Lab Scenario:

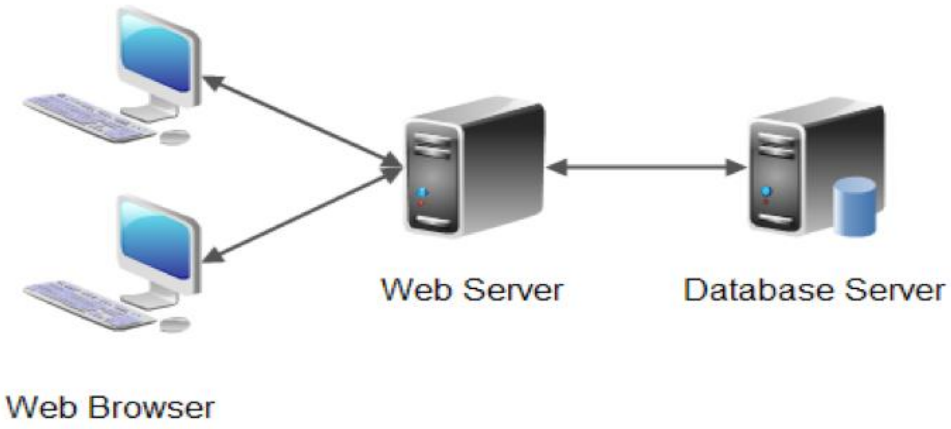
Simple Web Application

So far we have talked about Containers and Kubernetes and setup on the cloud. Time to test drive by building a simple application that will run in a Kubernetes cluster on the IBM Cloud. As part of this process we will run and test each of the containers that are deployed as part of the application. We build and then deploy both web and redis containers as an application to run as a single application.

We start with a containerized web application and a standard redis server container image to build a typical enterprise application as part of modernizing it. We will then move these containerized components of the application into the cloud on Kubernetes and scale it.

Section Scenario (we as an attendee of the class)

- We have an existing containerized application. It is being run in production without an orchestrator, or with some other VM focused orchestrator.
- We want to use a container native orchestrator
- We want to use a cloud provider to avoid knowing details of Kubernetes setup (and most kubernetes administration)
- We want to focus on using kubernetes.
- Our webapp is a standard three tier app
    - Web browser is the very simple presentation layer
    - The webserver is the business logic layer
        - This is the one that should be 12 factor style
    - Redis is the data storage layer
- We want to first get a feel for running the webapp by itself, and then hook it up to the services it needs, and expose it to the user.
- Once we have the webapp up and running, we'll show how to do some basic scaling.



Lab

Step 1: Get code from GITHUB link

Before we begin the exercise, please download the web application code from https://github.com/IBMDevConnect/DockerK8sIBMCloudPrivateWorkshop

Open a terminal on your local computer.

using the following command:

$ git clone https://github.com/IBMDevConnect/DockerK8sIBMCloudPrivateWorkshop.git

3-Tier Web Application

The application we're going to be using is a simple 3-tier application. It has a webpage that talks to an application, that's written in python. This application then then talk to a backend redis server to count the number of hits to the application. We're mostly ignoring any complexities in the presentation layer (webpage), and using the web-browser simply to access the output of the business logic tier.

Let's first take a quick look at the application itself (go to directory where you cloned the code base), and notice that will both show a very simple web page (see the "return" line) as well as increment the "hits" counter (see the "redis.incr" call):

