Index:

Practical No: 01

Aim: Building APT.NET Core MVC Application.

Description:

1)   Install .Net Core Sdk (Link:

https://dotnet.microsoft.com/learn/dotnet/hello-worldtutorial/install) 2)create folder MyMVC

 folder in D: drive or any other drive

   − **open command prompt and perform following operations Command: to create mvc project**

dotnet new mvc --auth none

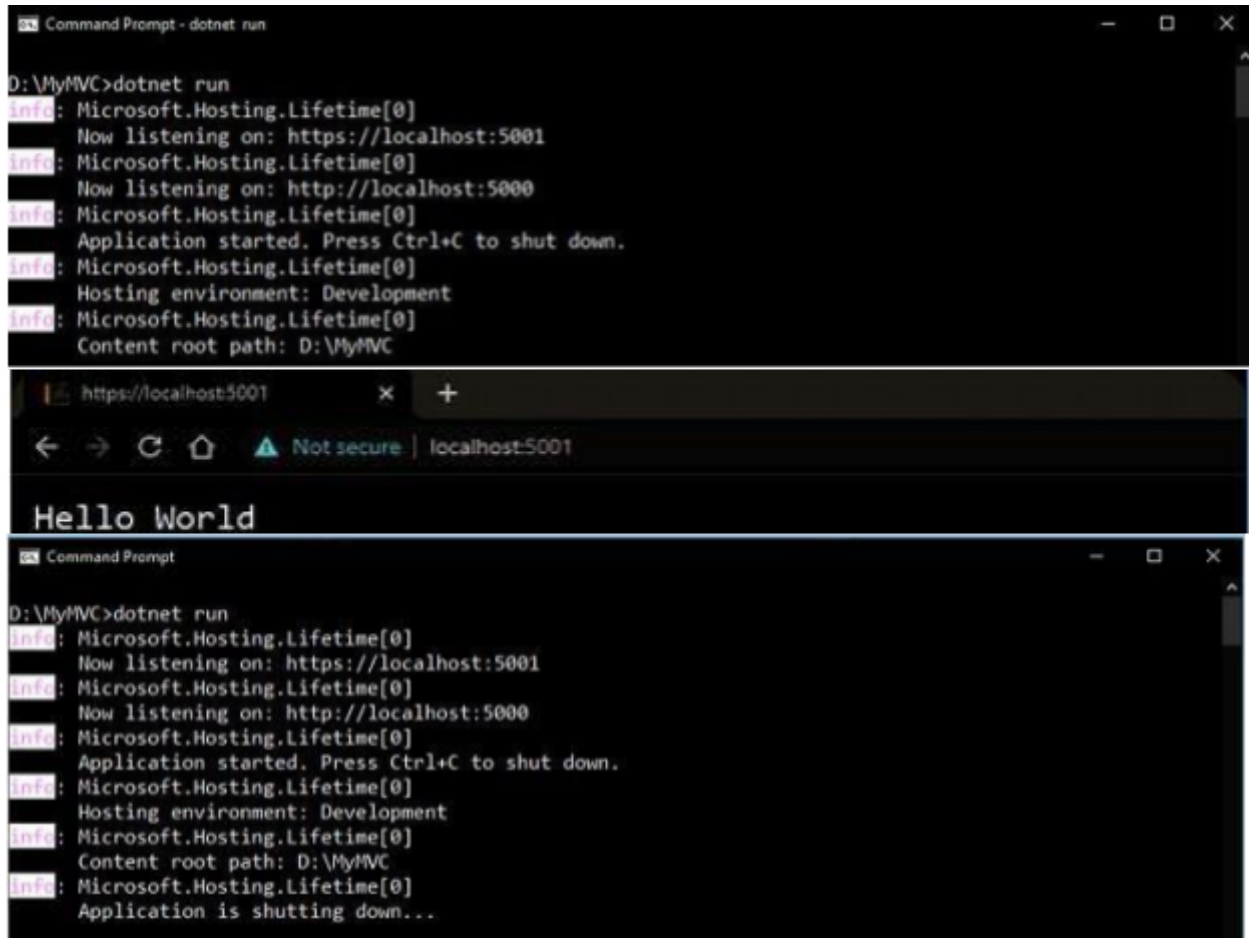## OUTPUT:

   − **Go to controllers' folder and modify HomeController.cs file to match following:**

```
using System;
    using      System.Collections.Generic;
    using System.Diagnostics;
    using System.Linq;
    using System.Threading.Tasks;
    using
    Microsoft.AspNetCore.Mvc;
    using
    Microsoft.Extensions.Logging;
    using MyMVC.Models:
     namespace MyMVC.Controllers
    {
    public class HomeController : Controller
    {
    public String Index()
    {
     return "Hello World";          }
    }
    }
```

Run the project Now open browser and and type URL: localhost:5000

Now go back to command prompt and stop running project using CTRL+C

– **Go to models folder and add new file StockQuote.cs to it with following content**

```
using System;

namespace

MyMVC.Models

{

public class StockQuote

{

 public string Symbol {get;set;} public int Price{get;set;}

}

}
```

– **Now Add View to folder then home folder in it and modify index.cshtml file to match following**

```
@{
ViewData["Title"] = "Home Page";

}
<div>
Symbol: @Model.Symbol <br/> Price: $@Model.Price <br/>
</div>
```

- **Now modify HomeController.cs file to match following:**

```
using System;

using
System.Collections.Generic;
using System.Diagnostics;
 using System.Linq;
using System.Threading.Tasks;
using
Microsoft.AspNetCore.Mvc;
using
Microsoft.Extensions.Logging;
using MyMVC.Models;
namespace MyMVC.Controllers
{
 public class HomeController : Controller
{
public async Task <IActionResult> Index()

{
var model= new StockQuote{ Symbol='HLLO', Price=3200}; return View(model);

}

}

}
```
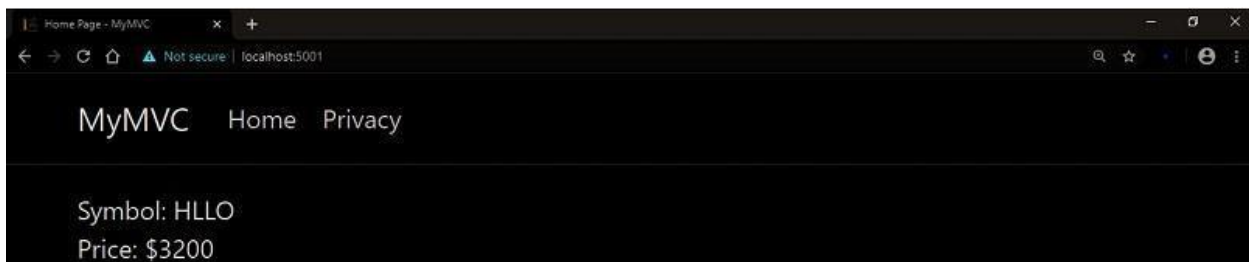
- **Now run the project using**

dotnet run

Now go back to browser and refresh to get modified view response

Practical No: 2

Aim: Building ASP.NET Core REST API.

Description:

Software requirement: Mr

1. Download and install

To start building .NET apps you just need to download and install the .NET SDK (Software Development Kit version 3.0

 https://dotnet.microsoft.com/learn/dotnet/hello-world-tutorial/install

Check everything installed correctly

Once you've installed, open a new command prompt and run the following

command: Command prompt

> dotnet

Create your web API

1. Open two command prompts

dotnet new webapi -o Glossary

```
Command Prompt

):\>dotnet new webapi -o Glossary
he template "ASP.NET Core Web API" was created successfully.

'rocessing post-creation actions...
lunning 'dotnet restore' on Glossary\Glossary.csproj...
  Restore completed in 1.13 sec for D:\Glossary\Glossary.csproj.

lestore succeeded.
```
OUTPUT:

Command:

Cd Glossary dotnet run

OUTPUT:

Command Prompt 2: (try running ready made weatherforecast class for testing)

Command: curl --insecure https://localhost:5001/weatherforecast

OUTPUT:

Now Change the content:

To get started, remove the WeatherForecast.cs file from the root of the project and the WeatherForecastController.cs file from the Controllers folder.

