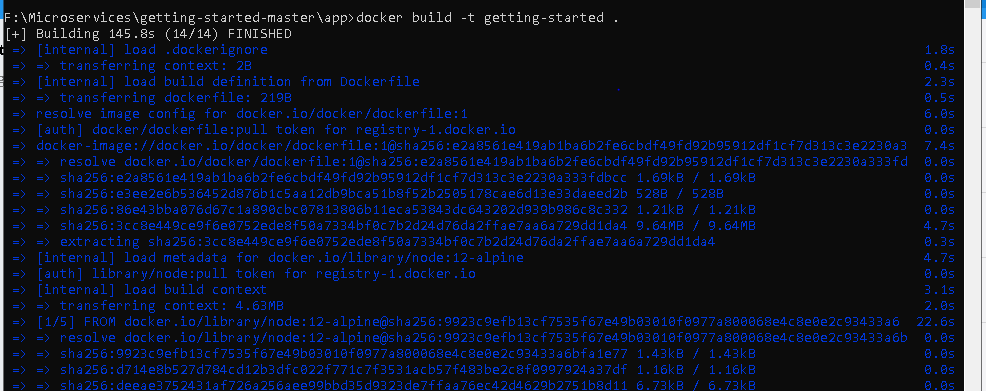
1. # syntax=docker/dockerfile:1
2. FROM node:12-alpine
3. RUN apk add --no-cache python g++ make
4. WORKDIR /app
5. COPY . .
6. RUN yarn install --production
7. CMD ["node", "src/index.js"]

Please check that the file Dockerfile has no file extension like .txt. Some editors may append this file extension automatically and this would result in an error in the next step.

2. If you haven’t already done so, open a terminal and go to the app directory with the Dockerfile. Now build the container image using the docker build command.

docker build -t getting-started .

As soon as you hit the above command in you cmd prompt it will start loading everything like below:-



This command used the Dockerfile to build a new container image. You might have noticed that a lot of “layers” were downloaded. This is because we instructed the builder that we wanted to start from the node:12-alpine image. But, since we didn’t have that on our machine, that image needed to be downloaded.

After the image was downloaded, we copied in our application and used yarn to install our application’s dependencies. The CMD directive specifies the default command to run when starting a container from this image.

Finally, the -t flag tags our image. Think of this simply as a human-readable name for the final image. Since we named the image getting-started, we can refer to that image when we run a container.

The . at the end of the docker build command tells that Docker should look for the Dockerfile in the current directory.

## Start an app container

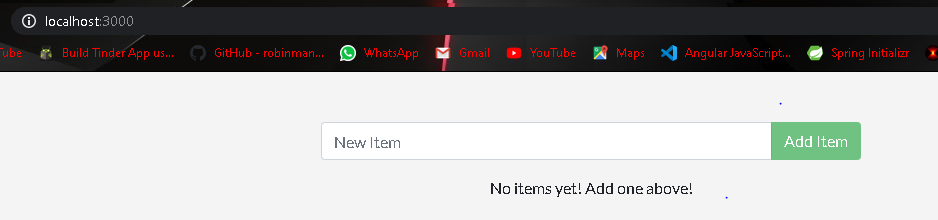
Now that we have an image, let’s run the application. To do so, we will use the docker run command (remember that from earlier?).

