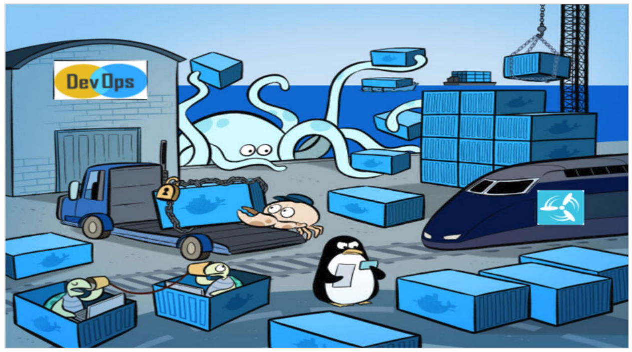

Figure 1: Containers are portable

Part I

Docker concepts

Docker is a platform for developers and sysadmins to **develop, deploy, and run** applications with containers. The use of Linux containers to deploy applications is called *containerization*. Containers are not new, but their use for easily deploying applications is.

Containerization is increasingly popular because containers are:

- Flexible: Even the most complex applications can be containerized.
- Lightweight: Containers leverage and share the host kernel.
- Interchangeable: You can deploy updates and upgrades on-the-fly.
- Portable: You can build locally, deploy to the cloud, and run anywhere.
- Scalable: You can increase and automatically distribute container replicas.
- Stackable: You can stack services vertically and on-the-fly.

Images and containers

A container is launched by running an image. An **image** is an executable package that includes everything needed to run an application–the code, a runtime, libraries, environment variables, and configuration files.

A **container** is a runtime instance of an image – what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, `docker ps`, just as you would in Linux.

Containers and virtual machines

A **container** runs *natively* on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, taking no more memory than any other executable, making it lightweight.

By contrast, a **virtual machine** (VM) runs a full-blown "guest" operating system with *virtual* access to host resources through a hypervisor. In general, VMs provide an environment with more resources than most applications need.

Prepare your Docker environment

Install a maintained version of Docker Community Edition (CE) or Enterprise Edition (EE) on a supported platform.

For full Kubernetes Integration

- Kubernetes on Docker Desktop for Mac is available in 17.12 Edge (mac45) or 17.12 Stable (mac46) and higher.

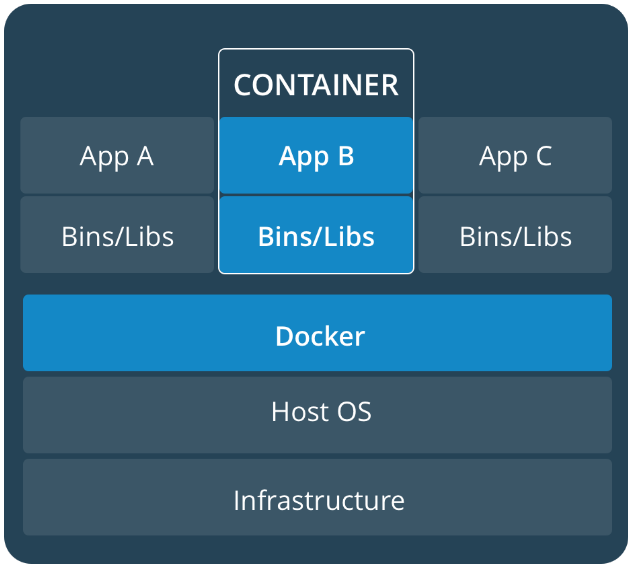

Figure 2: Container stack example

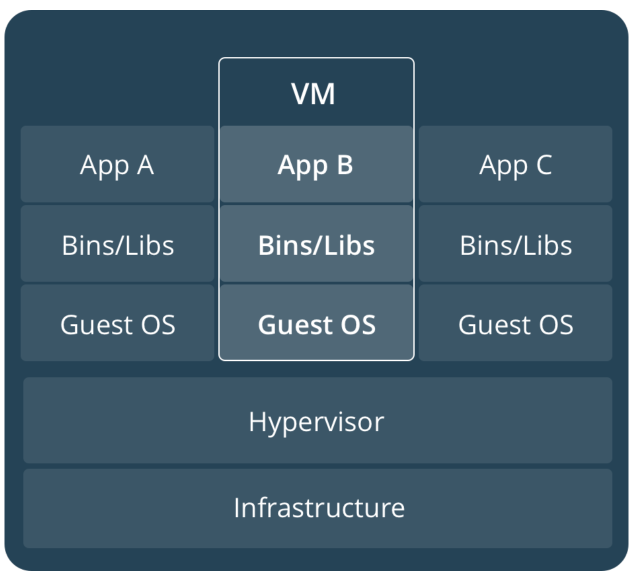

Figure 3: Virtual machine stack example

- Kubernetes on Docker Desktop for Windows is available in 18.02 Edge (win50) and higher edge channels only.

Install Docker

Test Docker version

1. Run `docker --version` and ensure that you have a supported version of Docker:

   ```
   docker --version

   Docker version 17.12.0-ce, build c97c6d6
   ```

2. Run `docker info` or (`docker version` without `--`) to view even more details about your docker installation:

   ```
   docker info

   Containers: 0
    Running: 0
    Paused: 0
    Stopped: 0
   Images: 0
   Server Version: 17.12.0-ce
   Storage Driver: overlay2
   ...
   ```

   To avoid permission errors (and the use of `sudo`), add your user to the `docker` group. Read more.

Test Docker installation

1. Test that your installation works by running the simple Docker image, hello-world{: target="_blank" class="_"}:

   ```
   docker run hello-world

   Unable to find image 'hello-world:latest' locally
   latest: Pulling from library/hello-world
   ca4f61b1923c: Pull complete
   Digest: sha256:ca0eeb6fb05351dfc8759c20733c91def84cb8007aa89a5bf606bc8b315b9fc7
   Status: Downloaded newer image for hello-world:latest

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

2. List the `hello-world` image that was downloaded to your machine:

   ```
   docker image ls
   ```

3. List the `hello-world` container (spawned by the image) which exits after displaying its message. If it were still running, you would not need the `--all` option:

   ```
   docker container ls --all

   CONTAINER ID     IMAGE           COMMAND      CREATED            STATUS
   54f4984ed6a8     hello-world     "/hello"     20 seconds ago     Exited (0) 19 seconds ago
   ```

Recap and cheat sheet

```
## List Docker CLI commands
docker
docker container --help

## Display Docker version and info
docker --version
docker version
docker info

## Execute Docker image
docker run hello-world

## List Docker images
docker image ls

## List Docker containers (running, all, all in quiet mode)
docker container ls
docker container ls --all
docker container ls -aq
```

Part II

Introduction

It's time to begin building an app the Docker way. We start at the bottom of the hierarchy of such app, a container, which this page covers. Above this level is a service, which defines how containers behave in production, covered in Part 3. Finally, at the top level is the stack, defining the interactions of all the services, covered in Part 5.

- Stack
- Services
- **Container** (you are here)

Your new development environment

In the past, if you were to start writing a Python app, your first order of business was to install a Python runtime onto your machine. But, that creates a situation where the environment on your machine needs to be perfect for your app to run as expected, and also needs to match your production environment.

With Docker, you can just grab a portable Python runtime as an image, no installation necessary. Then, your build can include the base Python image right alongside your app code, ensuring that your app, its dependencies, and the runtime, all travel together.

These portable images are defined by something called a `Dockerfile`.

Define a container with `Dockerfile`

`Dockerfile` defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to "copy in" to that environment. However, after doing that, you can expect that the build of your app defined in this `Dockerfile` behaves exactly the same wherever it runs.

Dockerfile

Create an empty directory on your local machine. Change directories (`cd`) into the new directory, create a file called `Dockerfile`, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

This `Dockerfile` refers to a couple of files we haven't created yet, namely `app.py` and `requirements.txt`. Let's create those next.

The app itself

Create two more files, `requirements.txt` and `app.py`, and put them in the same folder with the `Dockerfile`. This completes our app, which as you can see is quite simple. When the above `Dockerfile` is built into an image, `app.py` and `requirements.txt` is present because of that `Dockerfile`'s `COPY` command, and the output from `app.py` is accessible over HTTP thanks to the `EXPOSE` command.

requirements.txt