Add Following two files

1) D:\Glossary\GlossaryItem.cs (type it in notepad and save as all files)

```
//GlossaryItem.c
s namespace
Glossary
{
```

```csharp
public class GlossaryItem

    {

      public string Term { get; set;
      } public string Definition
      { get; set; }

    }

  }
```

D:\Glossary\Controllers\ GlossaryController.cs (type it in notepad and save as all files)

```csharp
cd Glossary dotnet run //Controllers/GlossaryController.cs  using
System; using System.Collections.Generic;
 using
 Microsoft.AspNetCore.Mvc;
 using System.IO;
namespace Glossary.Controllers
{
[ApiController] [Route("api/[controller]")]

    {

      private static List<GlossaryItem> Glossary
        = new List<GlossaryItem> {
        new GlossaryItem

        {

          Term= "HTML",

          Definition = "Hypertext Markup Language"

        },

        new GlossaryItem

        {

          Term= "MVC",

          Definition = "Model View Controller"

        },
```

```csharp
            new GlossaryItem

            {

                Term= "OpenID",

                Definition = "An open standard for authentication"

            }

        };
        [HttpGet] public ActionResult<List<GlossaryItem>> Get()

    {     return Ok(Glossary);

    }

[HttpDelete] [Route("{term}")] public

    ActionResult<GlossaryItem> Get(string term)

    {

        var glossaryItem = Glossary.Find(item =>
            item.Term.Equals(term,
            StringComparison.InvariantCultureIgnoreCase));


        if (glossaryItem == null)

        {       return NotFound();

        } else

        {

            return Ok(glossaryItem);

        }

}

        [HttpPost] public ActionResult Post(GlossaryItem glossaryItem)

        {

            var existingGlossaryItem = Glossary.Find(item =>
                item.Term.Equals(glossaryItem.Term,
                StringComparison.InvariantCultureIgnoreCase));
```
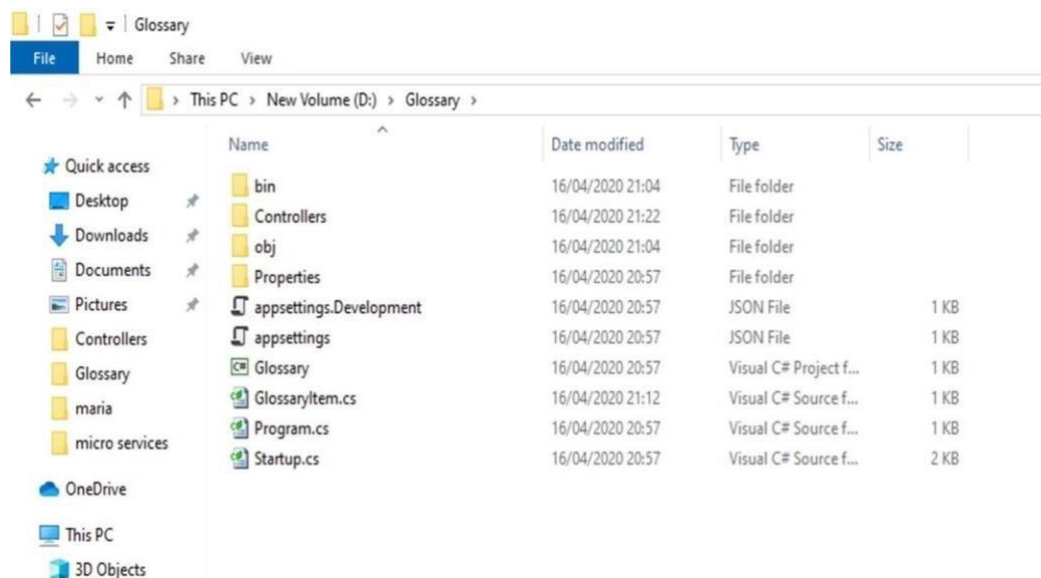
```csharp
        if (existingGlossaryItem != null)

        {

            return Conflict("Cannot create the term because it already exists.");

        }

        else

        {

            Glossary.Add(glossaryItem);

            var resourceUrl = Path.Combine(Request.Path.ToString(),
            Uri.EscapeUriString(glossaryItem.Term)); return Created(resourceUrl,
            glossaryItem);

        }

    }


[HttpPut] public ActionResult Put(GlossaryItem glossaryItem)
    {

        var existingGlossaryItem = Glossary.Find(item =>
        item.Term.Equals(glossaryItem.Term,
        StringComparison.InvariantCultureIgnoreCase));

        if (existingGlossaryItem == null)

        {

            return BadRequest("Cannot update a nont existing term.");

        } else

        {

            existingGlossaryItem.Definition
            = glossaryItem.Definition; return
            Ok();
```
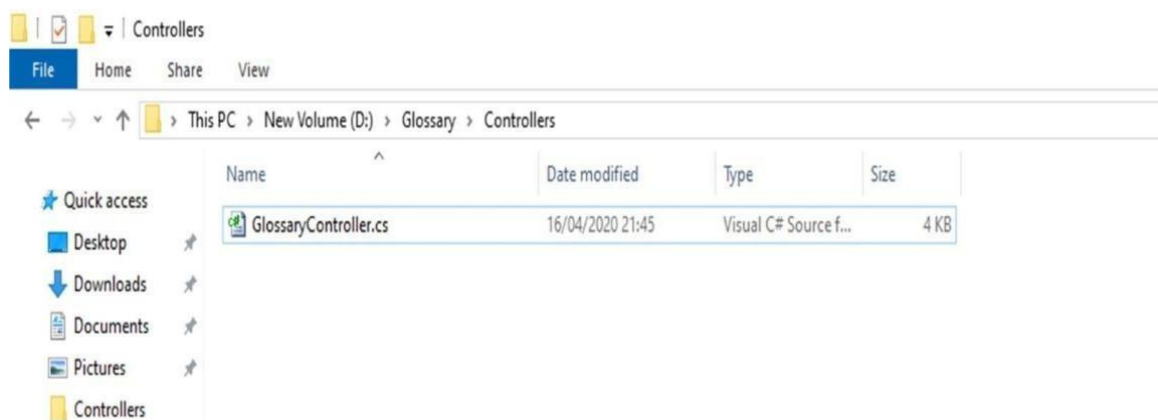
}}

```
public ActionResult Delete(string term)

    {

        var glossaryItem = Glossary.Find(item =>
            item.Term.Equals(term,
            StringComparison.InvariantCultureIgnoreCase));


        if (glossaryItem == null)

        {       return NotFound();

        }

        else

        {

                Glossary.Re
            move(glossaryIte m);
            return NoContent();

        }

        }

        }

        }
```

3.Now stop running previous dotnet run on command prompt 1 using Ctrl+C. and Run it again for new code. On Command prompt1:

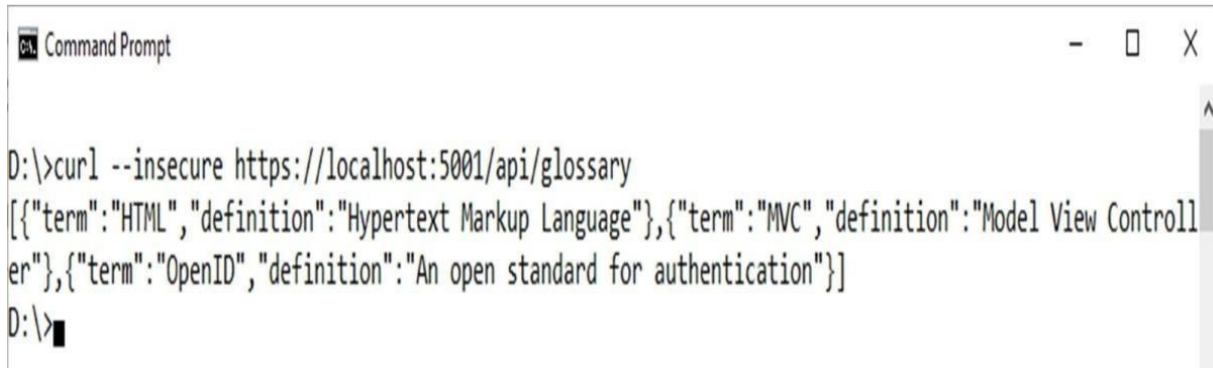Command: dotnet run

OUTPUT:

On Command prompt2:

1) Getting a list of items:

Command:

curl --insecure https://localhost:5001/api/glossary

```
Command Prompt                                        –  □  X

D:\>curl --insecure https://localhost:5001/api/glossary
[{"term":"HTML","definition":"Hypertext Markup Language"},{"term":"MVC","definition":"Model View Controll
er"},{"term":"OpenID","definition":"An open standard for authentication"}]
D:\>
```

OUTPUT:

Getting a single item Command:

curl --insecure https://localhost:5001/api/glossary/MVC

```
Command Prompt

D:\>curl --insecure https://localhost:5001/api/glossary/MVC
{"term":"MVC","definition":"Model View Controller"}
D:\>
```

2) Creating an item Command:

