Deploying MEAN stack application with Docker and Kubernetes

Overview

Our goal is to demonstrate a thin line flow though the process of delivering an application from development to production Kubernetes cluster.

By "thin line", I mean a process that is realistic, addresses many of the challenges and manages risks, but is not necessary fully fleshed out. I want this demo to be done in one sitting. Where i take shortcuts, I will try to note it.

Steps:

1. Dockerize your app

   Steps for taking a working application and publishing it to a Docker repository.

2. Run with Kubernetes

   Deploy the application, the database and related to services to our cluster.

Docker demo

Overview

To run this demo, we wil use VirtualBox and Vagrant.

While links below include all the steps to build the environment from scratch, we have also built an image you can use out of the box.

Prep:

In the "Prep" phase we setup a base machine with everything you need as well as instructions for using the prebuilt box.

1. Build machine setup

Steps:

1. Test the app
2. Build Docker image
3. Run container
4. Publish image

Next step: Build machine setup

Build machine setup

Back to

Overview

In order to "dockerize" our application we need a machine with a few prerequisites:

- we need the application itself
- we need Node to run application API
- we need Docker to create the image and run the container

We will execute these steps in a virtual machine running on VirtualBox and we'll use Vagrant to manage our images and VMs.VirtualBox and Vagrant will need to be installed on your machine.

Setup VirutalBox Docker Build VM from scratch

1. go to "from scratch"

   ```
   cd Vagrant-build/from-scratch
   ```

2. launch the virtual machine from this directory (uses Vagrantfile settings)

   ```
   vagrant up
   ```

3. Install tools (git, node, docker)

   i. connect to VM using vagrant

      ```
      vagrant ssh
      ```

   ii. run install scripts

      ```
      cd /vagrant
      ```

   iii. install git, wget, unzip,node:

      ```
      ./01_setup-build-server.sh
      ```

   iv. install docker

```
./02_install-docker.sh
```

4. relogin to update group membership and avoid using sudo every time we invoke the "docker" command

```
exit
vagrant halt
vagrant up
vagrant ssh
```

5. download docker images we will use to construct our application container (node base container)

```
./03_prep-docker-images.sh
```

From prebuilt box image:

1. go to "prebuilt" directory

```
cd Vagrant-build/prebuilt
```

2. download box image from S3 (1.7 GB will take a while to download)

```
curl https://s3.amazonaws.com/2017mongodb/Vagrant/build-mongodb-2017.box
```

3. import into virtual box

```
vagrant box add ./build-mongodb-2017.box --name polyakov/docker-k8s-build --force
```

4. launch

```
vagrant up
```

Next step: Run application

Run the app to test it

Overview

In the prior step, we setup a virtual machine with node and our application. Now it's time run it to make sure it works. Vagrantfile instructed the VM to forward port 3000 to the host, so we will be able to access the application on http://localhost:3000.

Our application uses MongoDB. For this test we'll use mLabs sandbox.

Prerequisites

1. You shoud have a MongoDB connection string. For purposes of the demo we are using mLab sandbox. Any MongoDB instance will do.

Run the app

1. ssh to the VM

   ```
   vagrant ssh
   ```

2. Run the app using node. This will start the app on port 3000

   ```
   export MONGO_CONNECTION_STR=<connectionstring>
   cd ~/demoapp/app/server
   node ./index.js &
   ```

3. Test the app in the VM using curl. You should see valid HTML.

   ```
   curl localhost: 3000
   ```

4. Test the app using forwarded port from host browser at http://localhost:3000

Next step: Build a Docker image

Build application Docker image

Overview

We have working app that we want to package in the docker container. To do this we need to give `docker build` command instructions on how to includes everything the app needs to run.

Build the image

1. ssh to the VM

   ```
   vagrant ssh
   ```

2. Setup docker file and view it

   ```
   cp /vagrant/Dockerfile-from-node ~/Dockerfile
   more ~/Dockerfile
   ```

   The Dockerfile is fully commented.

3. Build the image

   ```
   cd ~
   docker build . --file Dockerfile --tag polyakov/mongodb-demoapp-2017
   ```

4. Check for new image; there should be one tagged with "polyakov/mongodb-demoapp-2017"

   ```
   docker images
   ```

Next step: Run application with Docker

Run application as a Docker container

Overview

In this step we run both MongoDB and the application using docker. First we pull the official MongoDB docker image and then run both application and DB usinder docker

Run the application with Docker

1. Pull official MongoDB image from Docker Hub
   - https://hub.docker.com/_/mongo/.

   ```
   docker pull mongo
   ```

2. check images for mongo image

   ```
   docker images
   ```

3. launch database

   ```
   docker run --detach --name=2017mongo_db  mongo
   ```

   options:

   `--detach` - run as daemon
   `--name` - provide a meaningful name for the container. Otherwise docker will assign one.

4. check that the container is running by listing all running containers

   ```
   docker ps
   ```

5. get container details; note container's IP

   ```
   docker inspect 2017mongo_db
   ```

6. now that our database is running, let's connect the application:

   ```
   docker run --detach \
       --name=2017mongodb_app \
       -e "MONGO_CONNECTION_STR=mongodb://172.17.0.2:27017/demo" \
       -p 8080:8080 \
       polyakov/mongodb-demoapp-2017
   ```

options:

`-e` set environment variable for the container
`-p` forward port to the host

7. check that the application is running by accessing the container directly

    i. get container IP

    ```
    docker inspect 2017mongodb_app
    ```

    ii. hit container directly

    ```
    curl 172.17.0.3:8080
    ```

    iii. check application content in the VM; we are forwarding ports from docker conctainer to the VM.

    ```
    curl localhost:8080
    ```

    iv. hit application from host machine; we are forwarding port from VM to the host by access http://localhost:8080 from your browser.

Next step: Publish image

Publish Docker image

Overview

Once we have an image we want to store it in a way that allows our servers to access it for deployment. In the example, we are using Docker Hub. Amazon, Azure and others offer private registries in the cloud. It's also not very hard to establish a try a private repo on prem.

Publish the image

1. List images

   ```
   docker images
   ```

2. Tag the image with version. The first part of the image name, "polyakov", should match your docker hub image name, the second part is the image name and the third part, after the colon, is he tag.

   ```
   docker tag polyakov/mongodb-demoapp-2017 polyakov/mongodb-demoapp-2017:20170602
   ```

3. login to Docker hub

   ```
   docker login
   ```

4. And, push

   ```
   docker push polyakov/mongodb-demoapp-2017:20170602
   ```

5. Confirm that image have been uploaded by visting https://hub.docker.com/r/polyakov/mongodb-demoapp-2017/

Docker demo done!!

Next step: Kubernetes demo

Kubernetes Demo

Overview

The goal for this demo is to show how easy it is to get started deploying an application stack with kubernetes.

We'll start with the database - MongoDB. We'll deploy a MongoDB replica set with two data nodes and an arbiter.

Then we'll add our application and expose it publicly.

Prep

If the cluster is running start with #4.

1. Setup your client
2. Spin up your cluster (AWS)
3. Configure the cluster

Steps

4. Pull config
5. Set your namesapce
6. Deploy MongoDB
7. Deploy App

Next step: Setup your client

Connect to cluster

Overview

In order to send commands to the cluster we need to get the right config and place it `~/.kube/` directory.

Steps:

1. Change directory to `~/.kube`
   ```
   cd ~/.kube
   ```

   If directory does not exists, create it.

   ```
   mkdir ~/.kube
   ```

2. Download the config

   ```
   wget https://s3.amazonaws.com/mongodb-2017/config
   ```

3. Check that you can access the cluster by listing nodes.

   ```
   kubectl get nodes
   ```

Set Your Namespace

Overview

Kubernetes uses namespaces to partition the environment. In this set of steps, we'll create the namespace and set it as default.

Steps:

1. Set namespace as a env variable. **Change the namespace to your own name**

   ```
   export NAMESPACE=polyakov3519
   ```
2. Create namespace

   ```
   kubectl create namespace ${NAMESPACE}
   ```
3. Set your namespace default

   ```
   kubectl config set-context $(kubectl  config  current-context) --
   namespace=${NAMESPACE}
   ```
4. Check to see that the namespace has been set

   ```
   kubectl config view | grep namespace:
   ```

Deploy MongoDB

Overview

In this step, we are starting to create components. We'll do this step by step. However, it is also convenient (though less educational) to create all the components at once by passing in the entire component directory. `kubectl create -f ./create`
We are using "naked" pods, not managed by a ReplicaSet because we don't want them to restart automatically. If our database went down we want someone to figure out why.

For the purposes of this demo we are using "emotyDir" volume. In production MongoDB deployment we would want to use Persistent Volumes appropriate for our environment.

Steps:

1. Create all MongoDB components in one command

   ```
   kubectl create -f ./create/mongodb
   ```

   Skip to step #2 below.

Or,

1. Create components one at a time
   i. Create MongoDB pods

      ```
      kubectl create -f ./create/mongodb/pod-mongo-db1.json
      kubectl create -f ./create/mongodb/pod-mongo-db1.json
      ```

      check pods

      ```
      kubectl get pods
      ```
   ii. Create arbiter

      ```
      kubectl create -f ./create/mongodb/rc-arb.json
      ```
      check replication controllers

      ```
      kubectl get rc
      ```
   iii. Create services

```
kubectl create -f ./create/mongodb/svc-arb.json
kubectl create -f ./create/mongodb/svc-mongo-db1.json
kubectl create -f ./create/mongodb/svc-mongo-db2.json
```

check services

```
kubectl get svc
```

2. Once pods are running we can configure the MongoDB replica set.
   i. connect to first pod

   ```
   kubectl exec -ti mongo-db1 mongo
   ```

   Syntax is same as `docker exec`.

   ii. Initialize MongoDB RS. **Note: Set your namespace in "host" value.**

   ```
   rs.initiate(
   {
           "_id" : "demoRS",
           "version" : 1,
           "protocolVersion" : NumberLong(1),
           "members" : [
                   {
                           "_id" : 0,
                           "host" : "mongo1-
   svc.<namespace>.svc.cluster.local:27017",
                           "arbiterOnly" : false,
                           "buildIndexes" : true,
                           "hidden" : false,
                           "priority" : 1,
                           "tags" : {

                           },
                           "slaveDelay" : NumberLong(0),
                           "votes" : 1
                   }
           ]})
   ```

   iii. Add second node to config. **Note: Set your namespace in "host" value.**

   ```
   rs.add("mongo2-svc.<namespace>.svc.cluster.local:27017")
   ```

   iv. Add the arbiter. **Note: Set your namespace in "host" value.**

   ```
   rs.addArb("arbiter-svc.<namespace>.svc.cluster.local:27017")
   ```

   v. Check status

   ```
   rs.status()
   ```

Deploy app

Overview

Adding our app to cluster by deploying a RC and service with the the application pod we crated previously.

Steps:

1. Deploy app in one command by invoking `kubectl create` with a directory argument.
   ```
   kubectl create -f ./app
   ```

Or,

1. Deploy components one at a time

   i. Deploy Replication Controller

   ```
   kubectl create -f ./app/rc-app.json
   ```

   ii. List services

   ```
   kubectl get rc
   kubectl get pods
   ```

   iii. Deploy service

   ```
   kubectl create -f ./app/svc-app.json
   ```

   iv. List services

   ```
   kubectl get svc
   ```

2. Get service details, in partucular note the port of the service

   ```
   kubectl describe svc app-api-svc
   ```

3. List nodes to get an IP address. Pick any node and note its public IP address or DNS name (e.g. ec2-54-210-108-3.compute-1.amazonaws.com)

   ```
   kubectl describe nodes
   ```

4. Craft a URL from IP and port acquired in the prior two steps and check that the site is up.