Learn to build and deploy your distributed applications easily to the cloud with Docker

Written and developed by Prakhar Srivastav

INTRODUCTION

What is Docker?

Wikipedia defines Docker as

> *an open-source project that automates the deployment of software applications inside* ***containers*** *by providing an additional layer of abstraction and automation of* ***OS-level virtualization*** *on Linux.*

Wow! That's a mouthful. In simpler words, Docker is a tool that allows developers, sys-admins etc. to easily deploy their applications in a sandbox (called *containers*) to run on the host operating system i.e. Linux. The key benefit of Docker is that it allows users to **package an application with all of its dependencies into a standardized unit** for software development. Unlike virtual machines, containers do not have the high overhead and hence enable more efficient usage of the underlying system and resources.

What are containers?

The industry standard today is to use Virtual Machines (VMs) to run software applications. VMs run applications inside a guest Operating System, which runs on virtual hardware powered by the server's host OS.

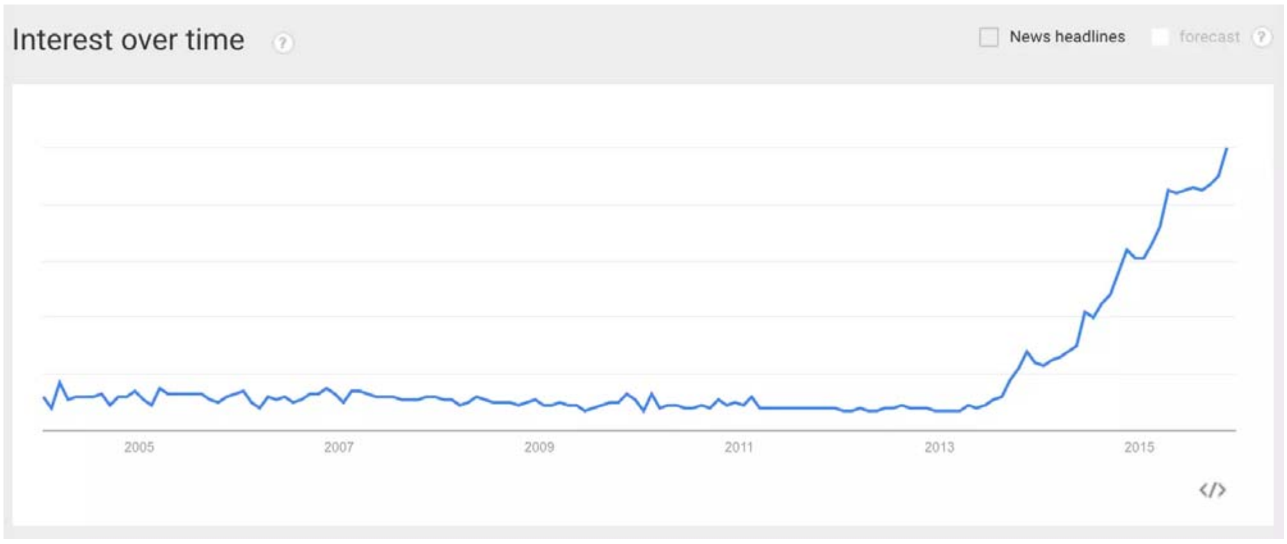
VMs are great at providing full process isolation for applications: there are very few ways a problem in the host operating system can affect the software running in the guest operating system, and vice-versa. But this isolation comes at great cost — the computational overhead spent virtualizing hardware for a guest OS to use is substantial.

Containers take a different approach: by leveraging the low-level mechanics of the host operating system, containers provide most of the isolation of virtual machines at a fraction of the computing power.

Why use containers?

Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer's personal laptop. This gives developers the ability to create predictable environments that are isolated from rest of the applications and can be run anywhere.

From an operations standpoint, apart from portability containers also give more granular control over resources giving your infrastructure improved efficiency which can result in better utilization of your compute resources.



Google Trends for Docker

Due to these benefits, containers (& Docker) have seen widespread adoption. Companies like Google, Facebook, Netflix and Salesforce leverage containers to make large engineering teams more productive and to improve utilization of compute resources. In fact, Google credited containers for eliminating the need for an entire data center.

What will this tutorial teach me?

This tutorial aims to be the one-stop shop for getting your hands dirty with Docker. Apart from demystifying the Docker landscape, it'll give you hands-on experience with building and deploying your own webapps on the Cloud. We'll be using Amazon Web Services to deploy a static website, and two dynamic webapps on EC2 using Elastic Beanstalk and Elastic Container Service. Even if you have no prior experience with deployments, this tutorial should be all you need to get started.

GETTING STARTED

This document contains a series of several sections, each of which explains a particular aspect of Docker. In each section, we will be typing commands (or writing code). All the code used in the tutorial is available in the Github repo.

> *Note: This tutorial uses version **18.05.0-ce** of Docker. If you find any part of the tutorial incompatible with a future version, please raise an issue. Thanks!*

Prerequisites

There are no specific skills needed for this tutorial beyond a basic comfort with the command line and using a text editor. Prior experience in developing web applications will be helpful but is not required. As we proceed further along the tutorial, we'll make use of a few cloud services. If you're interested in following along, please create an account on each of these websites:

- Amazon Web Services
- Docker Hub

Setting up your computer

Getting all the tooling setup on your computer can be a daunting task, but thankfully as Docker has become stable, getting Docker up and running on your favorite OS has become very easy.

Until a few releases ago, running Docker on OSX and Windows was quite a hassle. Lately however, Docker has invested significantly into improving the on-boarding experience for its users on these OSes, thus running Docker now is a cakewalk. The *getting started* guide on Docker has detailed instructions for setting up Docker on Mac, Linux and Windows.

Once you are done installing Docker, test your Docker installation by running the following:

```
$ docker run hello-world

Hello from Docker.
This message shows that your installation appears to be working correctly.
...
```

HELLO WORLD

Playing with Busybox

Now that we have everything setup, it's time to get our hands dirty. In this section, we are going to run a Busybox container on our system and get a taste of the `docker run` command.

To get started, let's run the following in our terminal:

```
$ docker pull busybox
```

> *Note: Depending on how you've installed docker on your system, you might see a `permission denied` error after running the above command. If you're on a Mac, make sure the Docker engine is running. If you're on Linux, then prefix your `docker` commands with `sudo`. Alternatively you can create a docker group to get rid of this issue.*

The `pull` command fetches the busybox **image** from the **Docker registry** and saves it to our system. You can use the `docker images` command to see a list of all images on your system.

```
$ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             VIRTUAL
busybox                 latest              c51f86c28340        4 weeks ago         1.109 MB
```

☰ Docker Run

Great! Let's now run a Docker **container** based on this image. To do that we are going to use the almighty `docker run` command.

```
$ docker run busybox
$
```

Wait, nothing happened! Is that a bug? Well, no. Behind the scenes, a lot of stuff happened. When you call `run`, the Docker client finds the image (busybox in this case), loads up the container and then runs a command in that container. When we run `docker run busybox`, we didn't provide a command, so the container booted up, ran an empty command and then exited. Well, yeah - kind of a bummer. Let's try something more exciting.

```
$ docker run busybox echo "hello from busybox"
hello from busybox
```

Nice - finally we see some output. In this case, the Docker client dutifully ran the `echo` command in our busybox container and then exited it. If you've noticed, all of that happened pretty quickly. Imagine booting up a virtual machine, running a command and then killing it. Now you know why they say containers are fast! Ok, now it's time to see the `docker ps` command. The `docker ps` command shows you all containers that are currently running.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
```

Since no containers are running, we see a blank line. Let's try a more useful variant: `docker ps -a`

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
305297d7a235        busybox             "uptime"            11 minutes ago      Exited (0) 1
ff0a5c3750b9        busybox             "sh"                12 minutes ago      Exited (0) 1
14e5bd11d164        hello-world         "/hello"            2 minutes ago       Exited (0) 2
```

So what we see above is a list of all containers that we ran. Do notice that the `STATUS` column shows that these containers exited a few minutes ago.

You're probably wondering if there is a way to run more than just one command in a container. Let's try that now:

```
$ docker run -it busybox sh
/ # ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
```

```
/ # uptime
 05:45:21 up  5:58,  0 users,  load average: 0.00, 0.01, 0.04
```

Running the `run` command with the `-it` flags attaches us to an interactive tty in the container. Now we can run as many commands in the container as we want. Take some time to run your favorite commands.

> ***Danger Zone**: If you're feeling particularly adventurous you can try `rm -rf bin` in the container. Make sure you run this command in the container and **not** in your laptop/desktop. Doing this will not make any other commands like `ls`, `echo` work. Once everything stops working, you can exit the container (type `exit` and press Enter) and then start it up again with the `docker run -it busybox sh` command. Since Docker creates a new container every time, everything should start working again.*

That concludes a whirlwind tour of the mighty `docker run` command, which would most likely be the command you'll use most often. It makes sense to spend some time getting comfortable with it. To find out more about `run`, use `docker run --help` to see a list of all flags it supports. As we proceed further, we'll see a few more variants of `docker run`.

Before we move ahead though, let's quickly talk about deleting containers. We saw above that we can still see remnants of the container even after we've exited by running `docker ps -a`. Throughout this tutorial, you'll run `docker run` multiple times and leaving stray containers will eat up disk space. Hence, as a rule of thumb, I clean up containers once I'm done with them. To do that, you can run the `docker rm` command. Just copy the container IDs from above and paste them alongside the command.

```
$ docker rm 305297d7a235 ff0a5c3750b9
305297d7a235
ff0a5c3750b9
```

On deletion, you should see the IDs echoed back to you. If you have a bunch of containers to delete in one go, copy-pasting IDs can be tedious. In that case, you can simply run -

```
$ docker rm $(docker ps -a -q -f status=exited)
```

This command deletes all containers that have a status of `exited`. In case you're wondering, the `-q` flag, only returns the numeric IDs and `-f` filters output based on conditions provided. One last thing that'll be useful is the `--rm` flag that can be passed to `docker run` which automatically deletes the container once it's exited from. For one off docker runs, `--rm` flag is very useful.

In later versions of Docker, the `docker container prune` command can be used to achieve the same effect.

```
$ docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
4a7f7eebae0f63178aff7eb0aa39f0627a203ab2df258c1a00b456cf20063
f98f9c2aa1eaf727e4ec9c0283bcaa4762fbdba7f26191f26c97f64090360

Total reclaimed space: 212 B
```

Lastly, you can also delete images that you no longer need by running `docker rmi`.

Terminology

In the last section, we used a lot of Docker-specific jargon which might be confusing to some. So before we go further, let me clarify some terminology that is used frequently in the Docker ecosystem.

- *Images* - The blueprints of our application which form the basis of containers. In the demo above, we used the `docker pull` command to download the **busybox** image.
- *Containers* - Created from Docker images and run the actual application. We create a container using `docker run` which we did using the busybox image that we downloaded. A list of running containers can be seen using the `docker ps` command.
- *Docker Daemon* - The background service running on the host that manages building, running and distributing Docker containers. The daemon is the process that runs in the operating system to which clients talk to.
- *Docker Client* - The command line tool that allows the user to interact with the daemon. More generally, there can be other forms of clients too - such as Kitematic which provide a GUI to the users.
- *Docker Hub* - A registry of Docker images. You can think of the registry as a directory of all available Docker images. If required, one can host their own Docker registries and can use them for pulling images.

WEBAPPS WITH DOCKER

Great! So we have now looked at `docker run`, played with a Docker container and also got a hang of some terminology. Armed with all this knowledge, we are now ready to get to the real-stuff, i.e. deploying web applications with Docker!

Static Sites

Let's start by taking baby-steps. The first thing we're going to look at is how we can run a dead-simple static website. We're going to pull a Docker image from Docker Hub, run the container and see how easy it is to run a webserver.