curl --insecure -X POST -d "{\"term\": \"MFA\", \"definition\":\"An authentication process.\"}" -H "Content- Type:application/json" https://localhost:5001/api/glossary

```
Command Prompt                                        –  □  X

D:\>curl --insecure -X POST  -d "{\"term\":  \"MFA\", \"definition\":\"An authentication process.\"}"  -H
 "Content-Type:application/json"   https://localhost:5001/api/glossary
{"term":"MFA","definition":"An authentication process."}
D:\>curl --insecure https://localhost:5001/api/glossary
[{"term":"HTML","definition":"Hypertext Markup Language"},{"term":"MVC","definition":"Model View Controll
er"},{"term":"OpenID","definition":"An open standard for authentication"},{"term":"MFA","definition":"An
authentication process."}]
D:\>
```
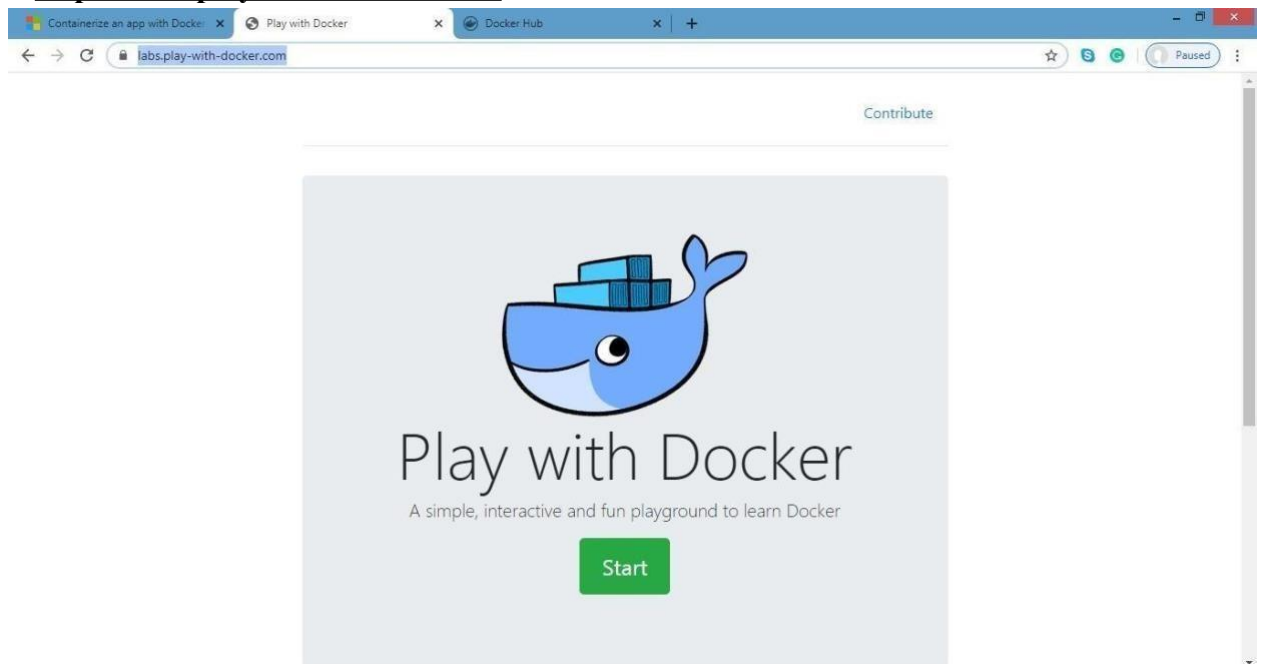
OUTPUT:

Update Item Command:

curl --insecure -X PUT -d "{\"term\": \"MVC\",
\"definition\":\"Modified record of Model View Controller.\"}"
-H "Content-Type:application/json"
https://localhost:5001/api/glossary

OUTPUT:

```
D:\>curl --insecure -X PUT  -d "{\"term\":  \"MVC\", \"definition\":\"Modified record of Model View Contr
oller.\"}"  -H "Content-Type:application/json"   https://localhost:5001/api/glossary

D:\>curl --insecure https://localhost:5001/api/glossary
[{"term":"HTML","definition":"Hypertext Markup Language"},{"term":"MVC","definition":"Modified record of
Model View Controller."},{"term":"OpenID","definition":"An open standard for authentication"},{"term":"MF
A","definition":"An authentication process."}]
D:\>
```

Delete Item Command:

curl --insecure --request DELETE --url https://localhost:5001/api/glossary/openid **OUTPUT:**

```
D:\>curl --insecure  --request DELETE  --url https://localhost:5001/api/glossary/openid

D:\>curl --insecure https://localhost:5001/api/glossary
[{"term":"HTML","definition":"Hypertext Markup Language"},{"term":"MVC","definition":"Modified record of
Model View Controller."},{"term":"MFA","definition":"An authentication process."}]
D:\>
```

Practical No: 3

Aim: **Working with Docker, Docker Commands, Docker Images and Containers** **Description:**

Step 1: create Docker Hub account (sign up)
Step 2 : login to https://labs.play-with-docker.com/

Click on start

Step 3: Add new instances
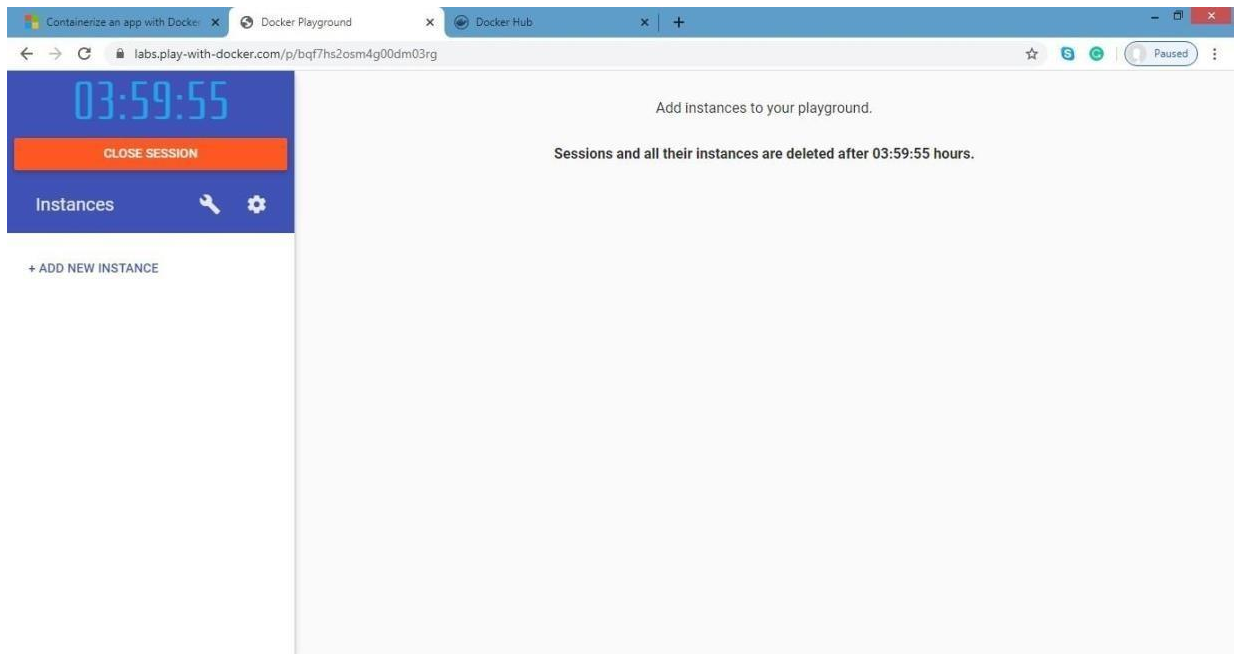
Step 4: perform following:
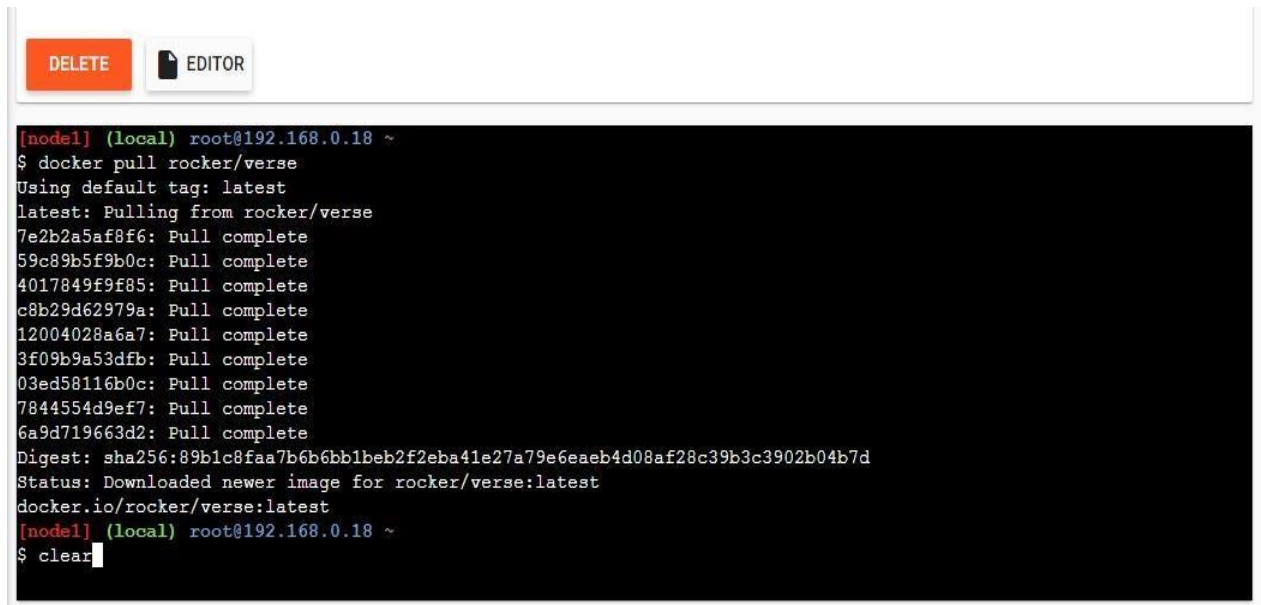
Method1: To pull and push images using docker