1. Start your container using the docker run command and specify the name of the image we just created:

docker run -dp 3000:3000 getting-started

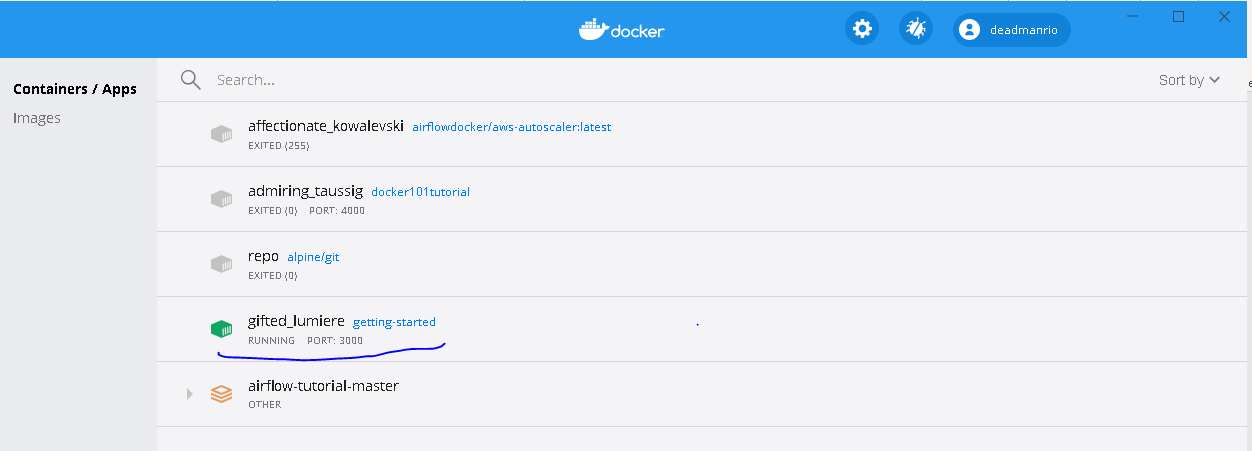
Remember the -d and -p flags? We’re running the new container in “detached” mode ( in the background) and creating a mapping between the host’s port 3000 to the container’s port 3000. Without the port mapping, we wouldn’t be able to access the application.

1. After a few seconds, open your web browser to [http://localhost:3000](http://localhost:3000/). You should see our app.



1. Go ahead and add an item or two and see that it works as you expect. You can mark items as complete and remove items. Your frontend is successfully storing items in the backend

As you can see in docker our container is running now



1. **Working with Docker Swarm**.( <https://docs.docker.com/get-started/swarm-deploy/>)

 Swarm provides many tools for scaling, networking, securing and maintaining your containerized applications, above and beyond the abilities of containers themselves.

In order to validate that our containerized application works well on Swarm, we’ll use Docker Desktop’s built in Swarm environment right on our development machine to deploy our application, before handing it off to run on a full Swarm cluster in production. The Swarm environment created by Docker Desktop is *fully featured*, meaning it has all the Swarm features your app will enjoy on a real cluster, accessible from the convenience of your development machine.

1. Pre-requisites:-

1. Docker 1.13 or higher

2. Docker Machine (pre installed for Docker for Windows and Docker for Mac)

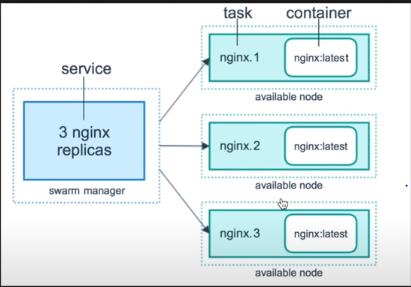
<https://docs.docker.com/machine/install-machine/#installing-machine-directly>

<https://docs.docker.com/get-started/swarm-deploy/>.

3 Here you will use same folder/project which we use in (Docker images and container of this file)

* Make sure that Swarm is enabled on your Docker Desktop by typing docker system info, and looking for a message Swarm: active (you might have to scroll up a little).

If Swarm isn’t running, simply type docker swarm init in a shell prompt to set it up.



## Describe apps using stack files

Swarm never creates individual containers like we did in the previous step of this tutorial. Instead, all Swarm workloads are scheduled as services, which are scalable groups of containers with added networking features maintained automatically by Swarm. Furthermore, all Swarm objects can and should be described in manifests called stack files. These YAML files describe all the components and configurations of your Swarm app, and can be used to easily create and destroy your app in any Swarm environment.