Let's begin. The image that we are going to use is a single-page website that I've already created for the purpose of this demo and hosted on the registry - `prakhar1989/static-site`. We can download and run the image directly in one go using `docker run`. As noted above, the `--rm` flag automatically removes the container when it exits.

```
$ docker run --rm prakhar1989/static-site
```

Since the image doesn't exist locally, the client will first fetch the image from the registry and then run the image. If all goes well, you should see a `Nginx is running...` message in your terminal. Okay now that the server is running, how to see the website? What port is it running on? And more importantly, how do we access the container directly from our host machine? Hit Ctrl+C to stop the container.

Well in this case, the client is not exposing any ports so we need to re-run the `docker run` command to publish ports. While we're at it, we should also find a way so that our terminal is not attached to the running container. This way, you can happily close your terminal and keep the container running. This is called **detached** mode.

```
$ docker run -d -P --name static-site prakhar1989/static-site
e61d12292d69556eabe2a44c16cbd54486b2527e2ce4f95438e504afb7b02810
```

In the above command, `-d` will detach our terminal, `-P` will publish all exposed ports to random ports and finally `--name` corresponds to a name we want to give. Now we can see the ports by running the `docker port [CONTAINER]` command

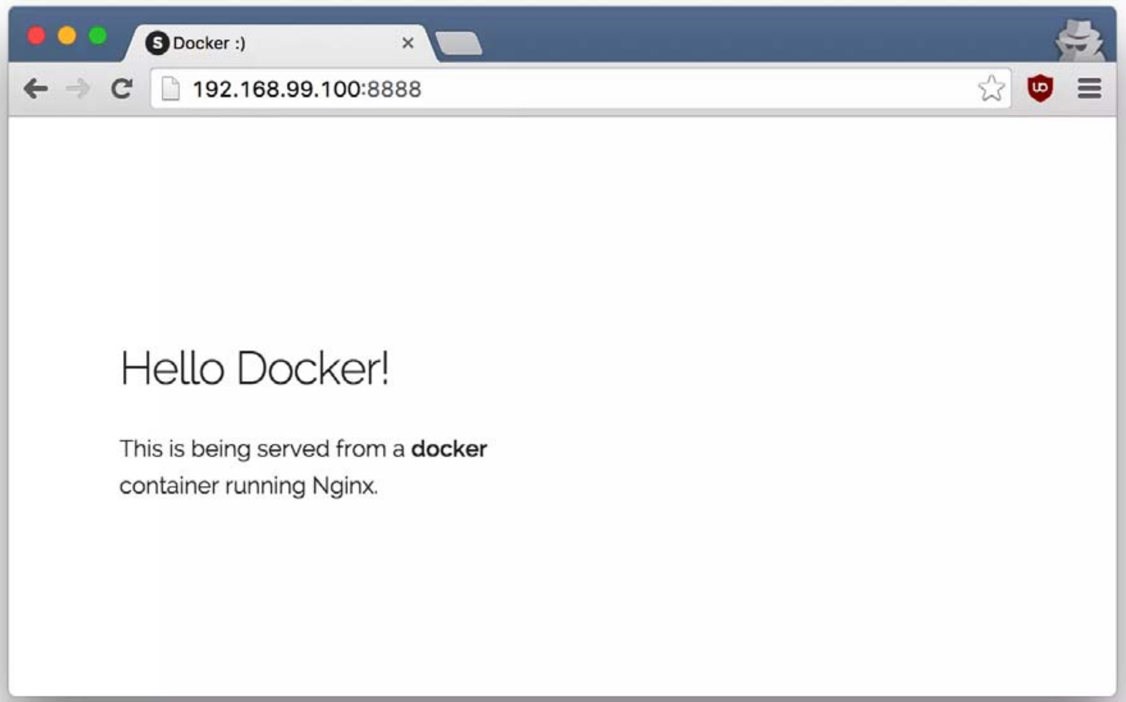
```
$ docker port static-site
80/tcp -> 0.0.0.0:32769
443/tcp -> 0.0.0.0:32768
```

You can open http://localhost:32769 in your browser.

> *Note: If you're using docker-toolbox, then you might need to use `docker-machine ip default` to get the IP.*

You can also specify a custom port to which the client will forward connections to the container.

```
$ docker run -p 8888:80 prakhar1989/static-site
Nginx is running...
```



Hello Docker!

This is being served from a **docker** container running Nginx.

To stop a detached container, run `docker stop` by giving the container ID. In this case, we can use the name `static-site` we used to start the container.

```
$ docker stop static-site
static-site
```

I'm sure you agree that was super simple. To deploy this on a real server you would just need to install Docker, and run the above Docker command. Now that you've seen how to run a webserver inside a Docker image, you must be wondering - how do I create my own Docker image? This is the question we'll be exploring in the next section.

Docker Images

We've looked at images before, but in this section we'll dive deeper into what Docker images are and build our own image! Lastly, we'll also use that image to run our application locally and finally deploy on AWS to share it with our friends! Excited? Great! Let's get started.

Docker images are the basis of containers. In the previous example, we **pulled** the *Busybox* image from the registry and asked the Docker client to run a container **based** on that image. To see the list of images that are available locally, use the `docker images` command.

```
$ docker images
REPOSITORY                      TAG                 IMAGE ID            CREATED
prakhar1989/catnip              latest              c7ffb5626a50        2 hours ago
```

```
prakhar1989/static-site        latest           b270625a1631        21 hours ago
python                         3-onbuild        cf4002b2c383        5 days ago
martin/docker-cleanup-volumes  latest           b42990daaca2        7 weeks ago
ubuntu                         latest           e9ae3c220b23        7 weeks ago
busybox                        latest           c51f86c28340        9 weeks ago
hello-world                    latest           0a6ba66e537a        11 weeks ago
```

The above gives a list of images that I've pulled from the registry, along with ones that I've created myself (we'll shortly see how). The `TAG` refers to a particular snapshot of the image and the `IMAGE ID` is the corresponding unique identifier for that image.

For simplicity, you can think of an image akin to a git repository - images can be committed with changes and have multiple versions. If you don't provide a specific version number, the client defaults to `latest`. For example, you can pull a specific version of `ubuntu` image

```
$ docker pull ubuntu:12.04
```

To get a new Docker image you can either get it from a registry (such as the Docker Hub) or create your own. There are tens of thousands of images available on Docker Hub. You can also search for images directly from the command line using `docker search`.

An important distinction to be aware of when it comes to images is the difference between base and child images.

- **Base images** are images that have no parent image, usually images with an OS like ubuntu, busybox or debian.
- **Child images** are images that build on base images and add additional functionality.

Then there are official and user images, which can be both base and child images.

- **Official images** are images that are officially maintained and supported by the folks at Docker. These are typically one word long. In the list of images above, the `python`, `ubuntu`, `busybox` and `hello-world` images are official images.
- **User images** are images created and shared by users like you and me. They build on base images and add additional functionality. Typically, these are formatted as `user/image-name`.

Our First Image

Now that we have a better understanding of images, it's time to create our own. Our goal in this section will be to create an image that sandboxes a simple Flask application. For the purposes of this workshop, I've already created a fun little Flask app that displays a random cat `.gif` every time it is loaded - because you know, who doesn't like cats? If you haven't already, please go ahead and clone the repository locally like so -

```
$ git clone https://github.com/prakhar1989/docker-curriculum
$ cd docker-curriculum/flask-app
```

> *This should be cloned on the machine where you are running the docker commands and not inside a docker container.*

The next step now is to create an image with this web app. As mentioned above, all user images are based off of a base image. Since our application is written in Python, the base image we're going to use will be Python 3. More specifically, we are going to use the `python:3-onbuild` version of the python image.

What's the `onbuild` version you might ask?

> *These images include multiple ONBUILD triggers, which should be all you need to bootstrap most applications. The build will COPY a `requirements.txt` file, RUN `pip install` on said file, and then copy the current directory into `/usr/src/app`.*

In other words, the `onbuild` version of the image includes helpers that automate the boring parts of getting an app running. Rather than doing these tasks manually (or scripting these tasks), these images do that work for you. We now have all the ingredients to create our own image - a functioning web app and a base image. How are we going to do that? The answer is - using a **Dockerfile**.

Dockerfile

A Dockerfile is a simple text-file that contains a list of commands that the Docker client calls while creating an image. It's a simple way to automate the image creation process. The best part is that the commands you write in a Dockerfile are *almost* identical to their equivalent Linux commands. This means you don't really have to learn new syntax to create your own dockerfiles.

The application directory does contain a Dockerfile but since we're doing this for the first time, we'll create one from scratch. To start, create a new blank file in our favorite text-editor and save it in the **same** folder as the flask app by the name of `Dockerfile`.

We start with specifying our base image. Use the `FROM` keyword to do that -

```
FROM python:3-onbuild
```

The next step usually is to write the commands of copying the files and installing the dependencies. Luckily for us, the `onbuild` version of the image takes care of that. The next thing we need to specify is the port number that needs to be exposed. Since our flask app is running on port `5000`, that's what we'll indicate.

```
EXPOSE 5000
```

The last step is to write the command for running the application, which is simply - `python ./app.py`. We use the CMD command to do that -

```
CMD ["python", "./app.py"]
```

The primary purpose of `CMD` is to tell the container which command it should run when it is started. With that, our `Dockerfile` is now ready. This is how it looks like -

```
# our base image
FROM python:3-onbuild

# specify the port number the container should expose
EXPOSE 5000

# run the application
CMD ["python", "./app.py"]
```

Now that we have our `Dockerfile`, we can build our image. The `docker build` command does the heavy-lifting of creating a Docker image from a `Dockerfile`.

The section below shows you the output of running the same. Before you run the command yourself (don't forget the period), make sure to replace my username with yours. This username should be the same one you created when you registered on Docker hub. If you haven't done that yet, please go ahead and create an account. The `docker build` command is quite simple - it takes an optional tag name with `-t` and a location of the directory containing the `Dockerfile`.

```
$ docker build -t prakhar1989/catnip .
Sending build context to Docker daemon 8.704 kB
Step 1 : FROM python:3-onbuild
# Executing 3 build triggers...
Step 1 : COPY requirements.txt /usr/src/app/
 ---> Using cache
Step 1 : RUN pip install --no-cache-dir -r requirements.txt
 ---> Using cache
Step 1 : COPY . /usr/src/app
 ---> 1d61f639ef9e
Removing intermediate container 4de6ddf5528c
Step 2 : EXPOSE 5000
 ---> Running in 12cfcf6d67ee
 ---> f423c2f179d1
Removing intermediate container 12cfcf6d67ee
Step 3 : CMD python ./app.py
 ---> Running in f01401a5ace9
 ---> 13e87ed1fbc2
```