Command: to check docker
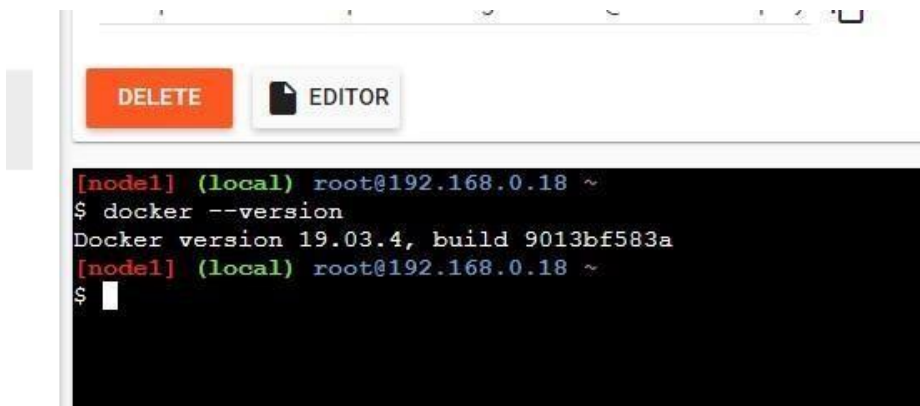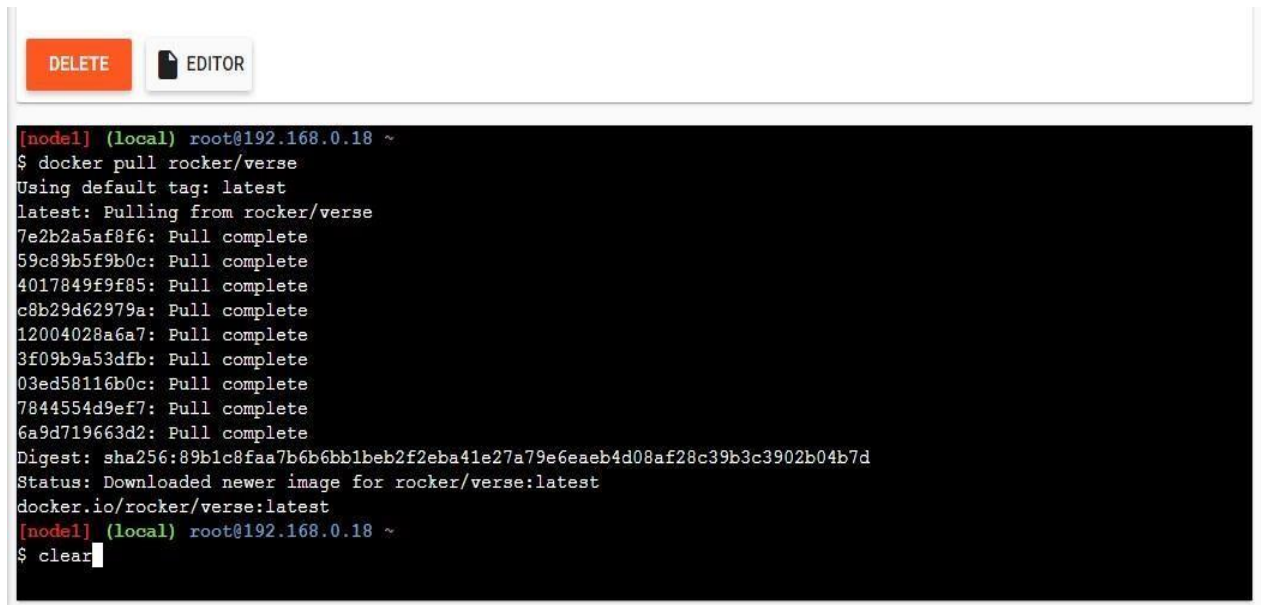version docker –version

OUTPUT:

Command: to pull readymade image
docker pull rocker/verse
OUTPUT:

Command: to check images in docker docker images

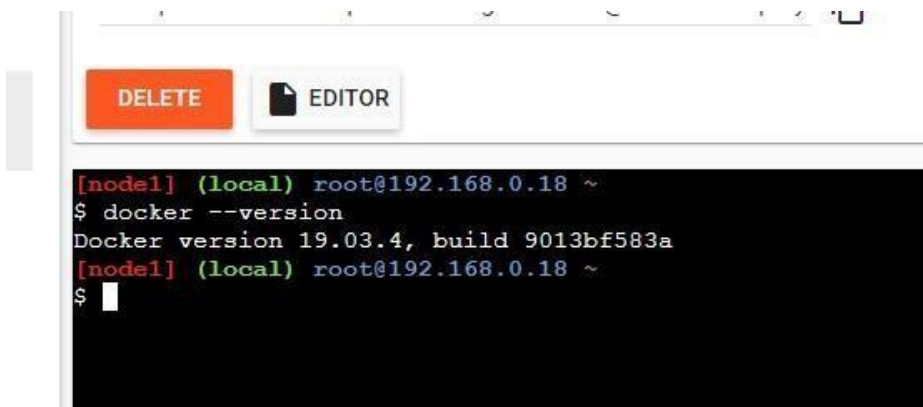4) perform following:

Method1:

To pull and push images using docker

Command: to check docker version docker
–version OUTPUT:

Command: to pull readymade image
        docker pull rocker/verse

OUTPUT:

Command: to check images in docker docker images

4) perform following:

Method1:

To pull and push images using docker

Command: to check docker version docker
–version **OUTPUT:**

Command: to pull readymade image
     docker pull rocker/verse
 OUTPUT:

```
[node1] (local) root@192.168.0.18 ~
$ docker pull rocker/verse
Using default tag: latest
latest: Pulling from rocker/verse
7e2b2a5af8f6: Pull complete
59c89b5f9b0c: Pull complete
4017849f9f85: Pull complete
c8b29d62979a: Pull complete
12004028a6a7: Pull complete
3f09b9a53dfb: Pull complete
03ed58116b0c: Pull complete
7844554d9ef7: Pull complete
6a9d719663d2: Pull complete
Digest: sha256:89b1c8faa7b6b6bb1beb2f2eba41e27a79e6eaeb4d08af28c39b3c3902b04b7d
Status: Downloaded newer image for rocker/verse:latest
docker.io/rocker/verse:latest
[node1] (local) root@192.168.0.18 ~
$ clear
```

Command: to check images in docker docker images

OUTPUT:


```
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
rocker/verse        latest          85c3e4e2c35e    4 days ago      3.15GB
[node1] (local) root@192.168.0.18 ~
$ docker tag 85c3e4e2c35e kbdocker11/repo1:firsttry
[node1] (local) root@192.168.0.18 ~
$ docker push kbdocker11/repo1:firsttry
The push refers to repository [docker.io/kbdocker11/repo1]
3e43a21d810a: Mounted from rocker/verse
8fdb254334fd: Mounted from rocker/verse
6611ef73af7c: Mounted from rocker/verse
7ec16b3cc818: Mounted from rocker/verse
a2f3120be52c: Mounted from rocker/verse
beb6bc4429d0: Mounted from rocker/verse
828281284548: Mounted from rocker/verse
61fb5e16e303: Mounted from rocker/verse
461719022993: Mounted from rocker/verse
firsttry: digest: sha256:89b1c8faa7b6b6bb1beb2f2eba41e27a79e6eaeb4d08af28c39b3c3902b04b7d size: 2211
[node1] (local) root@192.168.0.18 ~
$
```

Check it in docker hub now
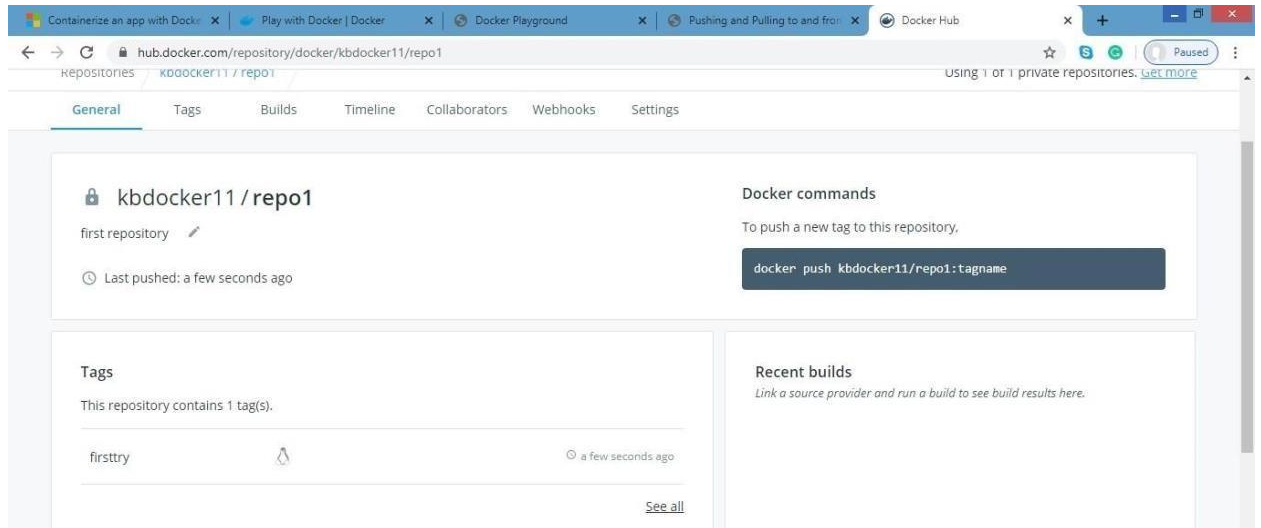
Click on tags and check
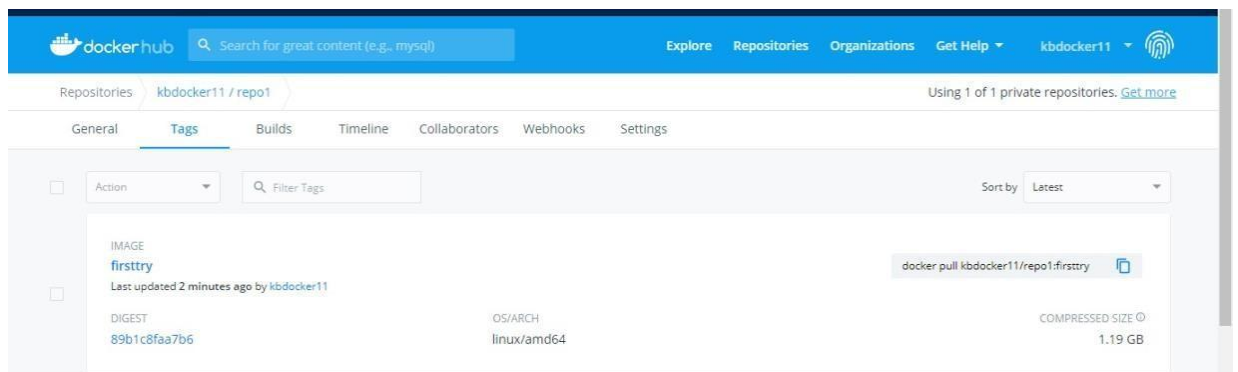
Method 2:

Build an image then push it to docker and run it

Command : to create docker file

1. cat > Dockerfile <<EOF
2. FROM busybox
3. CMD echo "Hello world! This is my first Docker image."
4. EOF

OUTPUT:

Command : to build image from docker file dokcer build
–t kbdocker11/repo2 .

OUTPUT:

Command: to check docker images docker

images OUTPUT:

Command: to push image to docker hub
docker push kbdocker11/repo2 .

OUTPUT:

Now check it on docker hub
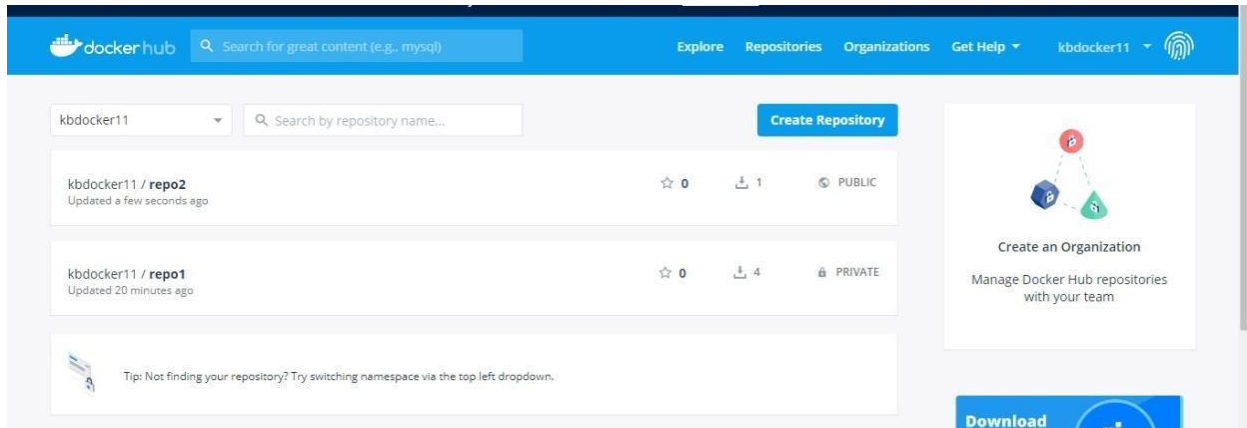
command: to run docker image: docker run kbdocker11/repo2 OUTPUT:

Now close session.

Practical No: 4

Aim: Installing software packages on Docker, Working with Docker Volume and Networks.
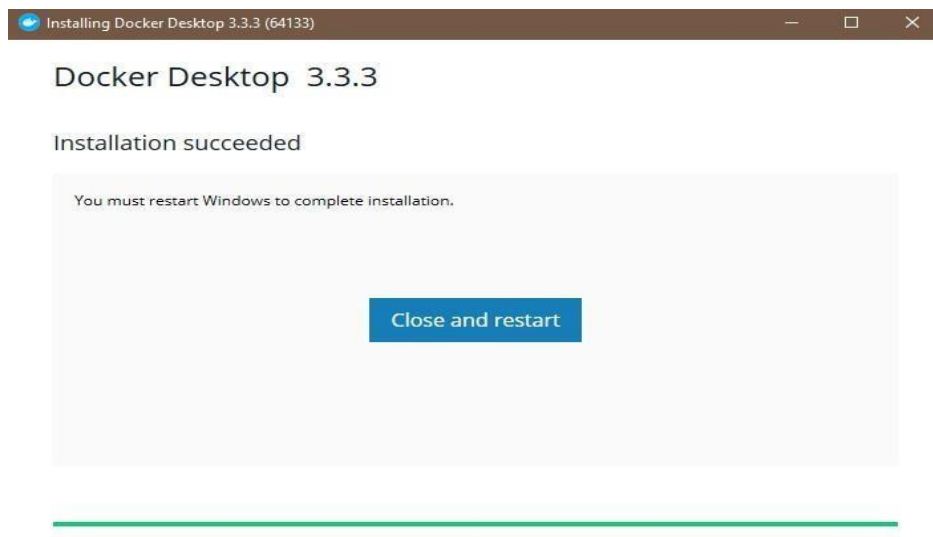
Description:

If you've already run the command to get started with the tutorial, congratulations! If not, open a command prompt or bash window, and run the command:

```
docker run -d -p 80:80 docker/getting-started
```

You'll notice a few flags being used. Here's some more info on them:

- -d - run the container in detached mode (in the background)
- -p 80:80 - map port 80 of the host to port 80 in the container
- docker/getting-started - the image to use

```
docker run -dp 80:80 docker/getting-started
```

Installing Docker Desktop 3.3.3 (64133)　　　　　　　　　　— □ ✕

Docker Desktop 3.3.3

Installation succeeded

You must restart Windows to complete installation.

Close and restart

What is a container?

Now that you've run a container, what *is* a container? Simply put, a container is simply another process on your machine that has been isolated from all other processes on the host machine. That isolation leverages kernel namespaces and cgroups, features that have been in Linux for a long time. Docker has worked to make these capabilities approachable and easy to use.
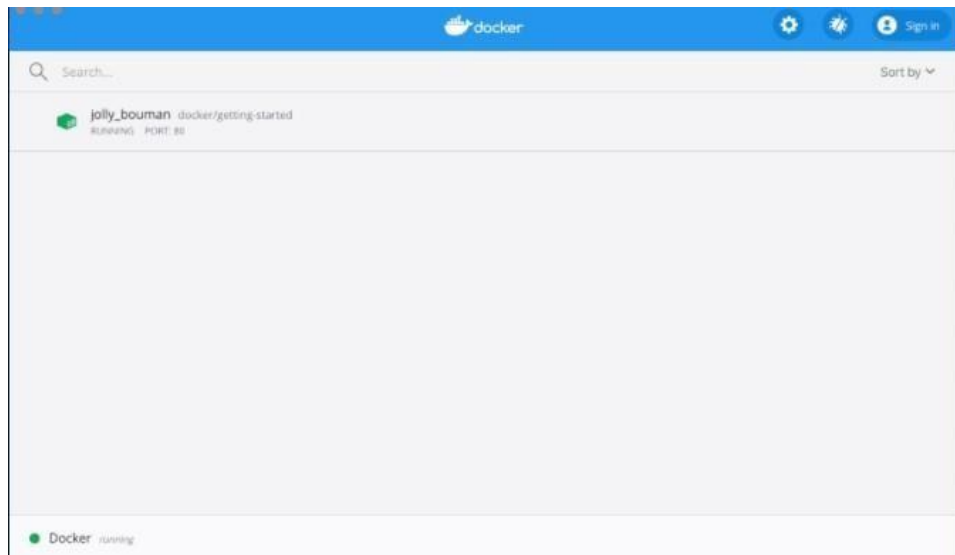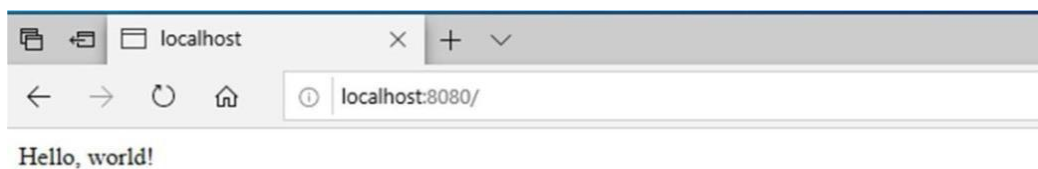
Step 1: $ docker run -p 8080:8080 dotnetcoreservices/hello-world


```
C:\Windows\system32>docker run -p 8080:8080 dotnetcoreservices/hello-world
Hosting environment: Production
Content root path: /pipeline/source/app/publish
Now listening on: http://0.0.0.0:8080
Application started. Press Ctrl+C to shut down.
```

Step 2: Run Localhost in browser

Hello, world!