Let’s write a simple stack file to run and manage our bulletin board. Place the following in a file called bb-stack.yaml:

version: '3.7'

services:

bb-app:

image: bulletinboard:1.0

ports:

- "8000:8080"

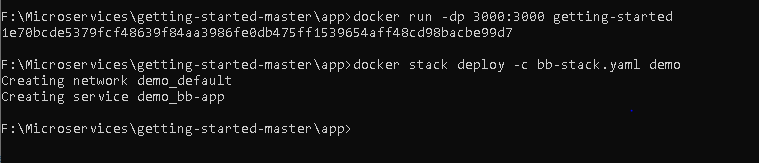
In this Swarm YAML file, we have just one object: a service, describing a scalable group of identical containers. In this case, you’ll get just one container (the default), and that container will be based on your bulletinboard:1.0 image created in (Docker images and container section of this file) In addition, We’ve asked Swarm to forward all traffic arriving at port 8000 on our development machine to port 8080 inside our bulletin board container.

## Deploy and check your application

1. Deploy your application to Swarm:

docker stack deploy -c bb-stack.yaml demo

If all goes well, Swarm will report creating all your stack objects with no complaints:



Notice that in addition to your service, Swarm also creates a Docker network by default to isolate the containers deployed as part of your stack.

1. Make sure everything worked by listing your service

docker service ls

If all has gone well, your service will report with 1/1 of its replicas created:



This indicates 1/1 containers you asked for as part of your services are up and running. Also, we see that port 8000 on your development machine is getting forwarded to port 8080 in your bulletin board container.

1. Open a browser and visit your bulletin board at localhost:8000; you should see your bulletin board, the same as when we ran it as a stand-alone container in Docker images and container section of this file.
2. Once satisfied, tear down your application

docker stack rm demo

1. **Working with Kubernetes.**

## Prerequisites

* Download and install Docker Desktop as described in [Orientation and setup](https://docs.docker.com/get-started/).
* Work through containerizing an application in [Part 2](https://docs.docker.com/get-started/02_our_app/).
* Make sure that Kubernetes is enabled on your Docker Desktop:
  + **Mac**: Click the Docker icon in your menu bar, navigate to **Preferences** and make sure there’s a green light beside ‘Kubernetes’.
  + **Windows**: Click the Docker icon in the system tray and navigate to **Settings** and make sure there’s a green light beside ‘Kubernetes’.

Kubernetes provides many tools for scaling, networking, securing and maintaining your containerized applications, above and beyond the abilities of containers themselves.

In order to validate that our containerized application works well on Kubernetes, we’ll use Docker Desktop’s built in Kubernetes environment right on our development machine to deploy our application, before handing it off to run on a full Kubernetes cluster in production. The Kubernetes environment created by Docker Desktop is *fully featured*, meaning it has all the Kubernetes features your app will enjoy on a real cluster, accessible from the convenience of your development machine.

## Describing apps using Kubernetes YAML

All containers in Kubernetes are scheduled as pods, which are groups of co-located containers that share some resources. Furthermore, in a realistic application we almost never create individual pods; instead, most of our workloads are scheduled as deployments, which are scalable groups of pods maintained automatically by Kubernetes. Lastly, all Kubernetes objects can and should be described in manifests called Kubernetes YAML files. These YAML files describe all the components and configurations of your Kubernetes app, and can be used to easily create and destroy your app in any Kubernetes environment.

1. You already wrote a very basic Kubernetes YAML file in the Orchestration overview part of this tutorial. Now, let’s write a slightly more sophisticated YAML file to run and manage our bulletin board. Place the following in a file called bb.yaml:

apiVersion: apps/v1

kind: Deployment

metadata:

name: bb-demo

namespace: default

spec:

replicas: 1

selector:

matchLabels:

bb: web

template:

metadata:

labels:

bb: web

spec:

containers:

- name: bb-site

image: bulletinboard:1.0

---

apiVersion: v1

kind: Service

metadata:

name: bb-entrypoint

namespace: default

spec:

type: NodePort

selector:

bb: web

ports:

- port: 8080

targetPort: 8080

nodePort: 30001

In this Kubernetes YAML file, we have two objects, separated by the ---:

* A Deployment, describing a scalable group of identical pods. In this case, you’ll get just one replica, or copy of your pod, and that pod (which is described under the template: key) has just one container in it, based off of your bulletinboard:1.0 image from the previous step in this tutorial.
* A NodePort service, which will route traffic from port 30001 on your host to port 8080 inside the pods it routes to, allowing you to reach your bulletin board from the network.