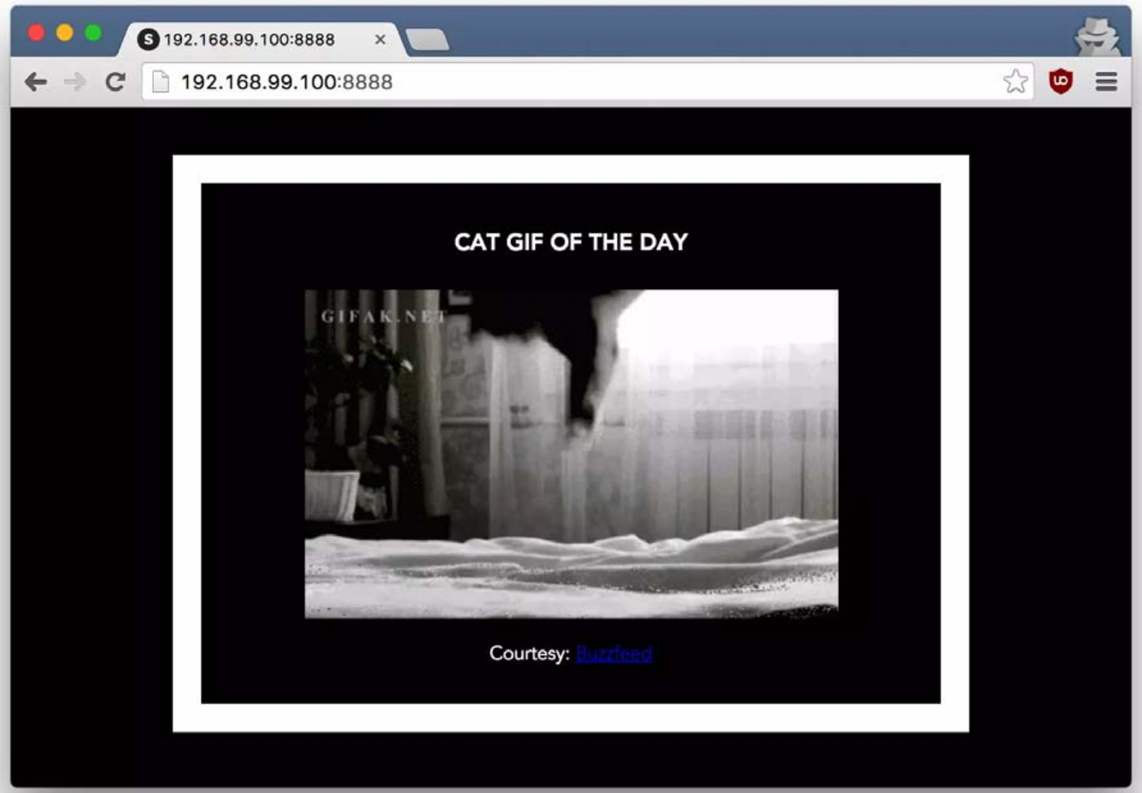
```
Removing intermediate container f01401a5ace9
Successfully built 13e87ed1fbc2
```

If you don't have the `python:3-onbuild` image, the client will first pull the image and then create your image. Hence, your output from running the command will look different from mine. Look carefully and you'll notice that the on-build triggers were executed correctly. If everything went well, your image should be ready! Run `docker images` and see if your image shows.

The last step in this section is to run the image and see if it actually works (replacing my username with yours).

```
$ docker run -p 8888:5000 prakhar1989/catnip
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

The command we just ran used port 5000 for the server inside the container, and exposed this externally on port 8888. Head over to the URL with port 8888, where your app should be live.



Congratulations! You have successfully created your first docker image.

Docker on AWS

What good is an application that can't be shared with friends, right? So in this section we are going to see how we can deploy our awesome application to the cloud so that we can share it with our friends! We're going to use AWS Elastic Beanstalk to get our application up and running in a few clicks. We'll also see how easy it is to make our application scalable and manageable with Beanstalk!

Docker push

The first thing that we need to do before we deploy our app to AWS is to publish our image on a registry which can be accessed by AWS. There are many different Docker registries you can use (you can even host your own). For now, let's use Docker Hub to publish the image. To publish, just type

```
$ docker push prakhar1989/catnip
```

If this is the first time you are pushing an image, the client will ask you to login. Provide the same credentials that you used for logging into Docker Hub.

```
$ docker login
Username: prakhar1989
WARNING: login credentials saved in /Users/prakhar/.docker/config.json
Login Succeeded
```

Remember to replace the name of the image tag above with yours. It is important to have the format of `username/image_name` so that the client knows where to publish.

Once that is done, you can view your image on Docker Hub. For example, here's the web page for my image.

> *Note: One thing that I'd like to clarify before we go ahead is that it is not **imperative** to host your image on a public registry (or any registry) in order to deploy to AWS. In case you're writing code for the next million-dollar unicorn startup you can totally skip this step. The reason why we're pushing our images publicly is that it makes deployment super simple by skipping a few intermediate configuration steps.*

Now that your image is online, anyone who has docker installed can play with your app by typing just a single command.

```
$ docker run -p 8888:5000 prakhar1989/catnip
```

If you've pulled your hair in setting up local dev environments / sharing application configuration in the past, you very well know how awesome this sounds. That's why Docker is so cool!

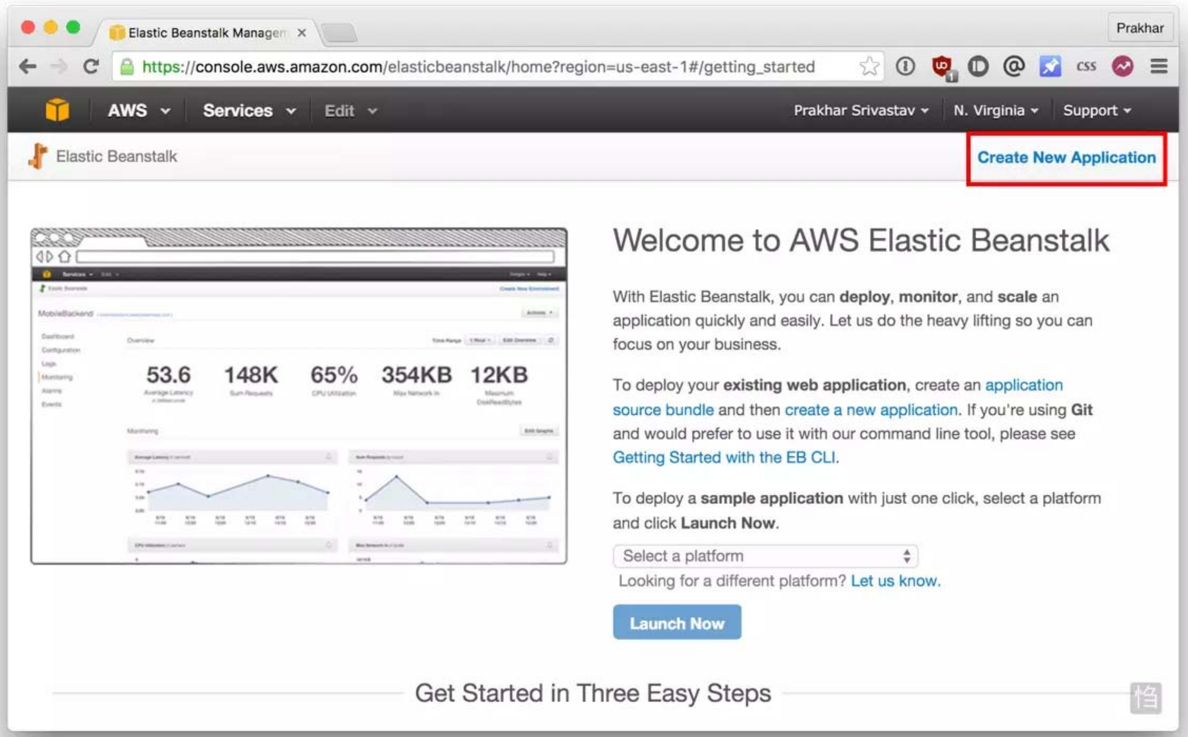
Beanstalk

AWS Elastic Beanstalk (EB) is a PaaS (Platform as a Service) offered by AWS. If you've used Heroku, Google App Engine etc. you'll feel right at home. As a developer, you just tell EB how to run your app and it takes care of the rest - including scaling, monitoring and even updates. In April 2014, EB added support for running single-container Docker deployments which is what we'll use to deploy our app. Although EB has a very intuitive CLI, it does require some setup, and to keep things simple we'll use the web UI to launch our application.

To follow along, you need a functioning AWS account. If you haven't already, please go ahead and do that now - you will need to enter your credit card information. But don't worry, it's free and anything we do in this tutorial will also be free! Let's get started.

Here are the steps:

- Login to your AWS console.
- Click on Elastic Beanstalk. It will be in the compute section on the top left. Alternatively, you can access the Elastic Beanstalk console.


Elastic Beanstalk Managem
Prakhar
https://console.aws.amazon.com/elasticbeanstalk/home?region=us-east-1#/getting_started
AWS
Services
Edit
Prakhar Srivastav
N. Virginia
Support
Elastic Beanstalk
Create New Application
Select a platform
Launch Now
Get Started in Three Easy Steps

Welcome to AWS Elastic Beanstalk

With Elastic Beanstalk, you can **deploy**, **monitor**, and **scale** an application quickly and easily. Let us do the heavy lifting so you can focus on your business.

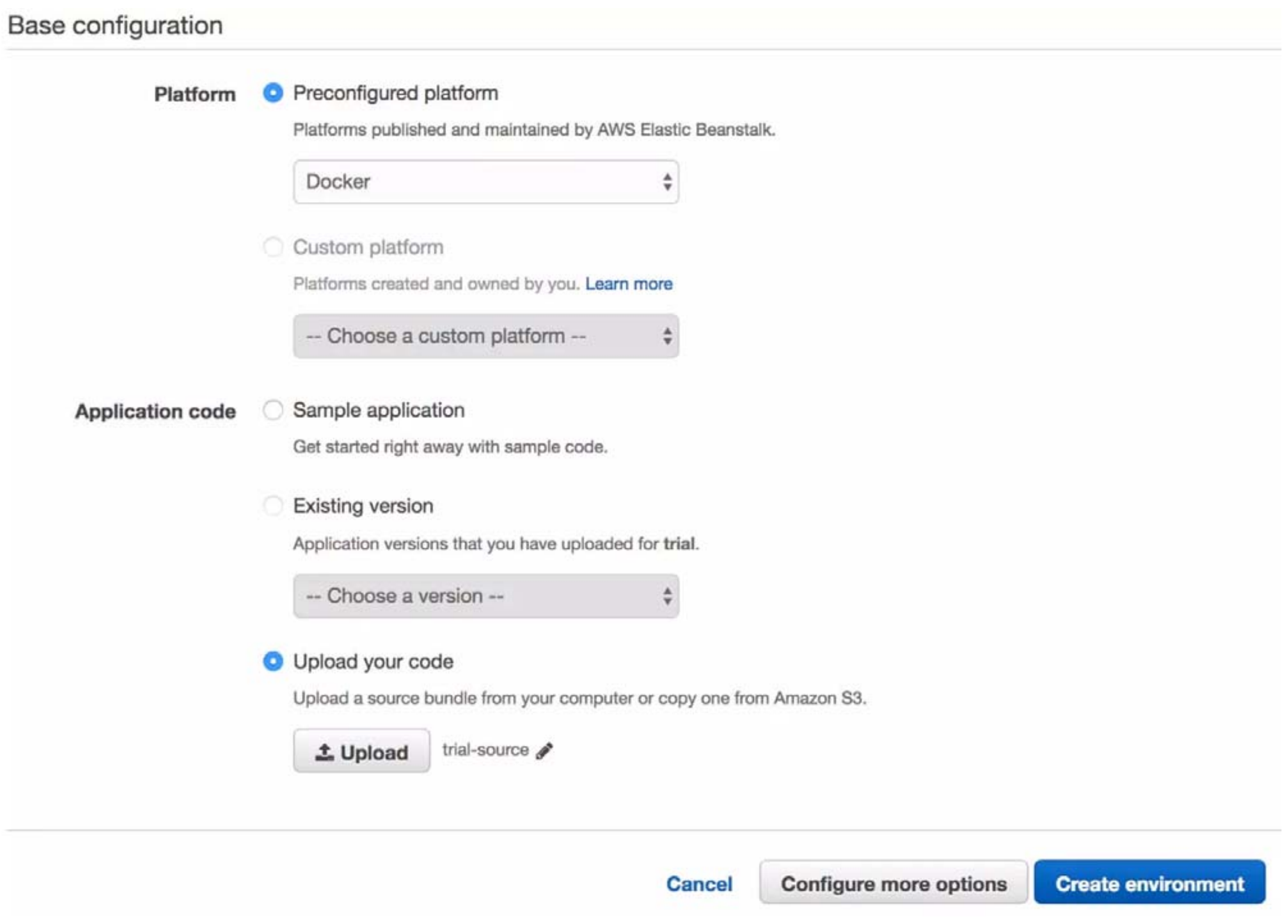
To deploy your **existing web application**, create an application source bundle and then create a new application. If you're using **Git** and would prefer to use it with our command line tool, please see Getting Started with the EB CLI.

To deploy a **sample application** with just one click, select a platform and click **Launch Now**.

Looking for a different platform? Let us know.

- Click on "Create New Application" in the top right
- Give your app a memorable (but unique) name and provide an (optional) description
- In the **New Environment** screen, create a new environment and choose the **Web Server Environment**.

- Fill in the environment information by choosing a domain. This URL is what you'll share with your friends so make sure it's easy to remember.
- Under base configuration section. Choose *Docker* from the *predefined platform*.



Base configuration

Platform: [✓ Preconfigured platform] — Platforms published and maintained by AWS Elastic Beanstalk. — Docker

Custom platform — Platforms created and owned by you. Learn more — -- Choose a custom platform --

Application code: Sample application — Get started right away with sample code.

Existing version — Application versions that you have uploaded for **trial**. — -- Choose a version --

[✓ Upload your code] — Upload a source bundle from your computer or copy one from Amazon S3. — Upload — trial-source

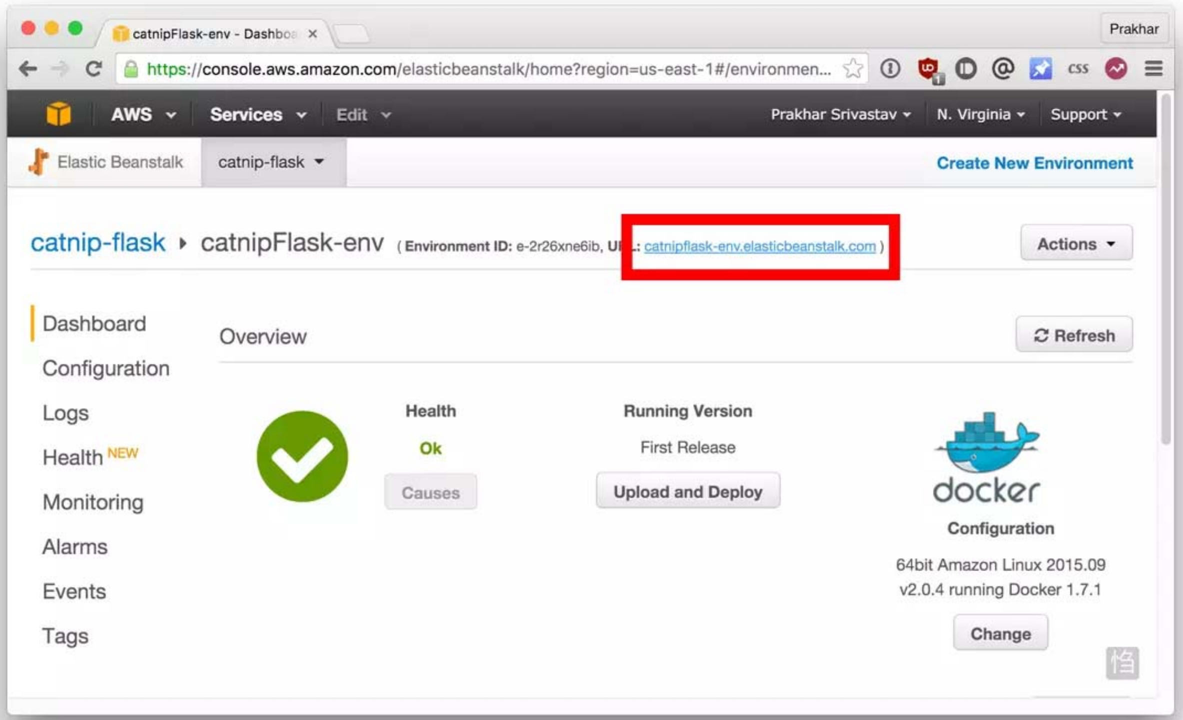
- Now we need upload our application code. But since our application is packaged in a Docker container, we just to tell EB about our contianer. Open the `Dockerrun.aws.json` file located in the `flask-app` folder and edit the `Name` of the image to your image's name. Don't worry, I'll explain the contents of the file shortly. When you are done, click on the radio button for "upload your own" and choose this file.
- The final screen that you see will have a few spinners indicating that your environment is being set up. It typically takes around 5 minutes for the first-time setup.

While we wait, let's quickly see what the `Dockerrun.aws.json` file contains. This file is basically an AWS specific file that tells EB details about our application and docker configuration.