Step 3: $ docker ps


```
C:\Windows\system32>docker ps
CONTAINER ID   IMAGE                            COMMAND             CREATED        STATUS        PORTS                    NAMES
ea1f20370993   dotnetcoreservices/hello-world   "/pipeline/source/ap…"  2 minutes ago  Up 2 minutes  0.0.0.0:8080->8080/tcp   recursing_lalande

C:\Windows\system32>
```

Step 4: curl http://localhost:8080/will/it/blend?

```
C:\Windows\system32>curl http://localhost:8080/will/it/blend?
Hello, world!
```

Step 5: $ docker kill PID (process id of application).

```
C:\Windows\system32>docker kill ea1f20370993
ea1f20370993

C:\Windows\system32>
```

Step 6 : Process Id terminated

```
C:\Windows\system32>docker ps
CONTAINER ID      IMAGE          COMMAND          CREATED          STATUS          PORTS          NAMES

C:\Windows\system32>
```
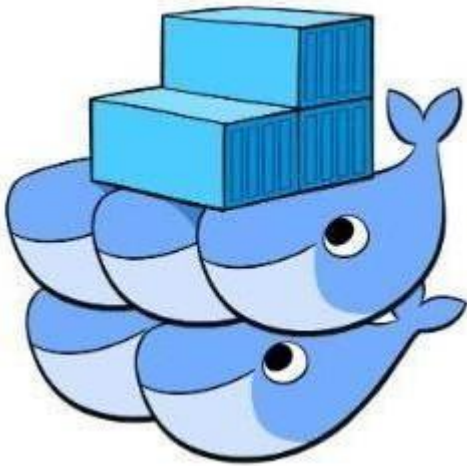
Practical No: 5

Aim: **Working with Docker Swarm.**

Description:

What is Docker Swarm?

Docker Swarm is an orchestration management tool that runs on Docker applications. It helps end-users in creating and deploying a cluster of Docker nodes.

Step 1: Update Software Repositories Run the following

command on the terminal: sudo apt-get update

Step 2: Uninstall Old Versions of Docker

Before proceeding, uninstall the old Docker software and use the following command:

sudo apt-get remove docker docker-engine docker.io

Step 3: Install Docker

To install Docker on Ubuntu, run the following command:

sudo apt install docker.io

Step 4: Set-up Docker

Set-up and run Docker service by entering the following commands in the terminal window:

sudo systemctl start docker sudo systemctl

enable docker

Step 5: Verify Docker Version

To check the installed Docker version, enter the following command:

sudo docker --version

Step 6: Run Docker Container

To run a Docker container, it's important to pull a Docker Image (such as MySQL) from Docker

Hub. sudo docker pull mysql sudo docker run -d -p0.0.0.0:80:80

mysql:latest

Now, Docker pulls the latest MySQL image from the hub. List down all the available Docker images on your machine by using the following command:

sudo docker ps -a

Step 7: Create Swarm

Here, create a cluster with the IP address of the manager

node. sudo Docker Swarm init --advertise-addr

192.168.2.151 Subsequently, you should see the following

OUTPUT:

Manager Node

This means that the manager node is successfully configured. Now, add worker node by copying the command of the "swarm init" and paste the output onto the worker node:

sudo Docker Swarm join --token SWMTKN-1- xxxxx

Your worker node is also created if you see the following OUTPUT:

Worker Node

Now, go back to the manager node and execute the following command to list the worker node: sudo docker node ls

Here, you must see the worker node in the following OUTPUT:

Swarm Cluster - Docker Swarm

The above image shows you have created the Swarm Cluster successfully. Now, launch the service in Swarm Mode. Go to your the manager node and execute the command below to deploy a service:

sudo docker service create --name HelloWorld alpine ping docker.com

Service Created - Docker Swarm

By executing the above command, you can access the HelloWorld file from the remote system. To see the output, you can check the services with the following command:

sudo docker service ls

Finally, you should be able to see the following OUTPUT:

Practical No: 06

Aim: **Working with Circle CI for continuous integration.**

Description:

Prerequisites

To follow along with the tutorial, a few things are required:

1. Python installed on your local system
2. A Circle CI account
3. A GitHub account

Building the app

For simplicity, we will create a Flask application. Flask is a microframework for Python. For our exercise, minimal knowledge of the framework is necessary.

First, create a project directory (folder) and cd into it. Type this into Terminal:

mkdir python_app && cd $_/

Next, open your favorite editor and create a hello.py file. Then, copy the following lines into that file: from   flask   import Flask app   =
Flask(_name_) @app.route("/")
def hello():
return "Hello World!"

Running the app

Now it is time to create a requirements.txt file in our editor. Add the word **Flask** to the file and save it.

Then, within the virtual environment, install the package by running:

The final command to run this application is:

<mark>FLASK_APP=hello.py flask run</mark>

You can see the application running on your browser at http://localhost:5000/.

CircleCI config file

Create a .circleci folder and inside of that create a config.yml file. Then, copy these lines into it:

```
version: 2 jobs: build:
docker:
-         image: circleci/python:3.6 steps:
-         checkout - restore_cache: key: deps1-{{ .Branch
}}-{{ checksum "requirements.txt" }} - run:
command: | python3 -m venv venv .
venv/bin/activate    pip    install    -r
requirements.txt - save_cache:
key: deps1-{{ .Branch }}-{{ checksum "requirements.txt" }}  paths: -
"venv" - run:
name: Running
tests command: |
.    venv/bin/activate    python3
tests.py - store_artifacts: path:
test-reports/          destination:
python_ap
```

Pushing to GitHub

Using the philosophy of committing your code early and often, we should have initialized Git earlier in this process, and we would have atomic commits. Because this tutorial is about integration of CircleCI and GitHub, I intentionally put it on hold until now.

Our current code structure looks like this:

We can now commit our code by running the following commands:

```
git add .
git commit -m "Initial commit"
```

Creating GitHub Repository

After creating your new repository, you will get to a page like this one:
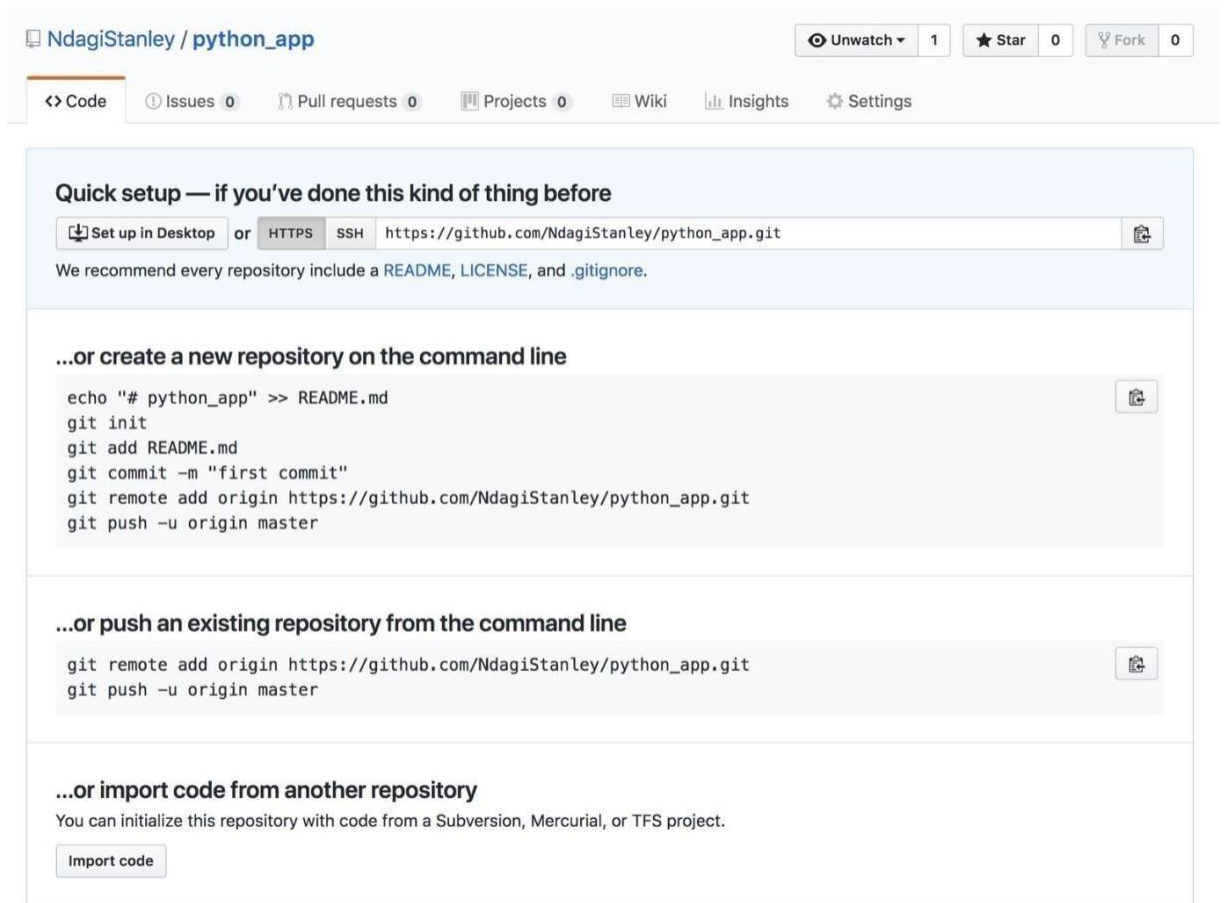
We will go with the second option, …push an existing repository. Run:

git remote add origin https://github.com/NdagiStanley/python_app.git
git push -u origin master