$ cat web/app.py

```
from flask import Flask
from redis import Redis
import os
import platform
app = Flask(__name__)
redis = Redis(host=os.environ.get('REDIS_HOST', 'redis'), port=6379)

@app.route('/')
def hello():
    redis.incr('hits')
    return 'Hello Container World from %s! I have been seen %s times.\n' %  (
platform.node(), redis.get('hits') )

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

Build the Application into a Docker Image

We have an application, but it needs to be packaged into a Docker image before we can deploy it. We do that using a file called “Dockerfile” as input to the Docker build process. A Dockerfile is a list of instructions for the Docker “builder” component that tells it how to construct the filesystem that will make up a container. Once that container has all of the files needed, it will then save the file system into an “image” that we can then deploy later. The concept is very similar to how you might build, and share, virtual machine images.

Step 2: Build image from a Docker File

$ cat web/Dockerfile
FROM python:2.7-onbuild
EXPOSE 5000
CMD ["python", "app.py"]

We won’t go into the details of what the various Dockerfile commands mean, but just know that the net result of the Docker “builder” running these commands will be a new Docker image that contains our application and the python runtime needed to execute our application.

If you do not have it yet, you can install docker from here: https://docs.docker.com/engine/installation/ . Before we run the Docker “build” command though we need to make sure that we give our new image a name so we can refer to it later. For our purposes we’re going to call it “webapp” but we also need to give it a “namespace” value so that when we upload it to a registry it will be stored in a location that is just for us. Meaning, each user will have their own “namespace” into which they can store images.

If you do not have a Docker ID account, you can set it up as follows: https://docs.docker.com/docker-id/ . With that we can now build the image using the following command (replace “<namespace>” with your Docker ID account name):

$ cd web

$ docker build -t <namespace>/webapp .

We can verify that it worked by looking at the list of Docker images that are available to our local Docker engine:

Step 3: verify image created

$ docker images

Notice the “webapp” image.

The image that we just built is only available to our local Docker engine. In order for us to use it elsewhere, like in our IBM Cloud hosted Kubernetes cluster, we need to upload it ("push" it) to the registry:

Step 4: Login to Docker Hub

$ docker login

Step 5: Push image to Docker Hub

$ docker push <namespace>/webapp

Now the Docker image is available for use outside of your machine.

Step 6: Run our first container

Run your container locally

$ docker run -d -p 5000:5000 mdpatankar/webapp

Use the docker container run command to run a container with the image created,

In the new terminal, use the docker container ls command to get the ID of the running container you just created.

$ docker container ls

CONTAINER
ID IMAGE COMMAND CREATED STATUS PORTS N
AMES
8cdc19312eed mdpatankar/webapp "python app.py" 3 minutes ago Up 3
minutes 0.0.0.0:8000->5000/tcp cranky_hopper

Then use that id to run bash inside that container using the docker container exec command. Since we are using bash and want to interact with this container from our terminal, use the -it flag to run using interactive mode while allocating a psuedo-terminal.

$ docker container exec -it b3ad2a23fab3 bash
root@b3ad2a23fab3:/#

We just used the docker container exec command to "enter" our container's namespaces with our bash process. Using docker container exec with bash is a common pattern to inspect a docker container.

Notice the change in the prefix of your terminal. For example `root@b3ad2a23fab3:/`. This is an indication that we are running bash “inside” of our container.

Note: This is not the same as ssh’ing into a separate host or a VM. We don’t need an ssh server to connect with a bash process. Remember that containers use kernel-level features to achieve isolation and that containers run on top of the kernel. Our container is just a group of processes running in isolation on the same host, and we can use docker container exec to enter that isolation with the bash process. After running docker container exec, the group of processes running in isolation (in other words, our container) include top and bash.

From the same terminal, run ps -ef to inspect the running processes.

```
root@b3ad2a23fab3:/# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 20:34 ?        00:00:00 top
root        17     0  0 21:06 ?        00:00:00 bash
root        27    17  0 21:14 ?        00:00:00 ps -ef
```

You should see only the \ bash process, and our ps process.

For comparison, exit the container, and run ps -ef or top on the host. These commands will work on Linux or Mac. For Windows, you can inspect the running processes using tasklist.

```
root@b3ad2a23fab3:/# exit
exit
$ ps -ef
# Lots of processes!
```

You have ran a web applications, web part. Still redis container is not up
If you try to run your application, will show below error

http://localhost: 5000

Step 7: Use docker compose to use database service with web

You will use Docker compose to run web and redis server.
If you see
Docker-compose file

```
version: '3'
services:
  web:
    build: .
    ports:
     - "5000:5000"
  redis:
    image: redis
```

This uses web : which is Dockerfile for building web portion
And redis as a backed database

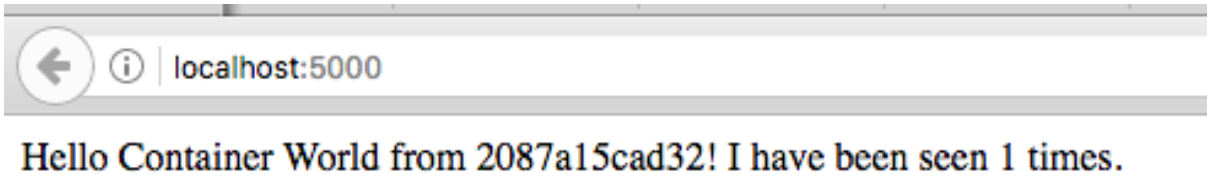
Run docker compose command from directory

$ docker-compose up

you will see below log…

```
Starting web_web_1   ... done
Creating web_redis_1 ... done
Attaching to web_web_1, web_redis_1
redis_1  | 1:C 05 Jun 18:10:06.338 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
redis_1  | 1:C 05 Jun 18:10:06.338 # Redis version=4.0.9, bits=64, commit=00000000, modified=0, pid=1, just started
redis_1  | 1:C 05 Jun 18:10:06.338 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
redis_1  | 1:M 05 Jun 18:10:06.339 * Running mode=standalone, port=6379.
redis_1  | 1:M 05 Jun 18:10:06.339 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
redis_1  | 1:M 05 Jun 18:10:06.339 # Server initialized
redis_1  | 1:M 05 Jun 18:10:06.339 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
redis_1  | 1:M 05 Jun 18:10:06.339 * Ready to accept connections
web_1    |  * Serving Flask app "app" (lazy loading)
web_1    |  * Environment: production
web_1    |    WARNING: Do not use the development server in a production environment.
web_1    |    Use a production WSGI server instead.
web_1    |  * Debug mode: on
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
web_1    |  * Restarting with stat
web_1    |  * Debugger is active!
web_1    |  * Debugger PIN: 554-986-091
web_1    | 172.19.0.1 - - [05/Jun/2018 18:10:13] "GET / HTTP/1.1" 200 -
```

Step 8: Application running sucessfully
Now run
http://localhost: 5000
∂
You should see,



As you go on hitting the application, hits are incremented

Congratulations!!! You have completed First Lab

More on Lab

You can try to get into container and check redis database

```
Mangeshs-Air-2:~ mangeshpatankar$ docker container ls
CONTAINER
ID        IMAGE          COMMAND                  CREATED          STATUS          PORTS                    NAMES
90e0d6ea9ce5        redis          "docker-entrypoint.s…"   6 minutes ago       Up 7
minutes         6379/tcp                 web_redis_1
2087a15cad32        web_web        "python app.py"          24 minutes ago      Up 7 minutes         0.0.0.0:5000-
>5000/tcp   web_web_1
```

```
Mangeshs-Air-2:~ mangeshpatankar$ docker container exec -it 90e0d6ea9ce5 bash
```

You can check

```
root@90e0d6ea9ce5:/data#
```

```
root@90e0d6ea9ce5:/data# redis-cli
127.0.0.1:6379>
127.0.0.1:6379> ping
PONG
```

```
127.0.0.1:6379> get hits
"1"
```

Go on hitting URL and you will see database update

```
127.0.0.1:6379> get hits
"3"
```

More to Read

Technical deep dive

PID is just one of the Linux namespaces that provides containers with isolation to system resources. Other Linux namespaces include:

- MNT – Mount and unmount directories without affecting other namespaces.
- NET – Containers have their own network stack.
- IPC – Isolated interprocess communication mechanisms such as message queues.
- User – Isolated view of users on the system.
- UTC – Set hostname and domain name per container.

These namespaces together provide the isolation for containers that allow them to run together securely and without conflict with other containers running on the same system. Next we will demonstrate different uses of containers. and the benefit of isolation as we run multiple containers on the same host.

Note: Namespaces are a feature of the *Linux* kernel. But Docker allows you to run containers on Windows and Mac… how does that work? The secret is that embedded in the Docker product is a Linux subsystem. Docker open-sourced this Linux subsystem to a new project: LinuxKit. Being able to run containers on many different platforms is one advantage of using the Docker tooling with containers.

In addition to running Linux containers on Windows using a Linux subsystem, native Windows containers are now possible due the creation of container primitives on the Windows OS. Native Windows containers can be run on Windows 10 or Windows Server 2016 or later.