```
{
  "AWSEBDockerrunVersion": "1",
  "Image": {
    "Name": "prakhar1989/catnip",
    "Update": "true"
  },
  "Ports": [
    {
      "ContainerPort": "5000"
      }
    ],
    "Logging": "/var/log/nginx"
  }
```

The file should be pretty self-explanatory, but you can always reference the official documentation for more information. We provide the name of the image that EB should use along with a port that the container should open.

Hopefully by now, our instance should be ready. Head over to the EB page and you should a green tick indicating that your app is alive and kicking.



Go ahead and open the URL in your browser and you should see the application in all its glory. Feel free to email / IM / snapchat this link to your friends and family so that they can enjoy a few cat gifs, too.

Congratulations! You have deployed your first Docker application! That might seem like a lot of steps, but with the command-line tool for EB you can almost mimic the functionality of Heroku in a few keystrokes! Hopefully you agree that Docker takes away a lot of the pains of building and deploying applications in the cloud. I would encourage you to read the AWS documentation on single-container Docker environments to get an idea of what features exist.

In the next (and final) part of the tutorial, we'll up the ante a bit and deploy an application that mimics the real-world more closely; an app with a persistent back-end storage tier. Let's get straight to it!

☰

MULTI-CONTAINER ENVIRONMENTS

In the last section, we saw how easy and fun it is to run applications with Docker. We started with a simple static website and then tried a Flask app. Both of which we could run locally and in the cloud with just a few commands. One thing both these apps had in common was that they were running in a **single container**.

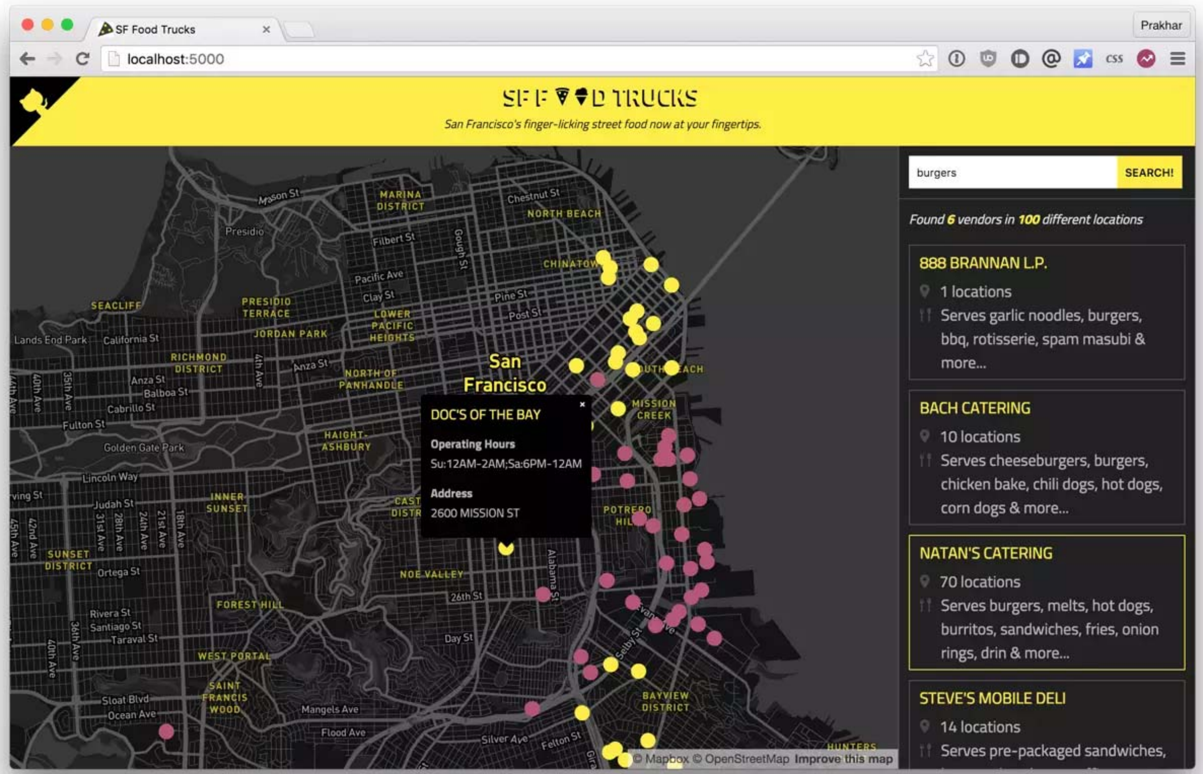
Those of you who have experience running services in production know that usually apps nowadays are not that simple. There's almost always a database (or any other kind of persistent storage) involved. Systems such as Redis and Memcached have become *de riguer* of most web application architectures. Hence, in this section we are going to spend some time learning how to Dockerize applications which rely on different services to run.

In particular, we are going to see how we can run and manage **multi-container** docker environments. Why multi-container you might ask? Well, one of the key points of Docker is the way it provides isolation. The idea of bundling a process with its dependencies in a sandbox (called containers) is what makes this so powerful.

Just like it's a good strategy to decouple your application tiers, it is wise to keep containers for each of the **services** separate. Each tier is likely to have different resource needs and those needs might grow at different rates. By separating the tiers into different containers, we can compose each tier using the most appropriate instance type based on different resource needs. This also plays in very well with the whole microservices movement which is one of the main reasons why Docker (or any other container technology) is at the forefront of modern microservices architectures.

SF Food Trucks

The app that we're going to Dockerize is called SF Food Trucks. My goal in building this app was to have something that is useful (in that it resembles a real-world application), relies on at least one service, but is not too complex for the purpose of this tutorial. This is what I came up with.


SF Food Trucks
localhost:5000
Prakhar
SF FOOD TRUCKS
San Francisco's finger-licking street food now at your fingertips.
burgers
SEARCH!
San Francisco
© Mapbox © OpenStreetMap Improve this map

DOC'S OF THE BAY
Operating Hours
Su:12AM-2AM;Sa:6PM-12AM
Address
2600 MISSION ST

Found 6 vendors in 100 different locations

888 BRANNAN L.P.
1 locations
Serves garlic noodles, burgers, bbq, rotisserie, spam masubi & more...

BACH CATERING
10 locations
Serves cheeseburgers, burgers, chicken bake, chili dogs, hot dogs, corn dogs & more...

NATAN'S CATERING
70 locations
Serves burgers, melts, hot dogs, burritos, sandwiches, fries, onion rings, drin & more...

STEVE'S MOBILE DELI
14 locations
Serves pre-packaged sandwiches,

The app's backend is written in Python (Flask) and for search it uses Elasticsearch. Like everything else in this tutorial, the entire source is available on Github. We'll use this as our candidate application for learning out how to build, run and deploy a multi-container environment.

First up, lets clone the repository locally.

```
$ git clone https://github.com/prakhar1989/FoodTrucks
$ cd FoodTrucks
$ tree -L 2
.
├── Dockerfile
├── README.md
├── aws-compose.yml
├── docker-compose.yml
├── flask-app
│   ├── app.py
│   ├── package-lock.json
│   ├── package.json
│   ├── requirements.txt
│   ├── static
│   ├── templates
│   └── webpack.config.js
├── setup-aws-ecs.sh
├── setup-docker.sh
├── shot.png
└── utils
    ├── generate_geojson.py
    └── trucks.geojson
```

The `flask-app` folder contains the Python application, while the `utils` folder has some utilities to load the data into Elasticsearch. The directory also contains some YAML files and a Dockerfile, all of which we'll see in greater detail as we progress through this tutorial. If you are curious, feel free to take a look at the files.

Now that you're excited (hopefully), let's think of how we can Dockerize the app. We can see that the application consists of a Flask backend server and an Elasticsearch service. A natural way to split this app would be to have two containers - one running the Flask process and another running the Elasticsearch (ES) process. That way if our app becomes popular, we can scale it by adding more containers depending on where the bottleneck lies.

Great, so we need two containers. That shouldn't be hard right? We've already built our own Flask container in the previous section. And for Elasticsearch, let's see if we can find something on the hub.