Configuring CircleCI

Now that the repo is on GitHub, we can finalize the CI by configuring CircleCI. Head on over to the CircleCI sign up page. Sign up for CircleCI with your GitHub account.
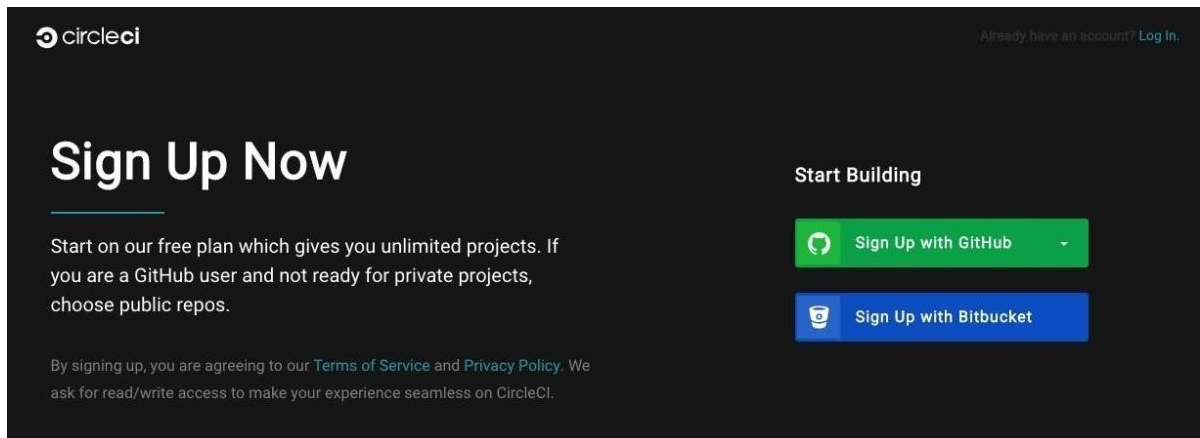
Once you are logged in, make sure that your personal GitHub account is active. If you are in several GitHub organizations, one of them might be active. Just click the drop-down menu (top left) and select your GitHub username. Then, click Add Projects. The most recent project, 'python_app', is listed there.
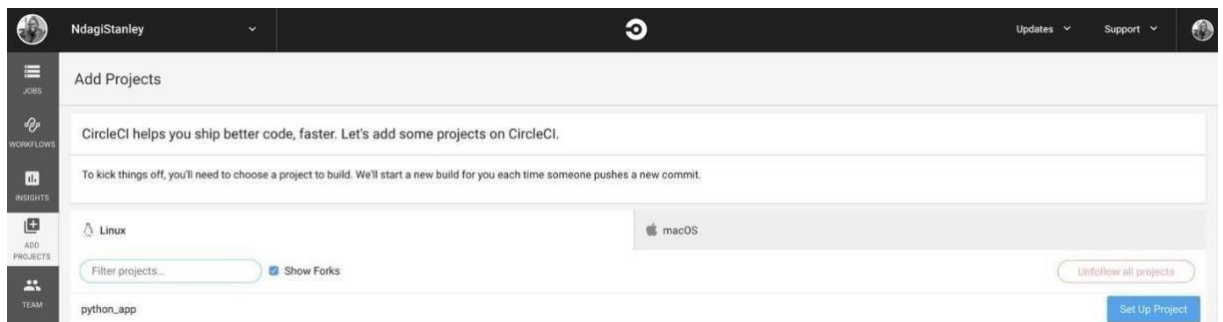
Click Set up Project at the right side of the row that includes our project. On the redirected page, you will notice the Next Steps section. Had we not had our own .circleci/config.yml file, we would have started at No.

1. Because we do have the config.yml file, we can just scroll to No. 5 and click Start building.
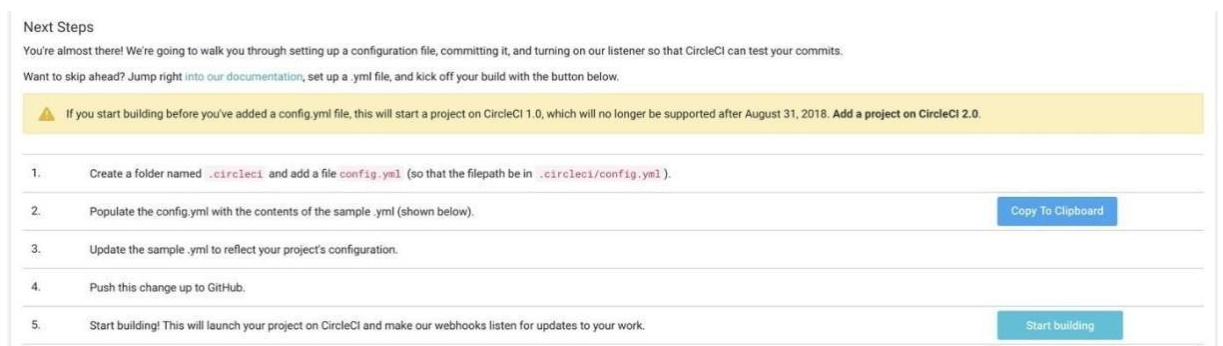
Within no time, the build passes. Success!
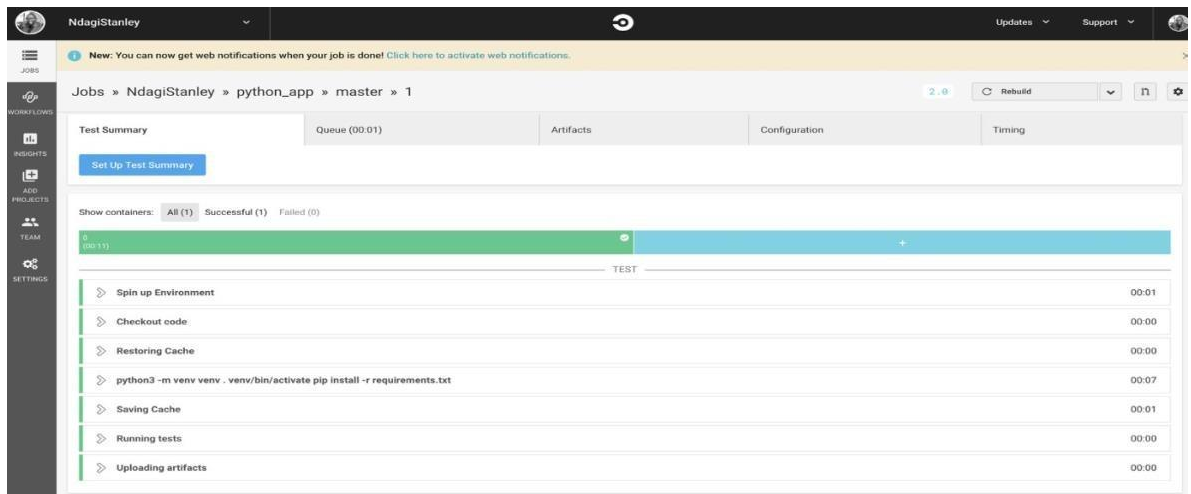
In the top right corner, click the Settings cog. Then click Projects on the left, and finally, python_app.
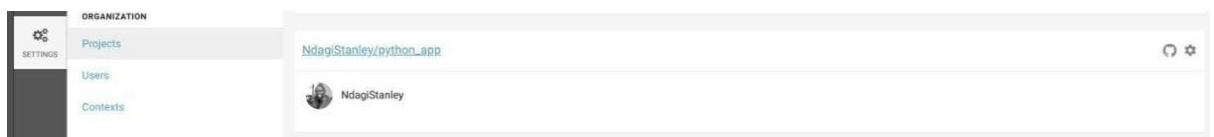
You will be on a path like this one: circleci.com/gh/<username>/python_app. Mine is https://circleci.com/gh/NdagiStanley/python_app. Click the settings cog next to the repo name: python_app.
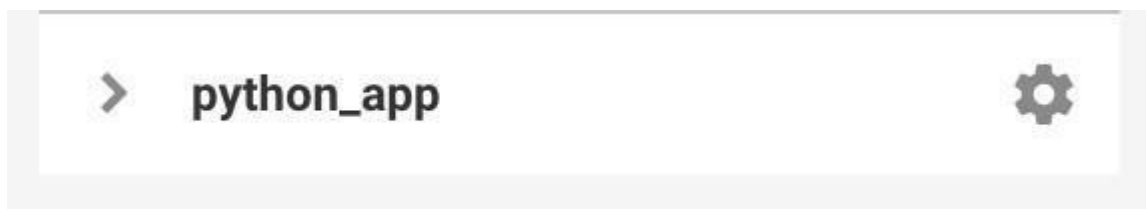
It is important that you become familiar with the settings that you can change for this project. I will touch on what is relevant to us now.

In Advanced Settings, notice that Only build pull requests is turned off. This means that every push to GitHub will run on CircleCI, including PRs. **ReadME - status badge**

On our local machine, check out to another Git branch by running:

git checkout -b add_readme

Open your editor and create a README.md file. Copy and paste the following lines into this file:

README.md
 # PYTHON APPLICATION

This Python application repo was created to showcase the integration between GitHub and CircleCI. [![CircleCI](https://circleci.com/gh/NdagiStanley/python_app.svg?style=svg)](https://c ircleci.com/ gh/NdagiStanley/python_app)

I added a title and a brief description to
mine. Now, run the following commands:

```
git add .
 git commit -m "Add README" git
push -u origin add_readme
```

If you go to https://github.com//python_app you will notice that we have a new branch:
`add_readme`. We can go ahead and click Compare and pull request.