Also, notice that while Kubernetes YAML can appear long and complicated at first, it almost always follows the same pattern:

* The apiVersion, which indicates the Kubernetes API that parses this object
* The kind indicating what sort of object this is
* Some metadata applying things like names to your objects
* The spec specifying all the parameters and configurations of your object.

## Deploy and check your application

1. In a terminal, navigate to where you created bb.yaml and deploy your application to Kubernetes:

kubectl apply -f bb.yaml

you should see output that looks like the following, indicating your Kubernetes objects were created successfully:

deployment.apps/bb-demo created

service/bb-entrypoint created

1. Make sure everything worked by listing your deployments:

kubectl get deployments

if all is well, your deployment should be listed as follows:

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

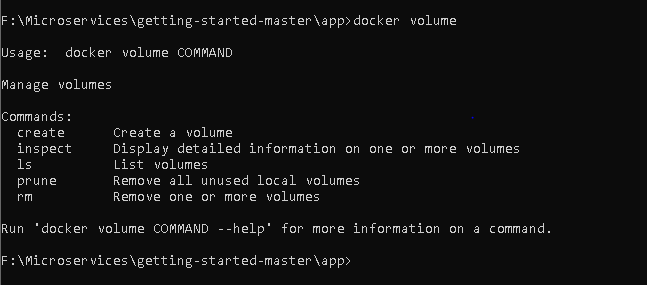
bb-demo 1 1 1 1 48s

1. In addition to the default kubernetes service, we see our bb-entrypoint service, accepting traffic on port 30001/TCP.
2. Open a browser and visit your bulletin board at localhost:30001; you should see your bulletin board, the same as when we ran it as a stand-alone container in container Docker images and container section of this file.
3. Installing software packages on Docker, Working with Docker Volumes and Networks

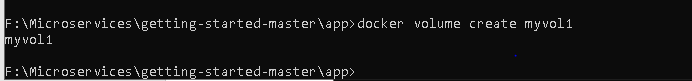
Installing software packages on Docker --- we can refer the web

Working with Docker Volumes explained below:-

1. Let us create the volume first. For the reference we will type below command:-
   * docker volume

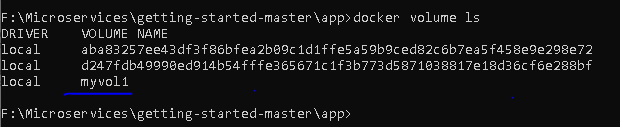


1. Now lets create the actual volume:-
   * docker volume create myvol1

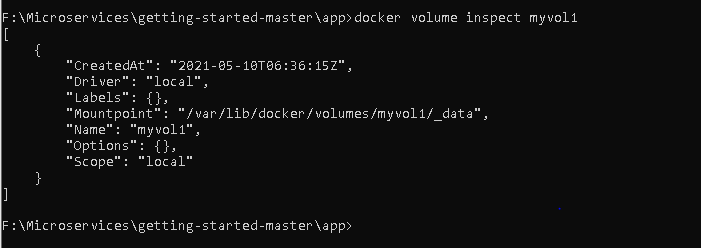


As you can see here our volume is created.

1. To list the volume we will write below command:-



1. To get the details of our volume we have to write below command:-
   * docker volume inspect myvol1



Here you can see all the details of our myvol1 i.e. name, created time, driver, mountpoint.

Our volume is located at the path mentioned in Mountpoint section.

1. To remove your volume you can write below command:-
   * docker volume rm myvol1

To remove all unused volumes we can write below command

* + docker volume prune

These are the basic functionalities of docker volume. You can explore more functionalities as well.

Working with docker network explained below:-

To write this command below is the syntax:-

* + docker network COMMAND

1. To Connect a container to a network
   * [docker network connect](https://docs.docker.com/engine/reference/commandline/network_connect/)
2. To create a network we have to write below command:-
   * docker network create
3. To disconnect a container from a network
   * docker network disconnect
4. To display detailed information on one or more networks
   * docker network inspect
5. To list the network:-
   * docker network ls
6. To remove all unused networks
   * docker network prune
7. To remove one or more networks
   * docker network rm