```
$ docker search elasticsearch
NAME                              DESCRIPTION                                     STARS
elasticsearch                     Elasticsearch is a powerful open source se...   697
itzg/elasticsearch                Provides an easily configurable Elasticsea...   17
tutum/elasticsearch               Elasticsearch image - listens in port 9200.     15
barnybug/elasticsearch            Latest Elasticsearch 1.7.2 and previous re...   15
digitalwonderland/elasticsearch   Latest Elasticsearch with Marvel & Kibana       12
monsantoco/elasticsearch          ElasticSearch Docker image                      9
```

Quite unsurprisingly, there exists an officially supported image for Elasticsearch. To get ES running, we can simply use `docker run` and have a single-node ES container running locally within no time.

> *Note: Elastic, the company behind Elasticsearch, maintains its own registry for Elastic products. It's recommended to use the images from that registry if you plan to use Elasticsearch.*

Let's first pull the image

```
$ docker pull docker.elastic.co/elasticsearch/elasticsearch:6.3.2
```

and then run it in development mode by specifying ports and setting an environment variable that configures Elasticsearch cluster to run as a single-node.

```
$ docker run -d --name es -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" docker.e
277451c15ec183dd939e80298ea4bcf55050328a39b04124b387d668e3ed3943
```

As seen above, we use `--name es` to give our container a name which makes it easy to use in subsequent commands. Once the container is started, we can see the logs by running `docker container logs` with the container name (or ID) to inspect the logs. You should logs similar to below if Elasticsearch started successfully.

> *Note: Elasticsearch takes a few seconds to start so you might need to wait before you see `initialized` in the logs.*

```
$ docker container ls
CONTAINER ID        IMAGE                                                  COMMAND
277451c15ec1        docker.elastic.co/elasticsearch/elasticsearch:6.3.2    "/usr/local/bin/dc

$ docker container logs es
[2018-07-29T05:49:09,304][INFO ][o.e.n.Node               ] [] initializing ...
[2018-07-29T05:49:09,385][INFO ][o.e.e.NodeEnvironment    ] [L1VMyzt] using [1] data paths,
[2018-07-29T05:49:09,385][INFO ][o.e.e.NodeEnvironment    ] [L1VMyzt] heap size [990.7mb], c
[2018-07-29T05:49:11,979][INFO ][o.e.p.PluginsService     ] [L1VMyzt] loaded module [x-pack-
[2018-07-29T05:49:11,980][INFO ][o.e.p.PluginsService     ] [L1VMyzt] loaded module [x-pack-
[2018-07-29T05:49:11,980][INFO ][o.e.p.PluginsService     ] [L1VMyzt] loaded module [x-pack-
[2018-07-29T05:49:11,980][INFO ][o.e.p.PluginsService     ] [L1VMyzt] loaded module [x-pack-
[2018-07-29T05:49:11,981][INFO ][o.e.p.PluginsService     ] [L1VMyzt] loaded plugin [ingest-
[2018-07-29T05:49:11,981][INFO ][o.e.p.PluginsService     ] [L1VMyzt] loaded plugin [ingest-
[2018-07-29T05:49:17,659][INFO ][o.e.d.DiscoveryModule    ] [L1VMyzt] using discovery type [
[2018-07-29T05:49:18,962][INFO ][o.e.n.Node               ] [L1VMyzt] initialized
[2018-07-29T05:49:18,963][INFO ][o.e.n.Node               ] [L1VMyzt] starting ...
[2018-07-29T05:49:19,218][INFO ][o.e.t.TransportService   ] [L1VMyzt] publish_address {172.1
[2018-07-29T05:49:19,302][INFO ][o.e.x.s.t.n.SecurityNetty4HttpServerTransport] [L1VMyzt] pu
[2018-07-29T05:49:19,303][INFO ][o.e.n.Node               ] [L1VMyzt] started
[2018-07-29T05:49:19,439][WARN ][o.e.x.s.a.s.m.NativeRoleMappingStore] [L1VMyzt] Failed to c
[2018-07-29T05:49:19,542][INFO ][o.e.g.GatewayService     ] [L1VMyzt] recovered [0] indices
```

Now, lets try to see if can send a request to the Elasticsearch container. We use the `9200` port to send a `cURL` request to the container.

```
$ curl 0.0.0.0:9200
{
  "name" : "ijJDAOm",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "a_nSV3XmTCqpzYYzb-LhNw",
  "version" : {
    "number" : "6.3.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "053779d",
    "build_date" : "2018-07-20T05:20:23.451332Z",
    "build_snapshot" : false,
    "lucene_version" : "7.3.1",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

Sweet! It's looking good! While we are at it, let's get our Flask container running too. But before we get to that, we need a `Dockerfile`. In the last section, we used `python:3-`

`onbuild` image as our base image. This time, however, apart from installing Python dependencies via `pip`, we want our application to also generate our minified Javascript file for production. For this, we'll require Nodejs. Since we need a custom build step, we'll start from the `ubuntu` base image to build our `Dockerfile` from scratch.

> *Note: if you find that an existing image doesn't cater to your needs, feel free to start from another base image and tweak it yourself. For most of the images on Docker Hub, you should be able to find the corresponding `Dockerfile` on Github. Reading through existing Dockerfiles is one of the best ways to learn how to roll your own.*

Our Dockerfile for the flask app looks like below -

```
# start from base
FROM ubuntu:14.04
MAINTAINER Prakhar Srivastav <prakhar@prakhar.me>

# install system-wide deps for python and node
RUN apt-get -yqq update
RUN apt-get -yqq install python-pip python-dev curl gnupg
RUN curl -sL https://deb.nodesource.com/setup_8.x | bash
RUN apt-get install -yq nodejs

# copy our application code
ADD flask-app /opt/flask-app
WORKDIR /opt/flask-app

# fetch app specific deps
RUN npm install
RUN npm run build
RUN pip install -r requirements.txt

# expose port
EXPOSE 5000

# start app
CMD [ "python", "./app.py" ]
```

Quite a few new things here so let's quickly go over this file. We start off with the Ubuntu LTS base image and use the package manager `apt-get` to install the dependencies namely - Python and Node. The `yqq` flag is used to suppress output and assumes "Yes" to all prompts.

We then use the `ADD` command to copy our application into a new volume in the container - `/opt/flask-app`. This is where our code will reside. We also set this as our working directory, so that the following commands will be run in the context of this location. Now that our system-wide dependencies are installed, we get around to install app-specific ones. First off we tackle Node by installing the packages from npm and running the build command as defined in our `package.json` file. We finish the file off by installing the Python packages, exposing the port and defining the `CMD` to run as we did in the last section.

Finally, we can go ahead, build the image and run the container (replace `prakhar1989` with your username below).

```
$ git clone https://github.com/prakhar1989/FoodTrucks && cd FoodTrucks
$ docker build -t prakhar1989/foodtrucks-web .
```

In the first run, this will take some time as the Docker client will download the ubuntu image, run all the commands and prepare your image. Re-running `docker build` after any subsequent changes you make to the application code will almost be instantaneous. Now let's try running our app.

```
$ docker run -P --rm prakhar1989/foodtrucks-web
Unable to connect to ES. Retying in 5 secs...
Unable to connect to ES. Retying in 5 secs...
Unable to connect to ES. Retying in 5 secs...
Out of retries. Bailing out...
```

Oops! Our flask app was unable to run since it was unable to connect to Elasticsearch. How do we tell one container about the other container and get them to talk to each other? The answer lies in the next section.

Docker Network

Before we talk about the features Docker provides especially to deal with such scenarios, let's see if we can figure out a way to get around the problem. Hopefully this should give you an appreciation for the specific feature that we are going to study.

Okay, so let's run `docker container ls` (which is same as `docker ps`) and see what we have.

```
$ docker container ls
CONTAINER ID        IMAGE                                                 COMMAND
277451c15ec1        docker.elastic.co/elasticsearch/elasticsearch:6.3.2   "/usr/local/bin/dc
```

So we have one ES container running on `0.0.0.0:9200` port which we can directly access. If we can tell our Flask app to connect to this URL, it should be able to connect and talk to ES, right? Let's dig into our Python code and see how the connection details are defined.

```
es = Elasticsearch(host='es')
```

To make this work, we need to tell the Flask container that the ES container is running on `0.0.0.0` host (the port by default is `9200`) and that should make it work, right?

Unfortunately that is not correct since the IP `0.0.0.0` is the IP to access ES container from the **host machine** i.e. from my Mac. Another container will not be able to access this on the same IP address. Okay if not that IP, then which IP address should the ES container be accessible by? I'm glad you asked this question.

Now is a good time to start our exploration of networking in Docker. When docker is installed, it creates three networks automatically.

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c2c695315b3a        bridge              bridge              local
a875bec5d6fd        host                host                local
ead0e804a67b        none                null                local
```

The **bridge** network is the network in which containers are run by default. So that means that when I ran the ES container, it was running in this bridge network. To validate this, let's inspect the network

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "c2c695315b3aaf8fc30530bb3c6b8f6692cedd5cc7579663f0550dfdd21c9a26",
        "Created": "2018-07-28T20:32:39.405687265Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "277451c15ec183dd939e80298ea4bcf55050328a39b04124b387d668e3ed3943": {
                "Name": "es",
                "EndpointID": "5c417a2fc6b13d8ec97b76bbd54aaf3ee2d48f328c3f7279ee335174fbb4
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

You can see that our container `277451c15ec1` is listed under the `Containers` section in the output. What we also see is the IP address this container has been allotted - `172.17.0.2`. Is this the IP address that we're looking for? Let's find out by running our flask container and trying to access this IP.

```
$ docker run -it --rm prakhar1989/foodtrucks-web bash
root@35180ccc206a:/opt/flask-app# curl 172.17.0.2:9200
{
  "name" : "Jane Foster",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.1.1",
    "build_hash" : "40e2c53a6b6c2972b3d13846e450e66f4375bd71",
    "build_timestamp" : "2015-12-15T13:05:55Z",
    "build_snapshot" : false,
    "lucene_version" : "5.3.1"
  },
  "tagline" : "You Know, for Search"
}
root@35180ccc206a:/opt/flask-app# exit
```

This should be fairly straightforward to you by now. We start the container in the interactive mode with the `bash` process. The `--rm` is a convenient flag for running one off commands since the container gets cleaned up when it's work is done. We try a `curl` but we need to install it first. Once we do that, we see that we can indeed talk to ES on `172.17.0.2:9200`. Awesome!

Although we have figured out a way to make the containers talk to each other, there are still two problems with this approach -

1. How do we tell the Flask container that `es` hostname stands for `172.17.0.2` or some other IP since the IP can change?
2. Since the *bridge* network is shared by every container by default, this method is **not secure**. How do we isolate our network?

The good news that Docker has a great answer to our questions. It allows us to define our own networks while keeping them isolated using the `docker network` command.

Let's first go ahead and create our own network.

```
$ docker network create foodtrucks-net
0815b2a3bb7a6608e850d05553cc0bda98187c4528d94621438f31d97a6fea3c
```

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c2c695315b3a        bridge              bridge              local
0815b2a3bb7a        foodtrucks-net      bridge              local
a875bec5d6fd        host                host                local
ead0e804a67b        none                null                local
```

The `network create` command creates a new *bridge* network, which is what we need at the moment. In terms of Docker, a bridge network uses a software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network. The Docker bridge driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other. There are other kinds of networks that you can create, and you are encouraged to read about them in the official docs.

Now that we have a network, we can launch our containers inside this network using the `--net` flag. Let's do that - but first, we will stop and delete our ES container that is running in the bridge (default) network.

```
$ docker container stop es
es

$ docker rm es
es

$ docker run -d --name es --net foodtrucks-net -p 9200:9200 -p 9300:9300 -e "discovery.type=
13d6415f73c8d88bddb1f236f584b63dbaf2c3051f09863a3f1ba219edba3673

$ docker network inspect foodtrucks-net
[
    {
        "Name": "foodtrucks-net",
        "Id": "0815b2a3bb7a6608e850d05553cc0bda98187c4528d94621438f31d97a6fea3c",
        "Created": "2018-07-30T00:01:29.1500984Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
                "13d6415f73c8d88bddb1f236f584b63dbaf2c3051f09863a3f1ba219edba3673": {
                    "Name": "es",
                    "EndpointID": "29ba2d33f9713e57eb6b38db41d656e4ee2c53e4a2f7cf636bdca0ec59cd3
                    "MacAddress": "02:42:ac:12:00:02",
                    "IPv4Address": "172.18.0.2/16",
                    "IPv6Address": ""
                }
            },
            "Options": {},
            "Labels": {}
        }
]
```

As you can see, our `es` container is now running inside the `foodtrucks-net` bridge network. Now let's inspect what happens when we launch in our `foodtrucks-net` network.

```
$ docker run -it --rm --net foodtrucks-net prakhar1989/foodtrucks-web bash
root@9d2722cf282c:/opt/flask-app# curl es:9200
{
  "name" : "wWALl9M",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "BA36XuOiRPaghPNBLBHleQ",
  "version" : {
    "number" : "6.3.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "053779d",
    "build_date" : "2018-07-20T05:20:23.451332Z",
    "build_snapshot" : false,
    "lucene_version" : "7.3.1",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
root@53af252b771a:/opt/flask-app# ls
app.py  node_modules  package.json  requirements.txt  static  templates  webpack.config.js
root@53af252b771a:/opt/flask-app# python app.py
Index not found...
Loading data in elasticsearch ...
Total trucks loaded:  733
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
root@53af252b771a:/opt/flask-app# exit
```

Wohoo! That works! On user-defined networks like foodtrucks-net, containers can not only communicate by IP address, but can also resolve a container name to an IP address. This capability is called *automatic service discovery*. Great! Let's launch our Flask container for real now -

```
$ docker run -d --net foodtrucks-net -p 5000:5000 --name foodtrucks-web prakhar1989/foodtruc
852fc74de2954bb72471b858dce64d764181dca0cf7693fed201d76da33df794

$ docker container ls
CONTAINER ID        IMAGE                                                     COMMAND
852fc74de295        prakhar1989/foodtrucks-web                                "python ./app.py"
```

```
13d6415f73c8        docker.elastic.co/elasticsearch/elasticsearch:6.3.2   "/usr/local/bin/dc

$ curl -I 0.0.0.0:5000
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 3697
Server: Werkzeug/0.11.2 Python/2.7.6
Date: Sun, 10 Jan 2016 23:58:53 GMT
```

Head over to http://0.0.0.0:5000 and see your glorious app live! Although that might have seemed like a lot of work, we actually just typed 4 commands to go from zero to running. I've collated the commands in a bash script.

```
#!/bin/bash

# build the flask container
docker build -t prakhar1989/foodtrucks-web .

# create the network
docker network create foodtrucks-net

# start the ES container
docker run -d --name es --net foodtrucks-net -p 9200:9200 -p 9300:9300 -e "discovery.type=si

# start the flask app container
docker run -d --net foodtrucks-net -p 5000:5000 --name foodtrucks-web prakhar1989/foodtrucks
```

Now imagine you are distributing your app to a friend, or running on a server that has docker installed. You can get a whole app running with just one command!

```
$ git clone https://github.com/prakhar1989/FoodTrucks
$ cd FoodTrucks
$ ./setup-docker.sh
```

And that's it! If you ask me, I find this to be an extremely awesome, and a powerful way of sharing and running your applications!

Docker Compose

Till now we've spent all our time exploring the Docker client. In the Docker ecosystem, however, there are a bunch of other open-source tools which play very nicely with Docker. A few of them are -

1. Docker Machine - Create Docker hosts on your computer, on cloud providers, and inside your own data center
2. Docker Compose - A tool for defining and running multi-container Docker applications.
3. Docker Swarm - A native clustering solution for Docker

☰

4. Kubernetes - Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.

In this section, we are going to look at one of these tools, Docker Compose, and see how it can make dealing with multi-container apps easier.

The background story of Docker Compose is quite interesting. Roughly two years ago, a company called OrchardUp launched a tool called Fig. The idea behind Fig was to make isolated development environments work with Docker. The project was very well received on Hacker News - I oddly remember reading about it but didn't quite get the hang of it.

The first comment on the forum actually does a good job of explaining what Fig is all about.

> *So really at this point, that's what Docker is about: running processes. Now Docker offers a quite rich API to run the processes: shared volumes (directories) between containers (i.e. running images), forward port from the host to the container, display logs, and so on. But that's it: Docker as of now, remains at the process level.*

> *While it provides options to orchestrate multiple containers to create a single "app", it doesn't address the managemement of such group of containers as a single entity. And that's where tools such as Fig come in: talking about a group of containers as a single entity. Think "run an app" (i.e. "run an orchestrated cluster of containers") instead of "run a container".*

It turns out that a lot of people using docker agree with this sentiment. Slowly and steadily as Fig became popular, Docker Inc. took notice, acquired the company and re-branded Fig as Docker Compose.

So what is *Compose* used for? Compose is a tool that is used for defining and running multi-container Docker apps in an easy way. It provides a configuration file called `docker-compose.yml` that can be used to bring up an application and the suite of services it depends on with just one command. Compose works in all environments: production, staging, development, testing, as well as CI workflows, although Compose is ideal for development and testing environments.

Let's see if we can create a `docker-compose.yml` file for our SF-Foodtrucks app and evaluate whether Docker Compose lives up to its promise.

The first step, however, is to install Docker Compose. If you're running Windows or Mac, Docker Compose is already installed as it comes in the Docker Toolbox. Linux users can easily get their hands on Docker Compose by following the instructions on the docs. Since Compose is written in Python, you can also simply do `pip install docker-compose`. Test your installation with -

```
$ docker-compose --version
docker-compose version 1.21.2, build a133471
```

Now that we have it installed, we can jump on the next step i.e. the Docker Compose file `docker-compose.yml`. The syntax for YAML is quite simple and the repo already contains the docker-compose file that we'll be using.

```
version: "3"
services:
  es:
    image: docker.elastic.co/elasticsearch/elasticsearch:6.3.2
    container_name: es
    environment:
      - discovery.type=single-node
    ports:
      - 9200:9200
    volumes:
      - esdata1:/usr/share/elasticsearch/data
  web:
    image: prakhar1989/foodtrucks-web
    command: python app.py
    depends_on:
      - es
    ports:
      - 5000:5000
    volumes:
      - ./flask-app:/opt/flask-app
volumes:
    esdata1:
      driver: local
```

Let me breakdown what the file above means. At the parent level, we define the names of our services - `es` and `web`. For each service, that Docker needs to run, we can add additional parameters out of which `image` is required. For `es`, we just refer to the `elasticsearch` image available on Elastic registry. For our Flask app, we refer to the image that we built at the beginning of this section.

Via other parameters such as `command` and `ports` we provide more information about the container. The `volumes` parameter specifies a mount point in our `web` container where the code will reside. This is purely optional and is useful if you need access to logs etc. We'll later see how this can be useful during development. Refer to the online reference to learn more about the parameters this file supports. We also add volumes for `es` container so that the data we load persists between restarts. We also specify `depends_on`, which tells docker to start the `es` container before `web`. You can read more about it on docker compose docs.

> *Note: You must be inside the directory with the `docker-compose.yml` file in order to execute most Compose commands.*

Great! Now the file is ready, let's see `docker-compose` in action. But before we start, we need to make sure the ports are free. So if you have the Flask and ES containers running, lets turn them off.

```
$ docker stop $(docker ps -q)
39a2f5df14ef
2a1b77e066e6
```

Now we can run `docker-compose`. Navigate to the food trucks directory and run `docker-compose up`.

```
$ docker-compose up
Creating network "foodtrucks_default" with the default driver
Creating foodtrucks_es_1
Creating foodtrucks_web_1
Attaching to foodtrucks_es_1, foodtrucks_web_1
es_1  | [2016-01-11 03:43:50,300][INFO ][node                     ] [Comet] version[2.1.1],
es_1  | [2016-01-11 03:43:50,307][INFO ][node                     ] [Comet] initializing ...
es_1  | [2016-01-11 03:43:50,366][INFO ][plugins                  ] [Comet] loaded [], sites
es_1  | [2016-01-11 03:43:50,421][INFO ][env                      ] [Comet] using [1] data p
es_1  | [2016-01-11 03:43:52,626][INFO ][node                     ] [Comet] initialized
es_1  | [2016-01-11 03:43:52,632][INFO ][node                     ] [Comet] starting ...
es_1  | [2016-01-11 03:43:52,703][WARN ][common.network           ] [Comet] publish address:
es_1  | [2016-01-11 03:43:52,704][INFO ][transport                ] [Comet] publish_address
es_1  | [2016-01-11 03:43:52,721][INFO ][discovery                ] [Comet] elasticsearch/cE
es_1  | [2016-01-11 03:43:55,785][INFO ][cluster.service          ] [Comet] new_master {Come
es_1  | [2016-01-11 03:43:55,818][WARN ][common.network           ] [Comet] publish address:
es_1  | [2016-01-11 03:43:55,819][INFO ][http                     ] [Comet] publish_address
es_1  | [2016-01-11 03:43:55,819][INFO ][node                     ] [Comet] started
es_1  | [2016-01-11 03:43:55,826][INFO ][gateway                  ] [Comet] recovered [0] in
es_1  | [2016-01-11 03:44:01,825][INFO ][cluster.metadata         ] [Comet] [sfdata] creatin
es_1  | [2016-01-11 03:44:02,373][INFO ][cluster.metadata         ] [Comet] [sfdata] update_
es_1  | [2016-01-11 03:44:02,510][INFO ][cluster.metadata         ] [Comet] [sfdata] update_
es_1  | [2016-01-11 03:44:02,593][INFO ][cluster.metadata         ] [Comet] [sfdata] update_
es_1  | [2016-01-11 03:44:02,708][INFO ][cluster.metadata         ] [Comet] [sfdata] update_
es_1  | [2016-01-11 03:44:03,047][INFO ][cluster.metadata         ] [Comet] [sfdata] update_
web_1 |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Head over to the IP to see your app live. That was amazing wasn't it? Just few lines of configuration and we have two Docker containers running successfully in unison. Let's stop the services and re-run in detached mode.

```
web_1 |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
Killing foodtrucks_web_1 ... done
Killing foodtrucks_es_1 ... done

$ docker-compose up -d
Creating es               ... done
Creating foodtrucks_web_1 ... done
```

```
$ docker-compose ps
      Name                        Command                  State                  Ports
----------------------------------------------------------------------------------------------
es                 /usr/local/bin/docker-entr ...   Up      0.0.0.0:9200->9200/tcp, 9300/tcp
foodtrucks_web_1   python app.py                    Up      0.0.0.0:5000->5000/tcp
```

Unsurprisingly, we can see both the containers running successfully. Where do the names come from? Those were created automatically by Compose. But does *Compose* also create the network automatically? Good question! Let's find out.

First off, let us stop the services from running. We can always bring them back up in just one command. Data volumes will persist, so it's possible to start the cluster again with the same data using docker-compose up. To destroy the cluster and the data volumes, just type `docker-compose down -v`.

```
$ docker-compose down -v
Stopping foodtrucks_web_1 ... done
Stopping es               ... done
Removing foodtrucks_web_1 ... done
Removing es               ... done
Removing network foodtrucks_default
Removing volume foodtrucks_esdata1
```

While we're are at it, we'll also remove the `foodtrucks` network that we created last time.