Opening a pull request

This is how I set up my PR:

Click Create pull request and in no time, this is what we get:

A successful build! Now, click Show all checks. Notice that the check is from CircleCI.

Even the browser's tab favicon shows a tick for the successful run. If you click Details, this will redirect you to the build on CircleCI:

Notice that the favicon here also shows that the build is successful:
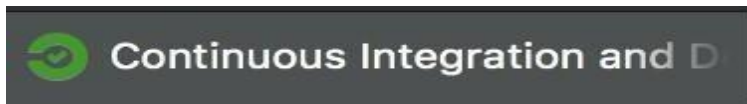
At the top, click python_app.

You will be redirected to the builds for this project:

Conclusion

There you have it! We have enabled continuous integration with CircleCI.

Practical No: 07

Aim: **Creating Microservice with ASP.NET Core.**

Description:

Create a new ASP.NET Core project. I called mine Hello Microservice (and the [full](#) [source code is available](#) [on](#) [GitHub](#) for your reference

Step 1: Create a new solution

Step 2: Add docker-compose support
Right click on the **project** and click "Add" and then "Container Orchestrator Support".
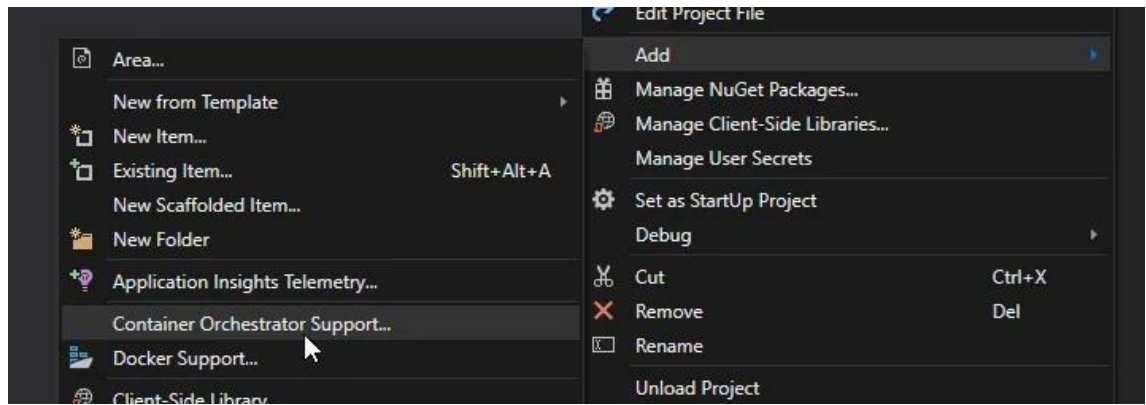
Keep in mind that the sole purpose of this project is to start development for a proof of concept for ASP.NET Core Microservices. Docker-compose is easier to deal with than Kubernetes for local machine development. This is why I'm choosing the "Docker Compose" option, even though I may eventually want to **deploy** to Kubernetes.
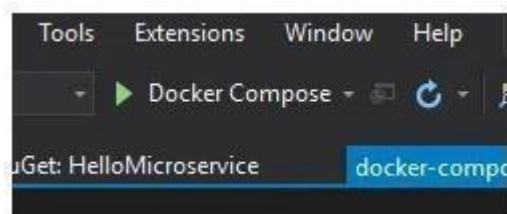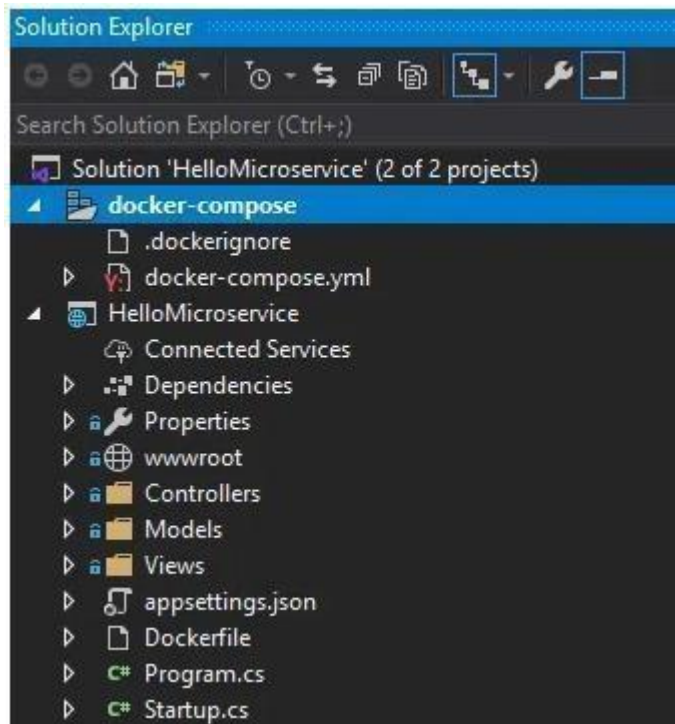
At this point, you can hit CTRL+F5 to run. Visual Studio will use Docker Compose to create an image of your project and run it within Docker. Your browser should open automatically, and you'll see the standard "Welcome" screen
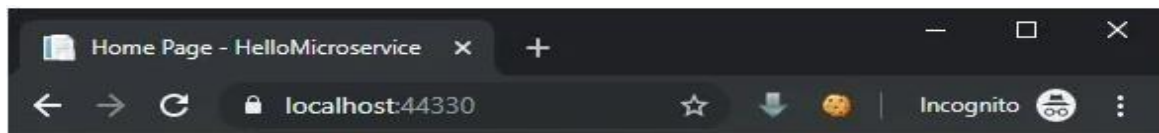
Step 3: Add database orchestration to docker-compose

Couchbase makes official container images available on Docker Hub. To use these images, lets add another service under services: in the docker-compose.yml file:

```
1  services:
2    couchbase:
3      image: couchbase:enterprise-6.0.3
4      volumes:
5        - "./couchbase:/opt/couchbase/var" # local folder where couchbase data stored
6      ports:
7        - "8091:8091" # Allows local access to mgmt UI
```

```
1  PS C:\WINDOWS\system32> docker ps
2  IMAGE                      COMMAND         CREATED    STATUS  PORTS         NAMES
3  hellomicroservice:dev      "tail -fâ€¦"   8 minâ€¦   Up 8â€¦  0.0.0.0...    dockercompose1524â€¦¦56534_hel1
4  couchbase:enterprise-6.0.3 "/entrypâ€¦¦"  8 minâ€¦   Up 8â€¦  8092-8096...  dockercompose1524â€¦¦56534_cou¢
5  PS C:\WINDOWS\system32>
```

Step 4: Configuration changes

Because I've used the stock couchbase: enterprise-6.0.3 Docker image, I still need to open the Couchbase UI (http://localhost:8091) and setup the cluster manually. See "next steps" at the end for some options to automate.

Couchbase > New Cluster

Cluster Name

HelloMicroserviceCouchbaseCluster

Create Admin Username

Administrator

Create Password   **Confirm Password**

....

< Back   Next: Accept Terms

```
1  hellomicroservice:
2    image: ${DOCKER_REGISTRY-}hellomicroservice
3    build:
4      context: .
5      dockerfile: HelloMicroservice/Dockerfile
6    environment:
7      Couchbase__Servers__0: http://couchbase:8091/ # Reference to the "couchbase" service name on line 4
8    depends_on:
9    - couchbase # Reference to the "couchbase" service name on line 4
10   command: ["./wait-for-it.sh", "http://couchbase:8091"]
```

Using wait-for-it **is optional**, but when you're right in the middle of development, this may save you some headaches. Finally, to make the ASP.NET Core project talk to Couchbase, I used the Dependency Injection Extension from NuGet

```
1  services.AddCouchbase(Configuration.GetSection("Couchbase"));
```

```
 2    "Logging": {
 3      "LogLevel": {
 4        "Default": "Debug",
 5        "System": "Information",
 6        "Microsoft": "Information"
 7      }
 8    },
 9    "Couchbase" : {
10      "Username": "Administrator",
11      "Password": "password",
12      "Servers" : ["http://thiswillbeoverwritten"]
13    }
14 }
```

Step 5: Using the database

Finally, let's make sure that the ASP.NET Core application is able to communicate with the database. I added

a very simple Insert and Get to the HomeController Index method:

```
 1  public IActionResult Index()
 2  {
 3      var id = Guid.NewGuid().ToString();
 4
 5      _bucket.Insert(new Document<dynamic>
 6      {
 7          Id = id,
 8          Content = new
 9          {
10              hello = "microservice",
11              foo = Path.GetRandomFileName()
12          }
13      });
14
15      var doc = _bucket.Get<dynamic>(id);
16
17      return View(doc);
18  }
```

```
 2  @{
 3      ViewData["Title"] = "Home Page";
 4  }
 5
 6  <div class="text-center">
 7      <h1 class="display-4">Welcome</h1>
 8      <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.<
 9  </div>
10
11  <p>Created a new document:</p>
12  <ul>
13      <li>ID: @Model.Id</li>
14      <li>Document: @Model.Value</li>
15  </ul>
```

And now, we have the basics of ASP.NET Core Microservices in place. Hit CTRL+F5 to run the service.

When the browser opens, you should see something like this: