# Kubernetes Deployment Project

*Brad - ASP.NET Core dockerization and deployment on Heroku*

*https://dev.to/alrobilliard/deploying-net-core-to-heroku-1lfe*

If you have already run your proposed by me, please type it into a word/text document and upload it hear.  Please include the following in your proposal:

1. A short paragraph outlining the MVP (Minimal Viable Product) that you will be demonstrate by the specified due date.

The MVP for my project will be a hosted ASP.NET Core application that is containerized and deployed with Kubernetes. We will include security services and mass deployment, so scalability is achieved. We will begin by creating an image of the application through Docker, Docker Client and DockerHub. We will then deploy our image/container with Kubernetes adding services and

1. Risks (specially if your project is experimental)
   1. Deployment errors
      1. Kubernetes resetting
   2. Image creation
   3. Services do not correct for Kubernetes scalability.
   4. Image not built correctly.
2. Mock ups of your Graphical User Interface.
3. Deployment architecture/diagram.

* Create Application
* Create image to docker hub

1. List of (high priority) functional and non-functional requirements
2. A backlog of tasks along with their due dates (and to whom the task has been assigned in case you are doing the project in pairs) that will help you achieve your MVP by the due date.
3. Runtime / Development environment requirements e.g. SDK, programming language, host platform.
4. What features/attributes make your product distinct/novel.  What new you will be learn due to this project.

## Part 1: Docker and Kubernetes Relationship

Steps:

* 1. *Download Docker Client*
     1. <https://www.docker.com/products/docker-desktop/>
  2. *Signup on DockerHub*
     1. <https://hub.docker.com/signup>

### Containers/Images

* **Image**
  + Single file with all dependencies and configurations required to run a program.
  + Saved on Docker Hub
  + Gets stored on hard drive.
  + Can be used to create a container.
* **Container**
  + An instance of an image
  + A running program
  + Like a thread
  + Has isolated set of resources.

### How it works “docker run hello-world”

1. Docker command “docker run hello-world”
   1. Docker run means we want to create a new container from the image “hello-world”
   2. Docker image has a program in it.
   3. Docker server saw we were trying to start up new container, checked for a local copy on machine (image cache: will be empty if first installed)
   4. Server reaches out to docker hub, repo of free public images, finds the image hello-world and stores it in the image cache(to be re-run easily at any point now)
   5. Create an instance of a container from the image.
      1. Instance of the image
      2. Runs one specific program.
   6. Docker saves to the cache when you download an image

### Containers revisited

* OS
  + Processes make system calls to the kernel
  + Kernel governs access to the CPU, Memory and Hard Disk etc
  + Process talks to kernel to perform task, kernel does the work
  + Processes interact with system calls.
  + Kernel exposes endpoints, takes info and performs tasks.
  + Name spacing: create segments on hard disk to housing different software, when system call is made, the kernel looks at the system call and places that one process is trying to access certain information from one of the one segments and will redirect there
    - Isolating resources per group of process or processes
  + Control groups
    - Can limit resources used by processes
      * Can limit memory, CPU usage, IO and Network Bandwidth
  + A container is not a physical construct, it is a process that has a grouping of resources specifically assigned to it
    - Container
      * Process makes system call.
      * Kernel receives call.
      * Directs to specific resources such as CPU or Hard drive.
      * Made available to the process.
    - Images
      * Is a file system snapshot?
      * Copy pastes of files or directory
      * Image can contain python or java etc. and a startup command.
      * When we containerize an image
        + Kernel isolates small section of hard drive for container.
        + File snapshot in image is placed in the hard drive segment.
  + Namespaceing and control groups are specific to LINUX OS not to windows or MAC
    - how is docker running then?
    - Docker runs a LINUX VM!
    - All the containers are created in VM
      * Has a LINUX Kernel
      * Host running processes in the VM
      * ./DockerCli.exe -SwitchDaemon
        + will switch to Linux if ran in PowerShell
      * being used to host containers

### Docker CLI Commands

* docker run <image name>
  + creates AND runs a container.
* docker run <image-name> <alt-command>
  + the alt command will override the command in the image (located in the dockerfile)
* docker ps
  + will list all the current running containers on your machine.
* docker ps –all
  + gets all containers ever created.
* docker run = docker create + docker start
  + create makes the container from the image.
  + start begins the container.
    - runs the startup command.
* docker create <image name>
  + will return an ID for the container.
* docker start -a <id>
  + will run the container.
  + -a attach for container and watch for output and print on my terminal, give me any output from the container
* Docker system prune
  + Will delete all containers and cache that are idle.
* Docker logs <container-id>
  + Will print what the output was of the container as a log.
  + Does not rerun the container, logs it during runtime and allows us to access and see them.
* Docker stop <contained id>
  + Send a SIGTERM message to container.
  + Received from process.
  + Telling it to shutdown
  + Stops a process, gives the process a second to shutdown and clean up a bit
    - Save files, emit message etc
  + If the container doesn’t stop in 10s it will automatically issue docker kill
* Docker kill <container id>
  + Issues a SIGKILL (kill signal) to primary process.
  + Shuts down RIGHT AWAY
  + No additional work given.
  + Only use if it is not responding to the stop command.
* Docker exec -it <container id> <command>
  + Allows us to execute an additional command in a container.
  + It allows us to type input directly into container.

### Redis CLI

* How to execute second command inside of running container
* Docker run redis
  + Will grab the redis image, a server and beging running it
  + In order to attack the cli to the container we must use the exec function
* Redis clis
  + Docker ps to get the container image for redis
  + Docker exec -it <image id> redis-cli
    - Will run the cli in the same container.
    - Started a second program in the container.
    - -it allowed us to input commands
* -it flags
  + Every container is running inside of Linux VM
  + All processes have 3 communication channels
    - STDIN, STDOUT, STDERR
    - Communicates information into, or out of the process.
    - STDIN is what we type, OUT is what shows up in the terminal in the VM, ERR will redirect to show us on our terminal
  + -it is two separate flags
    - -I and -t
      * -I means if we execute new command we want to attach our terminal to the STDIN process, makes sure what I type goes to the VM cli
      * -t makes sure all text that you enter in and receive is nicely formatted on the screen

### Shell access

* Want to run commands inside your container without using exec command constantly
* Docker exec -it <contained id> sh
  + Sh is a program being executed in container
  + It is a shell
  + Allows us to type commands in and execute them in the VM
  + When we start sh its another command shell to execute commands

### Container Isolation

## Docker Images

### Dockerfile

* Plain text file with configuration
* Defines how the container behaves
* What different programs it contains and how it acts son startup
* Pass the file to the docker client (CLI)
* Client provides file to the docker server
  + Takes the dockerfile and builds a usable image from the configuration
  + All complexity in the dockerfile
* How to write
  + Specify base image
  + Run somes commands
  + Specify a startup command for the image

### Building dockerfile

* Building an image that runs redis server when it starts up
* Writing the dockerfile
  + For our base redis image we will do the following



* + **FROM alpine**
    - Specifies the docker image as a base we are using.
    - The base image is alpine.
    - Like installing something.
  + **RUN apk add –update redis**
    - Executes a command while image is being prepared.
  + **CMD [“redis-server”]**
    - Specifies what should be executed when our image is used to startup a new container.
  + Run the command on terminal in the same directory as Dockerfile
    - **Docker build .**
    - Will build the file
    - Docker run <image id>
* Why alpine as the base image
  + Why use windows or ubuntu etc.
    - come pre-installed with programs etc.

## Build Process

* each line is a step.
  + pulling an image will check the cache first.
  + intermediate containers
    - creates a temp container when we get the image
    - has the fs snapshot, just created for the next instruction
    - each step creates an intermediate container for each step in the image
    - creates container on each step, takes a snapshot of the fs and changes the ID and moves to the next step

### Cache Rebuilds

* each build saves to the cache
* if you keep building an image the build process will grab from the cache for the image
  + that is each image from each command
* if the series of steps changes it will only use the the cache for those commands in order, the rest will be grabbed again

### Tagging Images

* docker build -t <tag> .
* the convention for the tag goes as follow
  + docker id/ project name:version
  + ie bradmasy/Kubernetes:latest
* finished comman
  + docker build -t bradmasy/redis:latest

### Manual Image Generation and Docker Commit

* we can create an image from a container
* first we docker run -it alpine sh to get a shell into an alpine image
* from here we can specify the commands from the previous dockerfile and get a container ID, we can then call that ID container like so
  + docker commit -c "CMD 'redis-server'" <container-id>

# Node.JS Application

1. make a folder and create a package.json file in the main directory
   1. dependencies
      1. everything the app needs to run
   2. scripts
      1. start
         1. what will run on startup
2. index.js
   1. server logic
   2. use express
3. node apps
   1. need to install npm
      1. npm install
   2. start server by calling npm start
   3. dockerfile for node minimum
      1. FROM node:14-alpine # stripped down image of alpine but has node and npm, can find with tags on docker hub
      2. RUN npm install
      3. CMD [“npm”, “start”]
   4. Theres a problem
      1. When we create a container, the container is a snapshot of the Linux file system, it does not have our files. We need to **COPY** our files into the container with the **COPY** command in the docker file
      2. **COPY**
         1. Copies into container
      3. **./**
         1. Path to folder to copy from on YOUR machine, relative to build context
         2. build context is the . in docker build  **.**
      4. **./**
         1. Path we want to copy to in the container
      5. Final command
         1. **COPY ./ ./**

## Putting on local host

* In order to make sure we can host the container we need setup explicit port mapping
* Forward the request to some port in the container
  + If someone makes a req for localhost 8080 forward to the containers port 8080
  + This is INCOMING requests
  + Docker container can make request to outside world
    - Ie installing dependency such as npm
  + To setup port mapping
  + Docker run **-p 8080<port on local>** :  **8080{port in container}** <image id>
    - The first port is for the local machine for someone on the browser, we are saying when anyone is trying to access port 8080 we will forward them to the next one specified in the container
    - The next 8080 is in reference to the port in the container that the application is being hosted on
    - These ports **DO NOT HAVE TO BE IDENTICAL**
    - The second one **HAS TO MATCH THE PORT BEING SERVED IN THE APPLICATION**

## Copying and Working Directories

* Although the copy command works great for getting our application into the container what if we have files with the same name, we should be copying the directory in instead of instances of the files etc
* We can use the instruction
  + **WORKDIR /{reference}**
  + Any commands added will be relative to the working directory
  + If the folder reference does not exist ie if we do **WORKDIR /usr/app**  docker will create a folder for us with that reference in the container
  + Usr location is a safe directory to put application

## Better ways to copy folders and directories into containers

* Split copy operation into 2 steps
  + Npm just needs package.json first
* COPY ./package.json ./
  + Now we will just copy this file into the current working directory of the container
* FROM node:14-alpine
* WORKDIR /usr/app
* COPY ./package.json ./
* RUN npm install
* COPY ./ ./
* CMD ["npm", "start"]
* The finished file
* The second copy will grab the rest of the files, this means when we want to make a change to source code we do not have to reinstall NPM because it will be cached
* If we make a change to the package.json file it **WILL** have to copy everything again(except the image FROM command)

## Web App 2

## COMPONENTS

* Web server
  + Node application
* Redis
  + Server
  + In memory data store
  + Tiny database
  + Will contain number of times page visited

## Scalability

* Lots of traffic to website
* As we get more traffic were going to need more servers
* In order to make more servers we might make more container’s with the server and node app
  + The problem with this is they will all be disconnected, have their own number of visits when in truth it’s the combination of them all
  + We want one single instance we can scale the node server not the app
* We will make
  + Separate containers for the node app and the redis server
* Package.json
  + Dependencies
    - Express: “\*” // any version
      * For the app
    - Redis:2.8.0
      * The server
* Index.js
* onst express = require ('express');
* const redis   = require ('redis');
* const app     = express();
* const client  = redis.createClient();
* client.set("visits", 0 );
* app.listen(3000,()=>{
* console.log('Server started on port 3000');
* })
* app.get("/",(req,res)=>{
* client.get("visits",(err,visits)=>{
* res.send("Number of visits is "+ visits);
* client.set("visits", parseInt(visits) +1 );
* })
* })

Create a dockerfile for the image of node

## Separate container running redis

* Docker run redis
* We need node app to communicate to redis server container
* Network functionality
  + Docker cli network
  + Docker compose cmd
    - This is the better of them

## Docker-compose

* Used to start multiple docker containers at the same time
* So we don’t have to keep running repetitive commands with docker
* Functions as Dockers CLI
* How it works
  + We will encode our commands in a yaml fle and then compose it
* Docker-compose.yml
  + Version:’3’ // is the version of compose we are using
  + Services:
    - A container essentially
    - A type of container
    - We will specify we want a redis server container and a node-app
  + Yml is vert specific to indentation
    - 2 space bars

version: '3'

services:

  redis-server:

    image: 'redis'

  node-app:

    build: .

    ports:

      - "4001:8081"

Docker compose will create these two containers on the same network and be able to communicate together

No port declaration needed

* In order to specify the host of the redis server we use the container reference from the yml file
* Host:”redis-server” will specify that the host will b the redis server container
* Default port number for redis is 6379
* When node app starts
  + Will try for a connection to redis server
  + It will reach out for a host of redis-server
  + Docker will redirect to the redis-server container
    - Automatically connects the containers

## Docker compose up

* To create an instance of all the images/services in compose
  + Docker-compose up
    - No image specified
    - It will see the images in the docker-compose file
* Prior there was 2 commands to get an image running
  + Docker build and docker run
  + To execute both of these we use
    - Docker-compose up –build

## Containers in Background

* Docker-compose up -d
  + Launches containers in the background
* Docker-compose down
  + Stops the containers

## Maintenance with Compose

* Restarting containers when software/containers have errors
* How to restart a container automatically
  + Restart policy
    - 4 polocies we have access to
    - “no”
      * If theres a crash do not attempt to restart
      * This policy HAS TO BE IN QUOTES
        + In yml NO means false, which is different etc
    - Always
      * If it stops try to restart it always
    - On failure
      * Only restart if the container stops with an error code
    - Unless-stopped
      * Always restart unless we (devs) forcibly stop it
  + Thsese policies must be applied to each container we would like to add it to

## Container Status with Docker Compose

* Docker-compose ps
  + Get the status of docker compose containers running

CMD ASPNETCORE\_URLS=http://\*:$PORT dotnet Web.dll