```
$ docker network rm foodtrucks-net
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c2c695315b3a        bridge              bridge              local
a875bec5d6fd        host                host                local
ead0e804a67b        none                null                local
```

Great! Now that we have a clean slate, let's re-run our services and see if *Compose* does it's magic.

```
$ docker-compose up -d
Recreating foodtrucks_es_1
Recreating foodtrucks_web_1

$ docker container ls
CONTAINER ID        IMAGE                        COMMAND                  CREATED
f50bb33a3242        prakhar1989/foodtrucks-web   "python app.py"          14 seconds ago
e299ceeb4caa        elasticsearch                "/docker-entrypoint.s"   14 seconds ago
```

So far, so good. Time to see if any networks were created.

```
$ docker network ls
NETWORK ID          NAME                 DRIVER
c2c695315b3a        bridge               bridge              local
f3b80f381ed3        foodtrucks_default   bridge              local
```

```
a875bec5d6fd        host                host                local
ead0e804a67b        none                null                local
```

You can see that compose went ahead and created a new network called `foodtrucks_default` and attached both the new services in that network so that each of these are discoverable to the other. Each container for a service joins the default network and is both reachable by other containers on that network, and discoverable by them at a hostname identical to the container name.

```
$ docker ps
CONTAINER ID        IMAGE                                                     COMMAND
8c6bb7e818ec        docker.elastic.co/elasticsearch/elasticsearch:6.3.2      "/usr/local/bin/dc
7640cec7feb7        prakhar1989/foodtrucks-web                                "python app.py"

$ docker network inspect foodtrucks_default
[
    {
        "Name": "foodtrucks_default",
        "Id": "f3b80f381ed3e03b3d5e605e42c4a576e32d38ba24399e963d7dad848b3b4fe7",
        "Created": "2018-07-30T03:36:06.0384826Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": true,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "7640cec7feb7f5615eaac376271a93fb8bab2ce54c7257256bf16716e05c65a5": {
                "Name": "foodtrucks_web_1",
                "EndpointID": "b1aa3e735402abafea3edfbba605eb4617f81d94f1b5f8fcc566a874660a0
                "MacAddress": "02:42:ac:13:00:02",
                "IPv4Address": "172.19.0.2/16",
                "IPv6Address": ""
            },
            "8c6bb7e818ec1f88c37f375c18f00beb030b31f4b10aee5a0952aad753314b57": {
                "Name": "es",
                "EndpointID": "649b3567d38e5e6f03fa6c004a4302508c14a5f2ac086ee6dcf13ddef936c
                "MacAddress": "02:42:ac:13:00:03",
                "IPv4Address": "172.19.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {
                "com.docker.compose.network": "default",
                "com.docker.compose.project": "foodtrucks",
                "com.docker.compose.version": "1.21.2"
            }
        }
]
```

Development Workflow

Before we jump to the next section, there's one last thing I wanted to cover about docker-compose. As stated earlier, docker-compose is really great for development and testing. So let's see how we can configure compose to make our lives easier during development.

Throughout this tutorial, we've worked with readymade docker images. While we've built images from scratch, we haven't touched any application code yet and mostly restricted ourselves to editing Dockerfiles and YAML configurations. One thing that you must be wondering is how does the workflow look during development? Is one supposed to keep creating Docker images for every change, then publish it and then run it to see if the changes works as expected? I'm sure that sounds super tedious. There has to be a better way. In this section, that's what we're going to explore.

Let's see how we can make a change in the Foodtrucks app we just ran. Make sure you have the app running,

```
$ docker container ls
CONTAINER ID        IMAGE                                                    COMMAND
5450ebedd03c        prakhar1989/foodtrucks-web                               "python app.py"
05d408b25dfe        docker.elastic.co/elasticsearch/elasticsearch:6.3.2     "/usr/local/bin/dc
```

Now let's see if we can change this app to display a `Hello world!` message when a request is made to `/hello` route. Currently, the app responds with a 404.

```
$ curl -I 0.0.0.0:5000/hello
HTTP/1.0 404 NOT FOUND
Content-Type: text/html
Content-Length: 233
Server: Werkzeug/0.11.2 Python/2.7.15rc1
Date: Mon, 30 Jul 2018 15:34:38 GMT
```

Why does this happen? Since ours is a Flask app, we can see `app.py` for answers. In Flask, routes are defined with @app.route syntax. In the file, you'll see that we only have three routes defined - `/`, `/debug` and `/search`. The `/` route renders the main app, the `debug` route is used to return some debug information and finally `search` is used by the app to query elasticsearch.

```
$ curl 0.0.0.0:5000/debug
{
  "msg": "yellow open sfdata Ibkx7WYjSt-g8NZXOEtTMg 5 1 618 0 1.3mb 1.3mb\n",
  "status": "success"
}
```

Given that context, how would we add a new route for `hello`? You guessed it! Let's open `flask-app/app.py` in our favorite editor and make the following change

```
@app.route('/')
def index():
  return render_template("index.html")

# add a new hello route
@app.route('/hello')
def hello():
  return "hello world!"
```

Now let's try making a request again

```
$ curl -I 0.0.0.0:5000/hello
HTTP/1.0 404 NOT FOUND
Content-Type: text/html
Content-Length: 233
Server: Werkzeug/0.11.2 Python/2.7.15rc1
Date: Mon, 30 Jul 2018 15:34:38 GMT
```

Oh no! That didn't work! What did we do wrong? While we did make the change in `app.py`, the file resides in our machine (or the host machine), but since Docker is running our containers based off the `prakhar1989/foodtrucks-web` image, it doesn't know about this change. To validate this, lets try the following -

```
$ docker-compose run web bash
Starting es ... done
root@581e351c82b0:/opt/flask-app# ls
app.py        package-lock.json  requirements.txt  templates
node_modules  package.json       static            webpack.config.js
root@581e351c82b0:/opt/flask-app# grep hello app.py
root@581e351c82b0:/opt/flask-app# exit
```

What we're trying to do here is to validate that our changes are not in the `app.py` that's running in the container. We do this by running the command `docker compose run`, which is similar to its cousin `docker run` but takes additional arguments for the service (which is `web` in our case). As soon as we run `bash`, the shell opens in `/opt/flask-app` as specified in our Dockerfile. From the grep command we can see that our changes are not in the file.

Lets see how we can fix it. First off, we need to tell docker compose to not use the image and instead use the files locally. We'll also set debug mode to `true` so that Flask knows to reload the server when `app.py` changes. Replace the `web` portion of the `docker-compose.yml` file like so:

```
version: "3"
services:
  es:
    image: docker.elastic.co/elasticsearch/elasticsearch:6.3.2
    container_name: es
    environment:
      - discovery.type=single-node
    ports:
      - 9200:9200
    volumes:
      - esdata1:/usr/share/elasticsearch/data
  web:
    build: . # replaced image with build
    command: python app.py
    environment:
      - DEBUG=True  # set an env var for flask
    depends_on:
      - es
    ports:
      - "5000:5000"
    volumes:
      - ./flask-app:/opt/flask-app
volumes:
    esdata1:
      driver: local
```

With that change (diff), let's stop and start the containers.

```
$ docker-compose down -v
Stopping foodtrucks_web_1 ... done
Stopping es               ... done
Removing foodtrucks_web_1 ... done
Removing es               ... done
Removing network foodtrucks_default
Removing volume foodtrucks_esdata1

$ docker-compose up -d
Creating network "foodtrucks_default" with the default driver
Creating volume "foodtrucks_esdata1" with local driver
Creating es ... done
Creating foodtrucks_web_1 ... done
```

As a final step, lets make the change in `app.py` by adding a new route. Now we try to curl

```
$ curl 0.0.0.0:5000/hello
hello world
```

Wohoo! We get a valid response! Try playing around by making more changes in the app.

That concludes our tour of Docker Compose. With Docker Compose, you can also pause your services, run a one-off command on a container and even scale the number of containers. I also recommend you checkout a few other use-cases of Docker compose. Hopefully I was able to show you how easy it is to manage multi-container environments with Compose. In the final section, we are going to deploy our app to AWS!

AWS Elastic Container Service

In the last section we used `docker-compose` to run our app locally with a single command: `docker-compose up`. Now that we have a functioning app we want to share this with the world, get some users, make tons of money and buy a big house in Miami. Executing the last three are beyond the scope of tutorial, so we'll spend our time instead on figuring out how we can deploy our multi-container apps on the cloud with AWS.

If you've read this far you are much pretty convinced that Docker is a pretty cool technology. And you are not alone. Seeing the meteoric rise of Docker, almost all Cloud vendors started working on adding support for deploying Docker apps on their platform. As of today, you can deploy containers on Google Cloud Platform, AWS, Azure and many others. We already got a primer on deploying single container apps with Elastic Beanstalk and in this section we are going to look at Elastic Container Service (or ECS) by AWS.

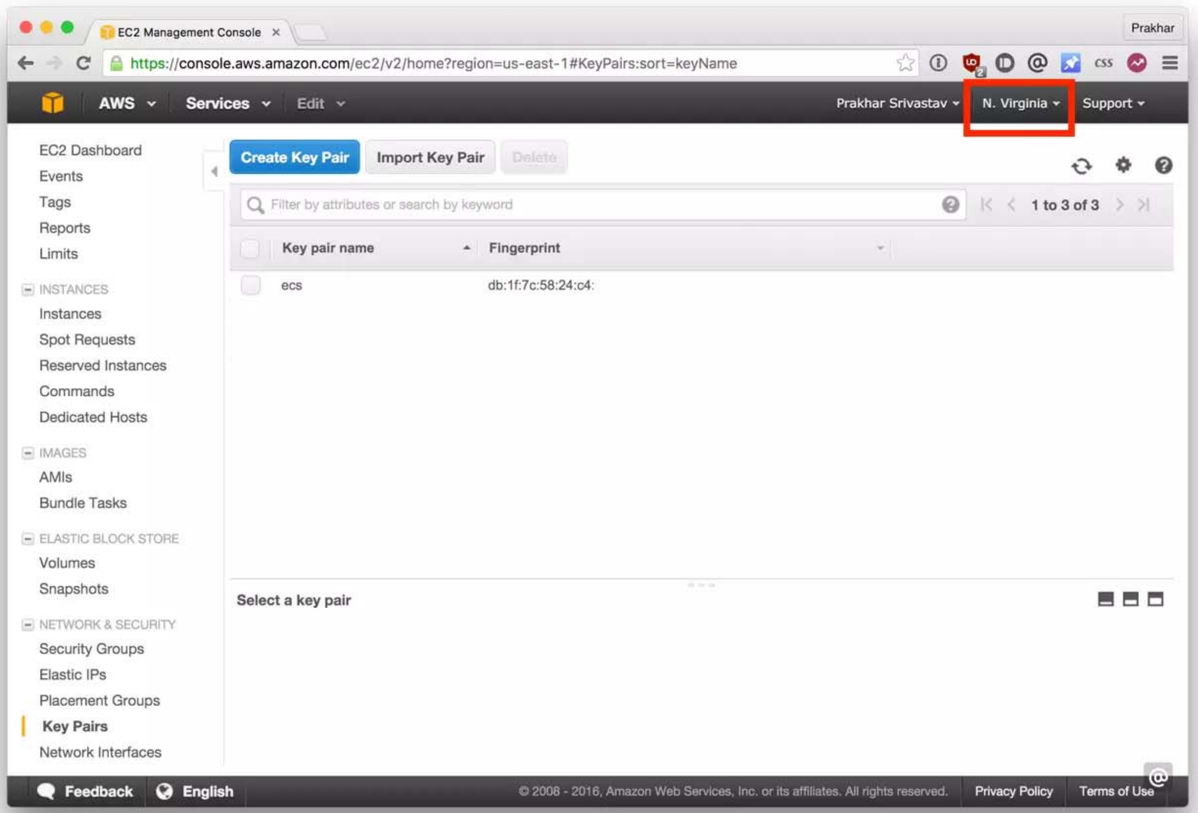
AWS ECS is a scalable and super flexible container management service that supports Docker containers. It allows you to operate a Docker cluster on top of EC2 instances via an easy-to-use API. Where Beanstalk came with reasonable defaults, ECS allows you to completely tune your environment as per your needs. This makes ECS, in my opinion, quite complex to get started with.

Luckily for us, ECS has a friendly CLI tool that understands Docker Compose files and automatically provisions the cluster on ECS! Since we already have a functioning `docker-compose.yml` it should not take a lot of effort in getting up and running on AWS. So let's get started!

The first step is to install the CLI. Instructions to install the CLI on both Mac and Linux are explained very clearly in the official docs. Go ahead, install the CLI and when you are done, verify the install by running

```
$ ecs-cli --version
ecs-cli version 0.1.0 (*cbdc2d5)
```

The first step is to get a keypair which we'll be using to log into the instances. Head over to your EC2 Console and create a new keypair. Download the keypair and store it in a safe location. Another thing to note before you move away from this screen is the region name. In my case, I have named my key - `ecs` and set my region as `us-east-1`. This is what I'll assume for the rest of this walkthrough.



Key pair name	Fingerprint
ecs	db:1f:7c:58:24:c4:

© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.

The next step is to configure the CLI.

```
$ ecs-cli configure --region us-east-1 --cluster foodtrucks
INFO[0000] Saved ECS CLI configuration for cluster (foodtrucks)
```

We provide the `configure` command with the region name we want our cluster to reside in and a cluster name. Make sure you provide the **same region name** that you used when creating the keypair. If you've not configured the AWS CLI on your computer before, you can use the official guide, which explains everything in great detail on how to get everything going.

The next step enables the CLI to create a CloudFormation template.

```
$ ecs-cli up --keypair ecs --capability-iam --size 2 --instance-type t2.micro
INFO[0000] Created cluster                               cluster=foodtrucks
INFO[0001] Waiting for your cluster resources to be created
INFO[0001] Cloudformation stack status                   stackStatus=CREATE_IN_PROGRESS
INFO[0061] Cloudformation stack status                   stackStatus=CREATE_IN_PROGRESS
INFO[0122] Cloudformation stack status                   stackStatus=CREATE_IN_PROGRESS
INFO[0182] Cloudformation stack status                   stackStatus=CREATE_IN_PROGRESS
INFO[0242] Cloudformation stack status                   stackStatus=CREATE_IN_PROGRESS
```

Here we provide the name of the keypair we downloaded initially (`ecs` in my case), the number of instances that we want to use (`--size`) and the type of instances that we want the containers to run on. The `--capability-iam` flag tells the CLI that we acknowledge that this command may create IAM resources.

The last and final step is where we'll use our `docker-compose.yml` file. We'll need to make a tiny change, so instead of modifying the original, let's make a copy of it and call it `aws-compose.yml`. The contents of this file (after making the changes) look like (below) -

```
es:
  image: elasticsearch
  cpu_shares: 100
  mem_limit: 262144000
web:
  image: prakhar1989/foodtrucks-web
  cpu_shares: 100
  mem_limit: 262144000
  ports:
    - "80:5000"
  links:
    - es
```

The only changes we made from the original `docker-compose.yml` are of providing the `mem_limit` and `cpu_shares` values for each container. We also got rid of the `version` and the `services` key, since AWS doesn't yet support version 2 of Compose file format. Since our apps will run on `t2.micro` instances, we allocate 250mb of memory. Another thing we need to do before we move onto the next step is to publish our image on Docker Hub. As of this writing, ecs-cli **does not** support the `build` command - which is supported perfectly by Docker Compose.

```
$ docker push prakhar1989/foodtrucks-web
```

Great! Now let's run the final command that will deploy our app on ECS!

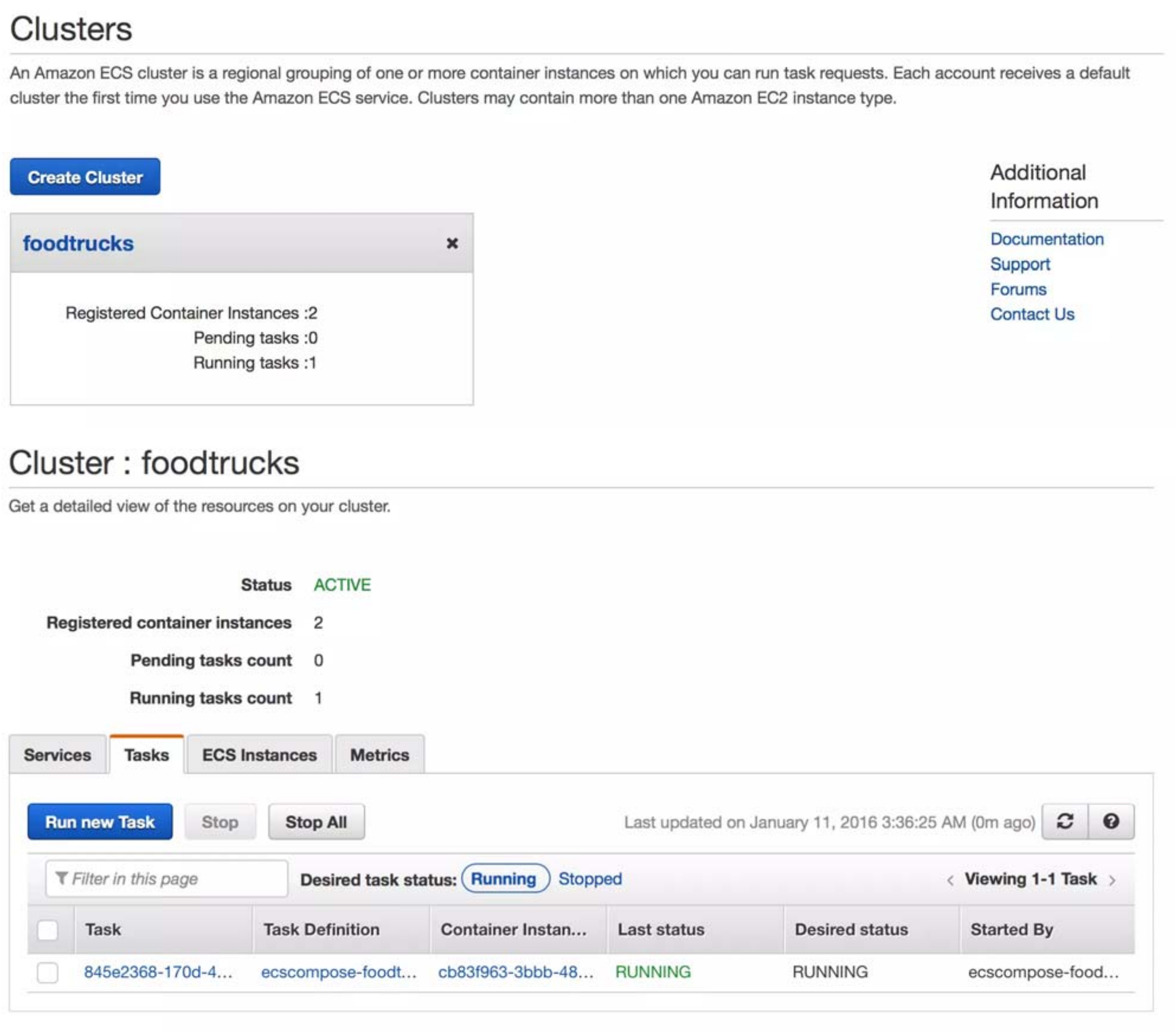
```
$ ecs-cli compose --file aws-compose.yml up
INFO[0000] Using ECS task definition                        TaskDefinition=ecscompose-foodtruck
INFO[0000] Starting container...                            container=845e2368-170d-44a7-bf9f-8
INFO[0000] Starting container...                            container=845e2368-170d-44a7-bf9f-8
INFO[0000] Describe ECS container status                    container=845e2368-170d-44a7-bf9f-8
INFO[0000] Describe ECS container status                    container=845e2368-170d-44a7-bf9f-8
INFO[0036] Describe ECS container status                    container=845e2368-170d-44a7-bf9f-8
INFO[0048] Describe ECS container status                    container=845e2368-170d-44a7-bf9f-8
INFO[0048] Describe ECS container status                    container=845e2368-170d-44a7-bf9f-8
INFO[0060] Started container...                             container=845e2368-170d-44a7-bf9f-8
INFO[0060] Started container...                             container=845e2368-170d-44a7-bf9f-8
```

It's not a coincidence that the invocation above looks similar to the one we used with **Docker Compose**. The `--file` argument is used to override the default file (`docker-compose.yml`) that the CLI will read. If everything went well, you should see a `desiredStatus=RUNNING lastStatus=RUNNING` as the last line.

Awesome! Our app is live, but how can we access it?

```
ecs-cli ps
Name                                          State    Ports                       TaskDefinition
845e2368-170d-44a7-bf9f-84c7fcd9ae29/web      RUNNING  54.86.14.14:80->5000/tcp    ecscompose-food
845e2368-170d-44a7-bf9f-84c7fcd9ae29/es       RUNNING                              ecscompose-food
```

Go ahead and open http://54.86.14.14 in your browser and you should see the Food Trucks in all its black-yellow glory! Since we're on the topic, let's see how our AWS ECS console looks.



Clusters

An Amazon ECS cluster is a regional grouping of one or more container instances on which you can run task requests. Each account receives a default cluster the first time you use the Amazon ECS service. Clusters may contain more than one Amazon EC2 instance type.

foodtrucks

Registered Container Instances :2
Pending tasks :0
Running tasks :1

Cluster : foodtrucks

Get a detailed view of the resources on your cluster.

Status	ACTIVE
Registered container instances	2
Pending tasks count	0
Running tasks count	1

Last updated on January 11, 2016 3:36:25 AM (0m ago)

Filter in this page Desired task status: Running Stopped Viewing 1-1 Task

Task	Task Definition	Container Instan...	Last status	Desired status	Started By
845e2368-170d-4...	ecscompose-foodt...	cb83f963-3bbb-48...	RUNNING	RUNNING	ecscompose-food...

We can see above that our ECS cluster called 'foodtrucks' was created and is now running 1 task with 2 container instances. Spend some time browsing this console to get a hang of all the options that are here.

So there you have it. With just a few commands we were able to deploy our awesome app on the AWS cloud!

CONCLUSION

And that's a wrap! After a long, exhaustive but fun tutorial you are now ready to take the container world by storm! If you followed along till the very end then you should definitely be proud of yourself. You learnt how to setup Docker, run your own containers, play with static and dynamic websites and most importantly got hands on experience with deploying your applications to the cloud!

I hope that finishing this tutorial makes you more confident in your abilities to deal with servers. When you have an idea of building your next app, you can be sure that you'll be able to get it in front of people with minimal effort.

Next Steps

Your journey into the container world has just started! My goal with this tutorial was to whet your appetite and show you the power of Docker. In the sea of new technology, it can be hard to navigate the waters alone and tutorials such as this one can provide a helping hand. This is the Docker tutorial I wish I had when I was starting out. Hopefully it served its purpose of getting you excited about containers so that you no longer have to watch the action from the sides.

Below are a few additional resources that will be beneficial. For your next project, I strongly encourage you to use Docker. Keep in mind - practice makes perfect!

Additional Resources

- Awesome Docker
- Hello Docker Workshop
- Why Docker
- Docker Weekly and archives
- Codeship Blog

Off you go, young padawan!

Give Feedback

Now that the tutorial is over, it's my turn to ask questions. How did you like the tutorial? Did you find the tutorial to be a complete mess or did you have fun and learn something?

Send in your thoughts directly to me or just create an issue. I'm on Twitter, too, so if that's your deal, feel free to holler there!

I would totally love to hear about your experience with this tutorial. Give suggestions on how to make this better or let me know about my mistakes. I want this tutorial to be one of the best introductory tutorials on the web and I can't do it without your help.