```
Flask
Redis
```

app.py

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
```

```
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Now we see that `pip install -r requirements.txt` installs the Flask and Redis libraries for Python, and the app prints the environment variable `NAME`, as well as the output of a call to `socket.gethostname()`. Finally, because Redis isn't running (as we've only installed the Python library, and not Redis itself), we should expect that the attempt to use it here fails and produces the error message.

> **Note**: Accessing the name of the host when inside a container retrieves the container ID, which is like the process ID for a running executable.

That's it! You don't need Python or anything in `requirements.txt` on your system, nor does building or running this image install them on your system. It doesn't seem like you've really set up an environment with Python and Flask, but you have.

Build the app

We are ready to build the app. Make sure you are still at the top level of your new directory. Here's what `ls` should show:

```
$ ls
Dockerfile      app.py          requirements.txt
```

Now run the build command. This creates a Docker image, which we're going to name using the `--tag` option. Use `-t` if you want to use the shorter option.

```
docker build --tag=friendlyhello .
```

Where is your built image? It's in your machine's local Docker image registry:

```
$ docker image ls

REPOSITORY            TAG                 IMAGE ID
friendlyhello         latest              326387cea398
```

Note how the tag defaulted to `latest`. The full syntax for the tag option would be something like `--tag=friendlyhello:v0.0.1`.

> Troubleshooting for Linux users
>
> *Proxy server settings*
>
> Proxy servers can block connections to your web app once it's up and running. If you are behind a proxy server, add the following lines to your Dockerfile, using the `ENV` command to specify the host and port for your proxy servers:
>
> ```
> # Set proxy server, replace host:port with values for your servers
> ENV http_proxy host:port
> ENV https_proxy host:port
> ```

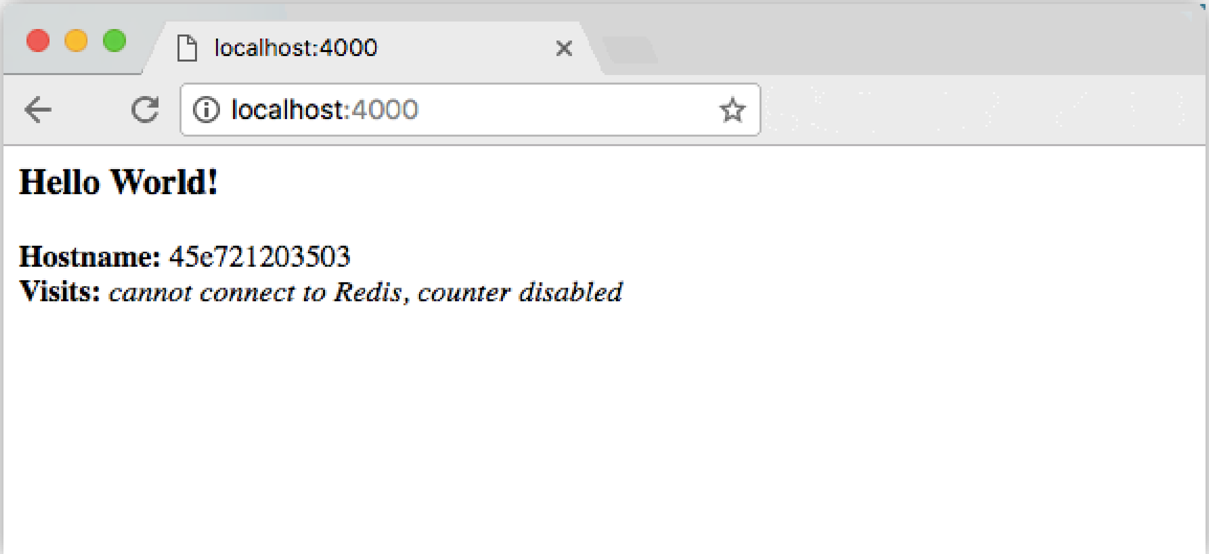

Figure 4: Hello World in browser

> *DNS settings*
>
> DNS misconfigurations can generate problems with `pip`. You need to set your own DNS server address to make `pip` work properly. You might want to change the DNS settings of the Docker daemon. You can edit (or create) the configuration file at `/etc/docker/daemon.json` with the `dns` key, as following:
>
> ```
> {
>   "dns": ["your_dns_address", "8.8.8.8"]
> }
> ```
>
> In the example above, the first element of the list is the address of your DNS server. The second item is Google's DNS which can be used when the first one is not available.
>
> Before proceeding, save `daemon.json` and restart the docker service.
>
> `sudo service docker restart`
>
> Once fixed, retry to run the `build` command.

Run the app

Run the app, mapping your machine's port 4000 to the container's published port 80 using `-p`:

```
docker run -p 4000:80 friendlyhello
```

You should see a message that Python is serving your app at `http://0.0.0.0:80`. But that message is coming from inside the container, which doesn't know you mapped port 80 of that container to 4000, making the correct URL `http://localhost:4000`.

Go to that URL in a web browser to see the display content served up on a web page.

> **Note**: If you are using Docker Toolbox on Windows 7, use the Docker Machine IP instead of `localhost`. For example, http://192.168.99.100:4000/. To find the IP address, use the command `docker-machine ip`.

You can also use the `curl` command in a shell to view the same content.

```
$ curl http://localhost:4000

<h3>Hello World!</h3><b>Hostname:</b> 8fc990912a14<br/><b>Visits:</b> <i>cannot connect to Redis, count
```

This port remapping of `4000:80` demonstrates the difference between `EXPOSE` within the `Dockerfile` and what the `publish` value is set to when running `docker run -p`. In later steps, map port 4000 on the host to port 80 in the container and use `http://localhost`.

Hit `CTRL+C` in your terminal to quit.

> On Windows, explicitly stop the container
>
> On Windows systems, `CTRL+C` does not stop the container. So, first type `CTRL+C` to get the prompt back (or open another shell), then type `docker container ls` to list the running containers, followed by `docker container stop <Container NAME or ID>` to stop the container. Otherwise, you get an error response from the daemon when you try to re-run the container in the next step.

Now let's run the app in the background, in detached mode:

```
docker run -d -p 4000:80 friendlyhello
```

You get the long container ID for your app and then are kicked back to your terminal. Your container is running in the background. You can also see the abbreviated container ID with `docker container ls` (and both work interchangeably when running commands):

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED
1fa4ab2cf395        friendlyhello       "python app.py"     28 seconds ago
```

Notice that `CONTAINER ID` matches what's on `http://localhost:4000`.

Now use `docker container stop` to end the process, using the `CONTAINER ID`, like so:

```
docker container stop 1fa4ab2cf395
```

Share your image

To demonstrate the portability of what we just created, let's upload our built image and run it somewhere else. After all, you need to know how to push to registries when you want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The `docker` CLI uses Docker's public registry by default.

> **Note**: We use Docker's public registry here just because it's free and pre-configured, but there are many public ones to choose from, and you can even set up your own private registry using Docker Trusted Registry.

Log in with your Docker ID

If you don't have a Docker account, sign up for one at hub.docker.com{: target="_blank" class="_" }. Make note of your username.

Log in to the Docker public registry on your local machine.

```
$ docker login
```

Tag the image

The notation for associating a local image with a repository on a registry is `username/repository:tag`. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. Give the repository and tag meaningful names for the context, such as `get-started:part2`. This puts the image in the `get-started` repository and tag it as `part2`.

Now, put it all together to tag the image. Run `docker tag image` with your username, repository, and tag names so that the image uploads to your desired destination. The syntax of the command is:

```
docker tag image username/repository:tag
```

For example:

```
docker tag friendlyhello gordon/get-started:part2
```

Run docker image ls to see your newly tagged image.

```
$ docker image ls

REPOSITORY           TAG              IMAGE ID          CREATED          SIZE
friendlyhello        latest           d9e555c53008      3 minutes ago    195MB
gordon/get-started     part2            d9e555c53008      3 minutes ago    195MB
python               2.7-slim         1c7128a655f6      5 days ago       183MB
...
```

Publish the image

Upload your tagged image to the repository:

```
docker push username/repository:tag
```

Once complete, the results of this upload are publicly available. If you log in to Docker Hub, you see the new image there, with its pull command.

Pull and run the image from the remote repository

From now on, you can use `docker run` and run your app on any machine with this command:

```
docker run -p 4000:80 username/repository:tag
```

If the image isn't available locally on the machine, Docker pulls it from the repository.

```
$ docker run -p 4000:80 gordon/get-started:part2
Unable to find image 'gordon/get-started:part2' locally
part2: Pulling from gordon/get-started
10a267c67f42: Already exists
f68a39a6a5e4: Already exists
9beaffc0cf19: Already exists
3c1fe835fb6b: Already exists
4c9f1fa8fcb8: Already exists
ee7d8f576a14: Already exists
fbccdcced46e: Already exists
Digest: sha256:0601c866aab2adcc6498200efd0f754037e909e5fd42069adeff72d1e2439068
Status: Downloaded newer image for gordon/get-started:part2
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

No matter where `docker run` executes, it pulls your image, along with Python and all the dependencies from `requirements.txt`, and runs your code. It all travels together in a neat little package, and you don't need to install anything on the host machine for Docker to run it.

Conclusion of part two

That's all for this page. In the next section, we learn how to scale our application by running this container in a **service**.

Continue to Part 3 >>

Recap and cheat sheet (optional)

Here's a terminal recording of what was covered on this page:

Here is a list of the basic Docker commands from this page, and some related ones if you'd like to explore a bit before moving on.

```
docker build -t friendlyhello .  # Create image using this directory's Dockerfile
docker run -p 4000:80 friendlyhello  # Run "friendlyname" mapping port 4000 to 80
docker run -d -p 4000:80 friendlyhello         # Same thing, but in detached mode
docker container ls                                # List all running containers
docker container ls -a             # List all containers, even those not running
docker container stop <hash>           # Gracefully stop the specified container
docker container kill <hash>         # Force shutdown of the specified container
docker container rm <hash>        # Remove specified container from this machine
docker container rm $(docker container ls -a -q)         # Remove all containers
docker image ls -a                             # List all images on this machine
docker image rm <image id>            # Remove specified image from this machine
docker image rm $(docker image ls -a -q)   # Remove all images from this machine
docker login             # Log in this CLI session using your Docker credentials
docker tag <image> username/repository:tag  # Tag <image> for upload to registry
docker push username/repository:tag            # Upload tagged image to registry
docker run username/repository:tag                   # Run image from a registry
```

Part III

Introduction

In part 3, we scale our application and enable load-balancing. To do this, we must go one level up in the hierarchy of a distributed application: the **service**.

- Stack
- **Services** (you are here)
- Container (covered in part 2)

About services

In a distributed application, different pieces of the app are called "services". For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

Services are really just "containers in production." A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Luckily it's very easy to define, run, and scale services with the Docker platform – just write a `docker-compose.yml` file.

Your first `docker-compose.yml` file

A `docker-compose.yml` file is a YAML file that defines how Docker containers should behave in production.

`docker-compose.yml`

Save this file as `docker-compose.yml` wherever you want. Be sure you have pushed the image you created in Part 2 to a registry, and update this `.yml` by replacing `username/repo:tag` with your image details.

```yaml
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "4000:80"
    networks:
      - webnet
networks:
  webnet:
```

This `docker-compose.yml` file tells Docker to do the following:

- Pull the image we uploaded in step 2 from the registry.
- Run 5 instances of that image as a service called `web`, limiting each one to use, at most, 10% of the CPU (across all cores), and 50MB of RAM.
- Immediately restart containers if one fails.
- Map port 4000 on the host to `web`'s port 80.
- Instruct `web`'s containers to share port 80 via a load-balanced network called `webnet`. (Internally, the containers themselves publish to `web`'s port 80 at an ephemeral port.)
- Define the `webnet` network with the default settings (which is a load-balanced overlay network).

Run your new load-balanced app

Before we can use the `docker stack deploy` command we first run:

`docker swarm init`

> **Note**: We get into the meaning of that command in part 4. If you don't run `docker swarm init` you get an error that "this node is not a swarm manager."

Now let's run it. You need to give your app a name. Here, it is set to `getstartedlab`:

`docker stack deploy -c docker-compose.yml getstartedlab`

Our single service stack is running 5 container instances of our deployed image on one host. Let's investigate.

Get the service ID for the one service in our application:

`docker service ls`

Look for output for the `web` service, prepended with your app name. If you named it the same as shown in this example, the name is `getstartedlab_web`. The service ID is listed as well, along with the number of replicas, image name, and exposed ports.

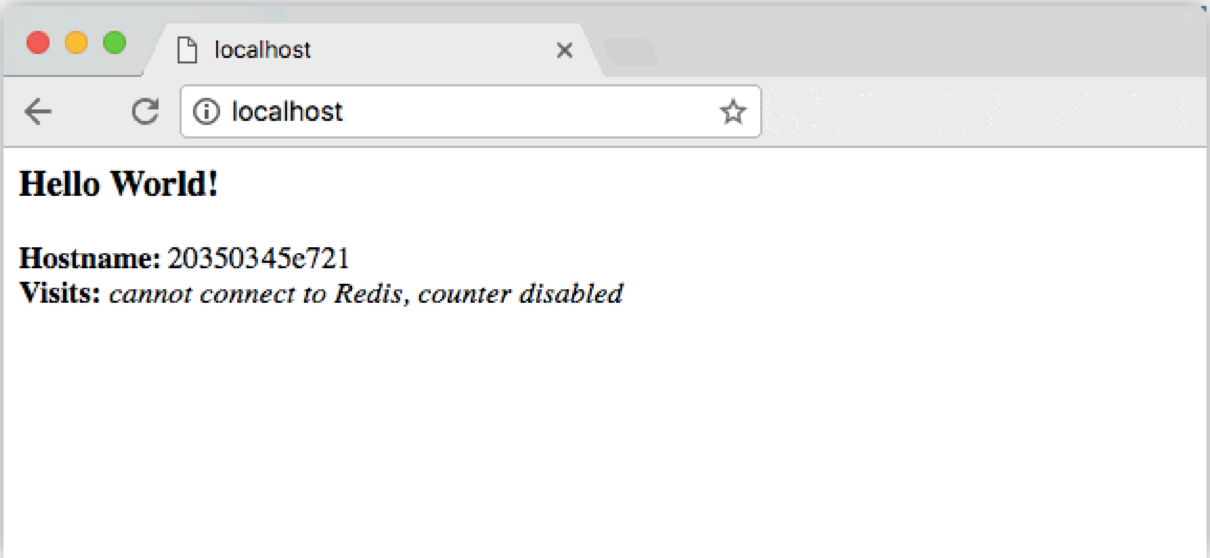

Figure 5: Hello World in browser

Alternatively, you can run `docker stack services`, followed by the name of your stack. The following example command lets you view all services associated with the `getstartedlab` stack:

```
docker stack services getstartedlab
ID                  NAME                MODE                REPLICAS            IMAGE               POR
bqpve1djnk0x        getstartedlab_web   replicated          5/5                 username/repo:tag   *:4
```

A single container running in a service is called a **task**. Tasks are given unique IDs that numerically increment, up to the number of `replicas` you defined in `docker-compose.yml`. List the tasks for your service:

```
docker service ps getstartedlab_web
```

Tasks also show up if you just list all the containers on your system, though that is not filtered by service:

```
docker container ls -q
```

You can run `curl -4 http://localhost:4000` several times in a row, or go to that URL in your browser and hit refresh a few times.

Either way, the container ID changes, demonstrating the load-balancing; with each request, one of the 5 tasks is chosen, in a round-robin fashion, to respond. The container IDs match your output from the previous command (`docker container ls -q`).

To view all tasks of a stack, you can run `docker stack ps` followed by your app name, as shown in the following example:

```
docker stack ps getstartedlab
ID                  NAME                  IMAGE               NODE                DESIRED STATE       C
uwiaw67sc0eh        getstartedlab_web.1   username/repo:tag   docker-desktop      Running             R
sk50xbhmcae7        getstartedlab_web.2   username/repo:tag   docker-desktop      Running             R
c4uuw5i6h02j        getstartedlab_web.3   username/repo:tag   docker-desktop      Running             R
0dyb70ixu25s        getstartedlab_web.4   username/repo:tag   docker-desktop      Running             R
aocrb88ap8b0        getstartedlab_web.5   username/repo:tag   docker-desktop      Running             R
```

Running Windows 10?

> Windows 10 PowerShell should already have `curl` available, but if not you can grab a Linux terminal emulator like Git BASH{: target="_blank" class="_"}, or download wget for Windows which is very similar.
>
> Slow response times?
>
> Depending on your environment's networking configuration, it may take up to 30 seconds for the containers to respond to HTTP requests. This is not indicative of Docker or swarm performance, but rather an unmet Redis dependency that we address later in the tutorial. For now, the visitor counter isn't working for the same reason; we haven't yet added a service to persist data.

Scale the app

You can scale the app by changing the `replicas` value in `docker-compose.yml`, saving the change, and re-running the `docker stack deploy` command:

```
docker stack deploy -c docker-compose.yml getstartedlab
```

Docker performs an in-place update, no need to tear the stack down first or kill any containers.

Now, re-run `docker container ls -q` to see the deployed instances reconfigured. If you scaled up the replicas, more tasks, and hence, more containers, are started.

Take down the app and the swarm

- Take the app down with `docker stack rm`:

```
shell   docker stack rm getstartedlab
```

- Take down the swarm.

```
docker swarm leave --force
```

It's as easy as that to stand up and scale your app with Docker. You've taken a huge step towards learning how to run containers in production. Up next, you learn how to run this app as a bonafide swarm on a cluster of Docker machines.

> **Note**: Compose files like this are used to define applications with Docker, and can be uploaded to cloud providers using Docker Cloud, or on any hardware or cloud provider you choose with Docker Enterprise Edition.

Recap and cheat sheet (optional)

Here's a terminal recording of what was covered on this page:

To recap, while typing `docker run` is simple enough, the true implementation of a container in production is running it as a service. Services codify a container's behavior in a Compose file, and this file can be used to scale, limit, and redeploy our app. Changes to the service can be applied in place, as it runs, using the same command that launched the service: `docker stack deploy`.

Some commands to explore at this stage:

```
docker stack ls                                            # List stacks or apps
docker stack deploy -c <composefile> <appname>  # Run the specified Compose file
docker service ls                 # List running services associated with an app
docker service ps <service>                  # List tasks associated with an app
docker inspect <task or container>                   # Inspect task or container
docker container ls -q                                      # List container IDs
docker stack rm <appname>                             # Tear down an application
docker swarm leave --force      # Take down a single node swarm from the manager
```

Part IV

Introduction

In part 3, you took an app you wrote in part 2, and defined how it should run in production by turning it into a service, scaling it up 5x in the process.

Here in part 4, you deploy this application onto a cluster, running it on multiple machines. Multi-container, multi-machine applications are made possible by joining multiple machines into a “Dockerized” cluster called a **swarm**.

Understanding Swarm clusters

A swarm is a group of machines that are running Docker and joined into a cluster. After that has happened, you continue to run the Docker commands you’re used to, but now they are executed on a cluster by a **swarm manager**. The machines in a swarm can be physical or virtual. After joining a swarm, they are referred to as **nodes**.

Swarm managers can use several strategies to run containers, such as “emptiest node” – which fills the least utilized machines with containers. Or “global”, which ensures that each machine gets exactly one instance of the specified container. You instruct the swarm manager to use these strategies in the Compose file, just like the one you have already been using.

Swarm managers are the only machines in a swarm that can execute your commands, or authorize other machines to join the swarm as **workers**. Workers are just there to provide capacity and do not have the authority to tell any other machine what it can and cannot do.

Up until now, you have been using Docker in a single-host mode on your local machine. But Docker also can be switched into **swarm mode**, and that’s what enables the use of swarms. Enabling swarm mode instantly makes the current machine a swarm manager. From then on, Docker runs the commands you execute on the swarm you’re managing, rather than just on the current machine.

Set up your swarm

A swarm is made up of multiple nodes, which can be either physical or virtual machines. The basic concept is simple enough: run `docker swarm init` to enable swarm mode and make your current machine a swarm manager, then run `docker swarm join` on other machines to have them join the swarm as workers. Choose a tab below to see how this plays out in various contexts. We use VMs to quickly create a two-machine cluster and turn it into a swarm.

Create a cluster

VMs on your local machine (Mac, Linux, Windows 7 and 8)

You need a hypervisor that can create virtual machines (VMs), so install Oracle VirtualBox for your machine’s OS.

> **Note**: If you are on a Windows system that has Hyper-V installed, such as Windows 10, there is no need to install VirtualBox and you should use Hyper-V instead. View the instructions for Hyper-V systems by clicking the Hyper-V tab above. If you are using Docker Toolbox, you should already have VirtualBox installed as part of it, so you are good to go.

Now, create a couple of VMs using `docker-machine`, using the VirtualBox driver:

```
docker-machine create --driver virtualbox myvm1
docker-machine create --driver virtualbox myvm2
```

{% endcapture %} {{ local-content | markdownify }}

{% capture localwin-content %}

VMs on your local machine (Windows 10)

First, quickly create a virtual switch for your virtual machines (VMs) to share, so they can connect to each other.

1. Launch Hyper-V Manager
2. Click **Virtual Switch Manager** in the right-hand menu
3. Click **Create Virtual Switch** of type **External**
4. Give it the name `myswitch`, and check the box to share your host machine's active network adapter

Now, create a couple of VMs using our node management tool, `docker-machine`:

```
docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" myvm1
docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" myvm2
```

{% endcapture %} {{ localwin-content | markdownify }}

List the VMs and get their IP addresses

You now have two VMs created, named `myvm1` and `myvm2`.

Use this command to list the machines and get their IP addresses.

```
docker-machine ls
```

Here is example output from this command.

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        ERRORS
myvm1   -        virtualbox   Running   tcp://192.168.99.100:2376           v17.06.2-ce
myvm2   -        virtualbox   Running   tcp://192.168.99.101:2376           v17.06.2-ce
```

Initialize the swarm and add nodes

The first machine acts as the manager, which executes management commands and authenticates workers to join the swarm, and the second is a worker.

You can send commands to your VMs using `docker-machine ssh`. Instruct `myvm1` to become a swarm manager with `docker swarm init` and look for output like this:

```
$ docker-machine ssh myvm1 "docker swarm init --advertise-addr <myvm1 ip>"
Swarm initialized: current node <node ID> is now a manager.

To add a worker to this swarm, run the following command:

  docker swarm join \
  --token <token> \
  <myvm ip>:<port>

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

> Ports 2377 and 2376
>
> Always run `docker swarm init` and `docker swarm join` with port 2377 (the swarm management port), or no port at all and let it take the default.
>
> The machine IP addresses returned by `docker-machine ls` include port 2376, which is the Docker daemon port. Do not use this port or you may experience errors{: target="_blank" class="_"}.
>
> Having trouble using SSH? Try the –native-ssh flag

Docker Machine has the option to let you use your own system's SSH, if for some reason you're having trouble sending commands to your Swarm manager. Just specify the `--native-ssh` flag when invoking the `ssh` command:

```
docker-machine --native-ssh ssh myvm1 ...
```

As you can see, the response to `docker swarm init` contains a pre-configured `docker swarm join` command for you to run on any nodes you want to add. Copy this command, and send it to `myvm2` via `docker-machine ssh` to have `myvm2` join your new swarm as a worker:

```
$ docker-machine ssh myvm2 "docker swarm join \
--token <token> \
<ip>:2377"

This node joined a swarm as a worker.
```

Congratulations, you have created your first swarm!

Run `docker node ls` on the manager to view the nodes in this swarm:

```
$ docker-machine ssh myvm1 "docker node ls"
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS
brtu9urxwfd5j0zrmkubhpkbd     myvm2               Ready               Active
rihwohkh3ph38fhillhhb84sk *   myvm1               Ready               Active              Leader
```

Leaving a swarm

If you want to start over, you can run `docker swarm leave` from each node.

Deploy your app on the swarm cluster

The hard part is over. Now you just repeat the process you used in part 3 to deploy on your new swarm. Just remember that only swarm managers like `myvm1` execute Docker commands; workers are just for capacity.

Configure a `docker-machine` shell to the swarm manager

So far, you've been wrapping Docker commands in `docker-machine ssh` to talk to the VMs. Another option is to run `docker-machine env <machine>` to get and run a command that configures your current shell to talk to the Docker daemon on the VM. This method works better for the next step because it allows you to use your local `docker-compose.yml` file to deploy the app "remotely" without having to copy it anywhere.

Type `docker-machine env myvm1`, then copy-paste and run the command provided as the last line of the output to configure your shell to talk to `myvm1`, the swarm manager.

The commands to configure your shell differ depending on whether you are Mac, Linux, or Windows.

Docker machine shell environment on Mac or Linux

Run `docker-machine env myvm1` to get the command to configure your shell to talk to `myvm1`.

```
$ docker-machine env myvm1
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/sam/.docker/machine/machines/myvm1"
export DOCKER_MACHINE_NAME="myvm1"
# Run this command to configure your shell:
# eval $(docker-machine env myvm1)
```

Run the given command to configure your shell to talk to `myvm1`.

```
eval $(docker-machine env myvm1)
```

Run `docker-machine ls` to verify that `myvm1` is now the active machine, as indicated by the asterisk next to it.

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        ERRORS
myvm1   *        virtualbox   Running   tcp://192.168.99.100:2376           v17.06.2-ce
myvm2   -        virtualbox   Running   tcp://192.168.99.101:2376           v17.06.2-ce
```

Docker machine shell environment on Windows

Run `docker-machine env myvm1` to get the command to configure your shell to talk to `myvm1`.

```
PS C:\Users\sam\sandbox\get-started> docker-machine env myvm1
$Env:DOCKER_TLS_VERIFY = "1"
$Env:DOCKER_HOST = "tcp://192.168.203.207:2376"
$Env:DOCKER_CERT_PATH = "C:\Users\sam\.docker\machine\machines\myvm1"
$Env:DOCKER_MACHINE_NAME = "myvm1"
$Env:COMPOSE_CONVERT_WINDOWS_PATHS = "true"
# Run this command to configure your shell:
# & "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression
```

Run the given command to configure your shell to talk to `myvm1`.

```
& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression
```

Run `docker-machine ls` to verify that `myvm1` is the active machine as indicated by the asterisk next to it.

```
PS C:PATH> docker-machine ls
NAME    ACTIVE   DRIVER   STATE     URL                          SWARM   DOCKER        ERRORS
myvm1   *        hyperv   Running   tcp://192.168.203.207:2376           v17.06.2-ce
myvm2   -        hyperv   Running   tcp://192.168.200.181:2376           v17.06.2-ce
```

Deploy the app on the swarm manager

Now that you have `myvm1`, you can use its powers as a swarm manager to deploy your app by using the same `docker stack deploy` command you used in part 3 to `myvm1`, and your local copy of `docker-compose.yml`.. This command may take a few seconds to complete and the deployment takes some time to be available. Use the `docker service ps <service_name>` command on a swarm manager to verify that all services have been redeployed.

You are connected to `myvm1` by means of the `docker-machine` shell configuration, and you still have access to the files on your local host. Make sure you are in the same directory as before, which includes the `docker-compose.yml` file you created in part 3.

Just like before, run the following command to deploy the app on `myvm1`.

```
docker stack deploy -c docker-compose.yml getstartedlab
```

And that's it, the app is deployed on a swarm cluster!

> **Note**: If your image is stored on a private registry instead of Docker Hub, you need to be logged in using `docker login <your-registry>` and then you need to add the `--with-registry-auth` flag to the above command. For example:
>
> ```
> docker login registry.example.com
>
> docker stack deploy --with-registry-auth -c docker-compose.yml getstartedlab
> ```
>
> This passes the login token from your local client to the swarm nodes where the service is deployed, using the encrypted WAL logs. With this information, the nodes are able to log into the registry and pull the image.

Now you can use the same docker commands you used in part 3. Only this time notice that the services (and associated containers) have been distributed between both `myvm1` and `myvm2`.

```
$ docker stack ps getstartedlab

ID            NAME                  IMAGE                       NODE   DESIRED STATE
jq2g3qp8nzwx  getstartedlab_web.1   gordon/get-started:part2  myvm1  Running
88wgshobzoxl  getstartedlab_web.2   gordon/get-started:part2  myvm2  Running
vbb1qbkb0o2z  getstartedlab_web.3   gordon/get-started:part2  myvm2  Running
ghii74p9budx  getstartedlab_web.4   gordon/get-started:part2  myvm1  Running
0prmarhavs87  getstartedlab_web.5   gordon/get-started:part2  myvm2  Running
```

> Connecting to VMs with `docker-machine env` and `docker-machine ssh`
>
> - To set your shell to talk to a different machine like `myvm2`, simply re-run `docker-machine env` in the same or a different shell, then run the given command to point to `myvm2`. This is always specific to the current shell. If you change to an unconfigured shell or open a new one, you need to re-run the commands. Use `docker-machine ls` to list machines, see what state they are in, get IP addresses, and find out which one, if any, you are connected to. To learn more, see the Docker Machine getting started topics.
> - Alternatively, you can wrap Docker commands in the form of `docker-machine ssh <machine> "<command>"`, which logs directly into the VM but doesn't give you immediate access to files on your local host.
> - On Mac and Linux, you can use `docker-machine scp <file> <machine>:~` to copy files across machines, but Windows users need a Linux terminal emulator like Git Bash{: target="_blank" class="_"} for this to work.
>
> This tutorial demos both `docker-machine ssh` and `docker-machine env`, since these are available on all platforms via the `docker-machine` CLI.

Accessing your cluster

You can access your app from the IP address of **either** `myvm1` or `myvm2`.

The network you created is shared between them and load-balancing. Run `docker-machine ls` to get your VMs' IP addresses and visit either of them on a browser, hitting refresh (or just `curl` them).

There are five possible container IDs all cycling by randomly, demonstrating the load-balancing.

The reason both IP addresses work is that nodes in a swarm participate in an ingress **routing mesh**. This ensures that a service deployed at a certain port within your swarm always has that port reserved to itself, no matter what node is actually running the container. Here's a diagram of how a routing mesh for a service called `my-web` published at port `8080` on a three-node swarm would look:

> Having connectivity trouble?
>
> Keep in mind that to use the ingress network in the swarm, you need to have the following ports open between the swarm nodes before you enable swarm mode:
>
> - Port 7946 TCP/UDP for container network discovery.
> - Port 4789 UDP for the container ingress network.

Iterating and scaling your app

From here you can do everything you learned about in parts 2 and 3.

Scale the app by changing the `docker-compose.yml` file.

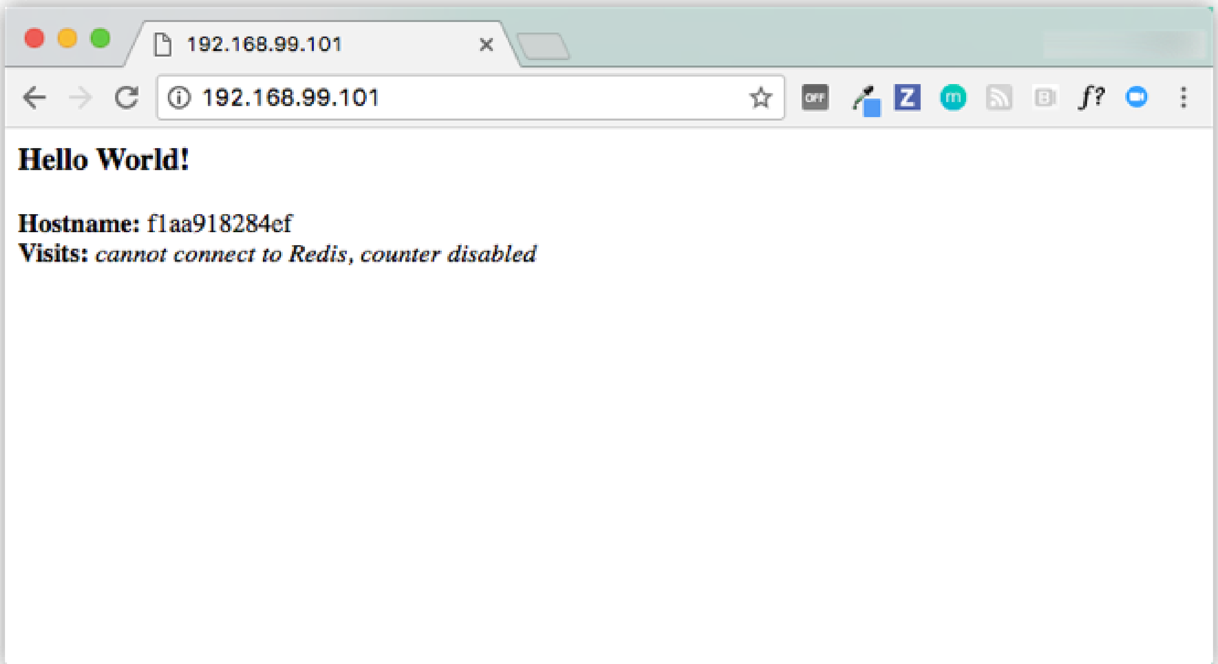

Figure 6: Hello World in browser

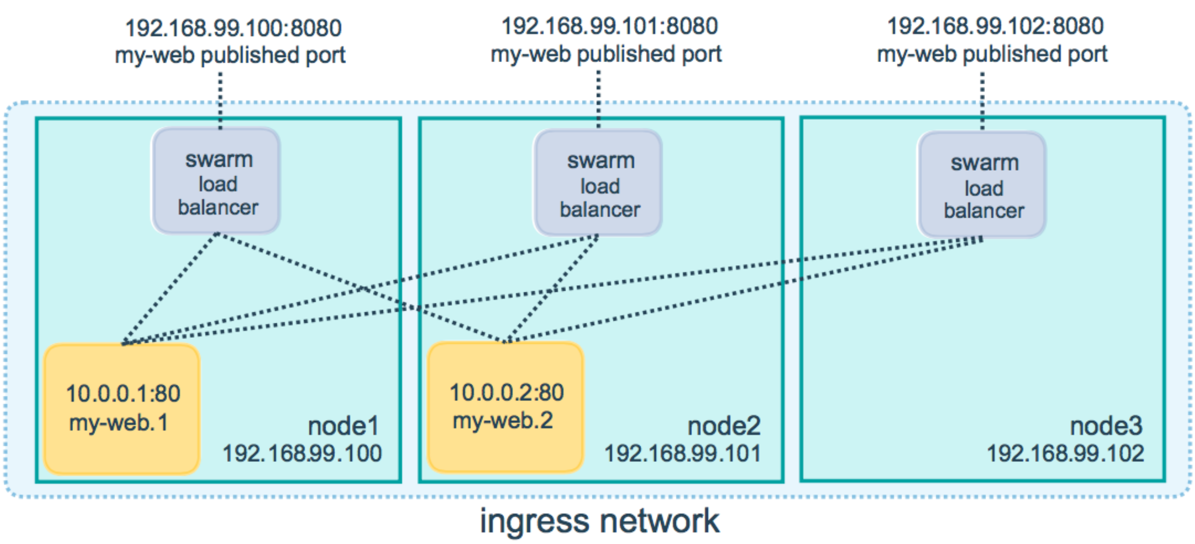

Figure 7: routing mesh diagram

Change the app behavior by editing code, then rebuild, and push the new image. (To do this, follow the same steps you took earlier to build the app and publish the image).

In either case, simply run `docker stack deploy` again to deploy these changes.

You can join any machine, physical or virtual, to this swarm, using the same `docker swarm join` command you used on `myvm2`, and capacity is added to your cluster. Just run `docker stack deploy` afterwards, and your app can take advantage of the new resources.

Cleanup and reboot

Stacks and swarms

You can tear down the stack with `docker stack rm`. For example:

```
docker stack rm getstartedlab
```

> Keep the swarm or remove it?
>
> At some point later, you can remove this swarm if you want to with `docker-machine ssh myvm2 "docker swarm leave"` on the worker and `docker-machine ssh myvm1 "docker swarm leave --force"` on the manager, but *you need this swarm for part 5, so keep it around for now.*

Unsetting docker-machine shell variable settings

You can unset the `docker-machine` environment variables in your current shell with the given command.

On **Mac or Linux** the command is:

```
shell   eval $(docker-machine env -u)
```

On **Windows** the command is:

```
shell   & "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env -u |
Invoke-Expression
```

This disconnects the shell from `docker-machine` created virtual machines, and allows you to continue working in the same shell, now using native `docker` commands (for example, on Docker Desktop for Mac or Docker Desktop for Windows). To learn more, see the Machine topic on unsetting environment variables.

Restarting Docker machines

If you shut down your local host, Docker machines stops running. You can check the status of machines by running `docker-machine ls`.

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL   SWARM   DOCKER    ERRORS
myvm1   -        virtualbox   Stopped                 Unknown
myvm2   -        virtualbox   Stopped                 Unknown
```

To restart a machine that's stopped, run:

```
docker-machine start <machine-name>
```

For example:

```
$ docker-machine start myvm1
Starting "myvm1"...
(myvm1) Check network to re-create if needed...
(myvm1) Waiting for an IP...
Machine "myvm1" was started.
Waiting for SSH to be available...
Detecting the provisioner...
```

```
Started machines may have new IP addresses. You may need to re-run the `docker-machine env` command.

$ docker-machine start myvm2
Starting "myvm2"...
(myvm2) Check network to re-create if needed...
(myvm2) Waiting for an IP...
Machine "myvm2" was started.
Waiting for SSH to be available...
Detecting the provisioner...
Started machines may have new IP addresses. You may need to re-run the `docker-machine env` command.
```

Recap and cheat sheet (optional)

Here's a terminal recording of what was covered on this page:

In part 4 you learned what a swarm is, how nodes in swarms can be managers or workers, created a swarm, and deployed an application on it. You saw that the core Docker commands didn't change from part 3, they just had to be targeted to run on a swarm master. You also saw the power of Docker's networking in action, which kept load-balancing requests across containers, even though they were running on different machines. Finally, you learned how to iterate and scale your app on a cluster.

Here are some commands you might like to run to interact with your swarm and your VMs a bit:

```
docker-machine create --driver virtualbox myvm1 # Create a VM (Mac, Win7, Linux)
docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" myvm1 # Win10
docker-machine env myvm1                # View basic information about your node
docker-machine ssh myvm1 "docker node ls"         # List the nodes in your swarm
docker-machine ssh myvm1 "docker node inspect <node ID>"        # Inspect a node
docker-machine ssh myvm1 "docker swarm join-token -q worker"   # View join token
docker-machine ssh myvm1   # Open an SSH session with the VM; type "exit" to end
docker node ls                # View nodes in swarm (while logged on to manager)
docker-machine ssh myvm2 "docker swarm leave"  # Make the worker leave the swarm
docker-machine ssh myvm1 "docker swarm leave -f" # Make master leave, kill swarm
docker-machine ls # list VMs, asterisk shows which VM this shell is talking to
docker-machine start myvm1            # Start a VM that is currently not running
docker-machine env myvm1      # show environment variables and command for myvm1
eval $(docker-machine env myvm1)         # Mac command to connect shell to myvm1
& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression   # Wi
docker stack deploy -c <file> <app>  # Deploy an app; command shell must be set to talk to manager (myvm
docker-machine scp docker-compose.yml myvm1:~ # Copy file to node's home dir (only required if you use s
docker-machine ssh myvm1 "docker stack deploy -c <file> <app>"   # Deploy an app using ssh (you must hav
eval $(docker-machine env -u)     # Disconnect shell from VMs, use native docker
docker-machine stop $(docker-machine ls -q)               # Stop all running VMs
docker-machine rm $(docker-machine ls -q) # Delete all VMs and their disk images
```

Part V

Introduction

In part 4, you learned how to set up a swarm, which is a cluster of machines running Docker, and deployed an application to it, with containers running in concert on multiple machines.

Here in part 5, you reach the top of the hierarchy of distributed applications: the **stack**. A stack is a group of interrelated services that share dependencies, and can be orchestrated and scaled together. A single stack is capable of defining and coordinating the functionality of an entire application (though very complex applications may want to use multiple stacks).

Some good news is, you have technically been working with stacks since part 3, when you created a Compose file and used `docker stack deploy`. But that was a single service stack running on a single host, which is not usually what takes place in production. Here, you can take what you've learned, make multiple services relate to each other, and run them on multiple machines.

You're doing great, this is the home stretch!

Add a new service and redeploy

It's easy to add services to our `docker-compose.yml` file. First, let's add a free visualizer service that lets us look at how our swarm is scheduling containers.

1. Open up `docker-compose.yml` in an editor and replace its contents with the following. Be sure to replace `username/repo:tag` with your image details.

   ```
   version: "3"
   services:
     web:
       # replace username/repo:tag with your name and image details
       image: username/repo:tag
       deploy:
         replicas: 5
         restart_policy:
           condition: on-failure
         resources:
           limits:
             cpus: "0.1"
             memory: 50M
       ports:
         - "80:80"
       networks:
         - webnet
     visualizer:
       image: dockersamples/visualizer:stable
       ports:
         - "8080:8080"
       volumes:
         - "/var/run/docker.sock:/var/run/docker.sock"
       deploy:
         placement:
           constraints: [node.role == manager]
       networks:
         - webnet
   networks:
     webnet:
   ```

   The only thing new here is the peer service to `web`, named `visualizer`. Notice two new things here: a `volumes` key, giving the visualizer access to the host's socket file for Docker, and a `placement` key, ensuring that this service only ever runs on a swarm manager – never a worker. That's because this container, built from an open source project created by Docker, displays Docker services running on a swarm in a diagram.

   We talk more about placement constraints and volumes in a moment.

2. Make sure your shell is configured to talk to `myvm1` (full examples are here).

   - Run `docker-machine ls` to list machines and make sure you are connected to `myvm1`, as indicated by an asterisk next to it.

- If needed, re-run `docker-machine env myvm1`, then run the given command to configure the shell.

On **Mac or Linux** the command is:

`shell eval $(docker-machine env myvm1)`

On **Windows** the command is:

`shell & "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression`

3. Re-run the `docker stack deploy` command on the manager, and whatever services need updating are updated:

```
$ docker stack deploy -c docker-compose.yml getstartedlab
Updating service getstartedlab_web (id: angi1bf5e4to03qu9f93trnxm)
Creating service getstartedlab_visualizer (id: l9mnwkeq2jiononb5ihz9u7a4)
```

4. Take a look at the visualizer.

You saw in the Compose file that `visualizer` runs on port 8080. Get the IP address of one of your nodes by running `docker-machine ls`. Go to either IP address at port 8080 and you can see the visualizer running:

The single copy of `visualizer` is running on the manager as you expect, and the 5 instances of `web` are spread out across the swarm. You can corroborate this visualization by running `docker stack ps <stack>`:

`docker stack ps getstartedlab`

The visualizer is a standalone service that can run in any app that includes it in the stack. It doesn't depend on anything else. Now let's create a service that *does* have a dependency: the Redis service that provides a visitor counter.

Persist the data

Let's go through the same workflow once more to add a Redis database for storing app data.

1. Save this new `docker-compose.yml` file, which finally adds a Redis service. Be sure to replace `username/repo:tag` with your image details.

```yaml
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    ports:
      - "80:80"
    networks:
      - webnet
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
```

Figure 8: Visualizer screenshot

```
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
  redis:
    image: redis
    ports:
      - "6379:6379"
    volumes:
      - "/home/docker/data:/data"
    deploy:
      placement:
        constraints: [node.role == manager]
    command: redis-server --appendonly yes
    networks:
      - webnet
networks:
  webnet:
```

Redis has an official image in the Docker library and has been granted the short `image` name of just `redis`, so no `username/repo` notation here. The Redis port, 6379, has been pre-configured by Redis to be exposed from the container to the host, and here in our Compose file we expose it from the host to the world, so you can actually enter the IP for any of your nodes into Redis Desktop Manager and manage this Redis instance, if you so choose.

Most importantly, there are a couple of things in the `redis` specification that make data persist between deployments of this stack:

- `redis` always runs on the manager, so it's always using the same filesystem.
- `redis` accesses an arbitrary directory in the host's file system as `/data` inside the container, which is where Redis stores data.

Together, this is creating a "source of truth" in your host's physical filesystem for the Redis data. Without this, Redis would store its data in `/data` inside the container's filesystem, which would get wiped out if that container were ever redeployed.

This source of truth has two components:

- The placement constraint you put on the Redis service, ensuring that it always uses the same host.
- The volume you created that lets the container access `./data` (on the host) as `/data` (inside the Redis container). While containers come and go, the files stored on `./data` on the specified host persists, enabling continuity.

You are ready to deploy your new Redis-using stack.

2. Create a `./data` directory on the manager:

   ```
   docker-machine ssh myvm1 "mkdir ./data"
   ```

3. Make sure your shell is configured to talk to `myvm1` (full examples are here).

   - Run `docker-machine ls` to list machines and make sure you are connected to `myvm1`, as indicated by an asterisk next to it.

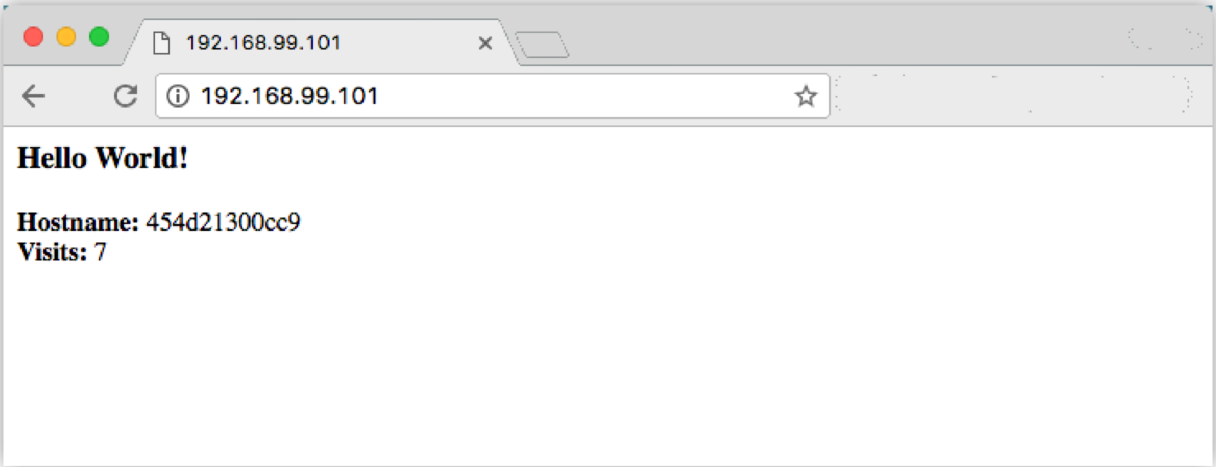

Figure 9: Hello World in browser with Redis

- If needed, re-run `docker-machine env myvm1`, then run the given command to configure the shell.

On **Mac or Linux** the command is:

```
shell   eval $(docker-machine env myvm1)
```

On **Windows** the command is:

```
shell   & "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env
myvm1 | Invoke-Expression
```

4. Run `docker stack deploy` one more time.

```
$ docker stack deploy -c docker-compose.yml getstartedlab
```

5. Run `docker service ls` to verify that the three services are running as expected.

```
$ docker service ls
ID                  NAME                       MODE                REPLICAS            IMAGE
x7uij6xb4foj        getstartedlab_redis        replicated          1/1                 redis:latest
n5rvhm52ykq7        getstartedlab_visualizer   replicated          1/1                 dockersample
mifd433bti1d        getstartedlab_web          replicated          5/5                 gordon/getst
```

6. Check the web page at one of your nodes, such as `http://192.168.99.101`, and take a look at the results of the visitor counter, which is now live and storing information on Redis.

Also, check the visualizer at port 8080 on either node's IP address, and notice see the `redis` service running along with the `web` and `visualizer` services.

Recap (optional)

Here's a terminal recording of what was covered on this page:

You learned that stacks are inter-related services all running in concert, and that – surprise! – you've been using stacks since part three of this tutorial. You learned that to add more services to your stack, you insert them in your Compose file. Finally, you learned that by using a combination of placement constraints and volumes you can create a permanent home for persisting data, so that your app's data survives when the container is torn down and redeployed.

Figure 10: Visualizer with redis screenshot

Part VI

Introduction

You've been editing the same Compose file for this entire tutorial. Well, we have good news. That Compose file works just as well in production as it does on your machine. In this section, we will go through some options for running your Dockerized application.

Choose an option

Install Docker Engine — Community

Find the install instructions for Docker Engine — Community on the platform of your choice.

Create your swarm

Run `docker swarm init` to create a swarm on the node.

Deploy your app

Run `docker stack deploy -c docker-compose.yml getstartedlab` to deploy the app on the cloud hosted swarm.

```
docker stack deploy -c docker-compose.yml getstartedlab

Creating network getstartedlab_webnet
Creating service getstartedlab_web
Creating service getstartedlab_visualizer
Creating service getstartedlab_redis
```

Your app is now running on your cloud provider.

Run some swarm commands to verify the deployment

You can use the swarm command line, as you've done already, to browse and manage the swarm. Here are some examples that should look familiar by now:

- Use `docker node ls` to list the nodes in your swarm.

```
[getstartedlab] ~ $ docker node ls
ID                            HOSTNAME                                      STATUS              AVAILAB
n2bsny0r2b8fey6013kwnom3m *   ip-172-31-20-217.us-west-1.compute.internal   Ready               Active
```

- Use `docker service ls` to list services.

```
[getstartedlab] ~/sandbox/getstart $ docker service ls
ID                  NAME                         MODE                REPLICAS            IMAGE
ioipby1vcxzm        getstartedlab_redis          replicated          0/1                 redis:latest
u5cxv7ppv5o0        getstartedlab_visualizer     replicated          0/1                 dockersamples/vi
vy7n2piyqrtr        getstartedlab_web            replicated          5/5                 sam/getstarted:pa
```

- Use `docker service ps <service>` to view tasks for a service.

```
[getstartedlab] ~/sandbox/getstart $ docker service ps vy7n2piyqrtr
ID                  NAME                  IMAGE                           NODE
qrcd4a9lvjel        getstartedlab_web.1   sam/getstarted:part6   ip-172-31-20-217.us-west-1.compute.int
sknya8t4m51u        getstartedlab_web.2   sam/getstarted:part6   ip-172-31-20-217.us-west-1.compute.int
ia730lfnrslg        getstartedlab_web.3   sam/getstarted:part6   ip-172-31-20-217.us-west-1.compute.int
1edaa97h9u4k        getstartedlab_web.4   sam/getstarted:part6   ip-172-31-20-217.us-west-1.compute.int
uh64ez6ahuew        getstartedlab_web.5   sam/getstarted:part6   ip-172-31-20-217.us-west-1.compute.int
```

Open ports to services on cloud provider machines

At this point, your app is deployed as a swarm on your cloud provider servers, as evidenced by the `docker` commands you just ran. But, you still need to open ports on your cloud servers in order to:

- if using many nodes, allow communication between the `redis` service and `web` service
- allow inbound traffic to the `web` service on any worker nodes so that Hello World and Visualizer are accessible from a web browser.
- allow inbound SSH traffic on the server that is running the `manager` (this may be already set on your cloud provider)

{: id="table-of-ports"}

These are the ports you need to expose for each service:

Service	Type	Protocol	Port
`web`	HTTP	TCP	80
`visualizer`	HTTP	TCP	8080
`redis`	TCP	TCP	6379

Methods for doing this vary depending on your cloud provider.

We use Amazon Web Services (AWS) as an example.

> What about the redis service to persist data?
>
> To get the `redis` service working, you need to `ssh` into the cloud server where the `manager` is running, and make a `data/` directory in `/home/docker/` before you run `docker stack deploy`. Another option is to change the data path in the `docker-stack.yml` to a pre-existing path on the `manager` server. This example does not include this step, so the `redis` service is not up in the example output.

Iteration and cleanup

From here you can do everything you learned about in previous parts of the tutorial.

- Scale the app by changing the `docker-compose.yml` file and redeploy on-the-fly with the `docker stack deploy` command.
- Change the app behavior by editing code, then rebuild, and push the new image. (To do this, follow the same steps you took earlier to build the app and publish the image).
- You can tear down the stack with `docker stack rm`. For example:

```
docker stack rm getstartedlab
```

Unlike the scenario where you were running the swarm on local Docker machine VMs, your swarm and any apps deployed on it continue to run on cloud servers regardless of whether you shut down your local host.

Customers of Docker Enterprise Edition run a stable, commercially-supported version of Docker Engine, and as an add-on they get our first-class management software, Docker Datacenter. You can manage every aspect of your application through the interface using Universal Control Plane, run a private image registry with Docker Trusted Registry, integrate with your LDAP provider, sign production images with Docker Content Trust, and many other features.

Once you're all set up and Docker Enterprise is running, you can deploy your Compose file from directly within the UI{: onclick="ga('send', 'event', 'Get Started Referral', 'Enterprise', 'Deploy app in UI');"}.

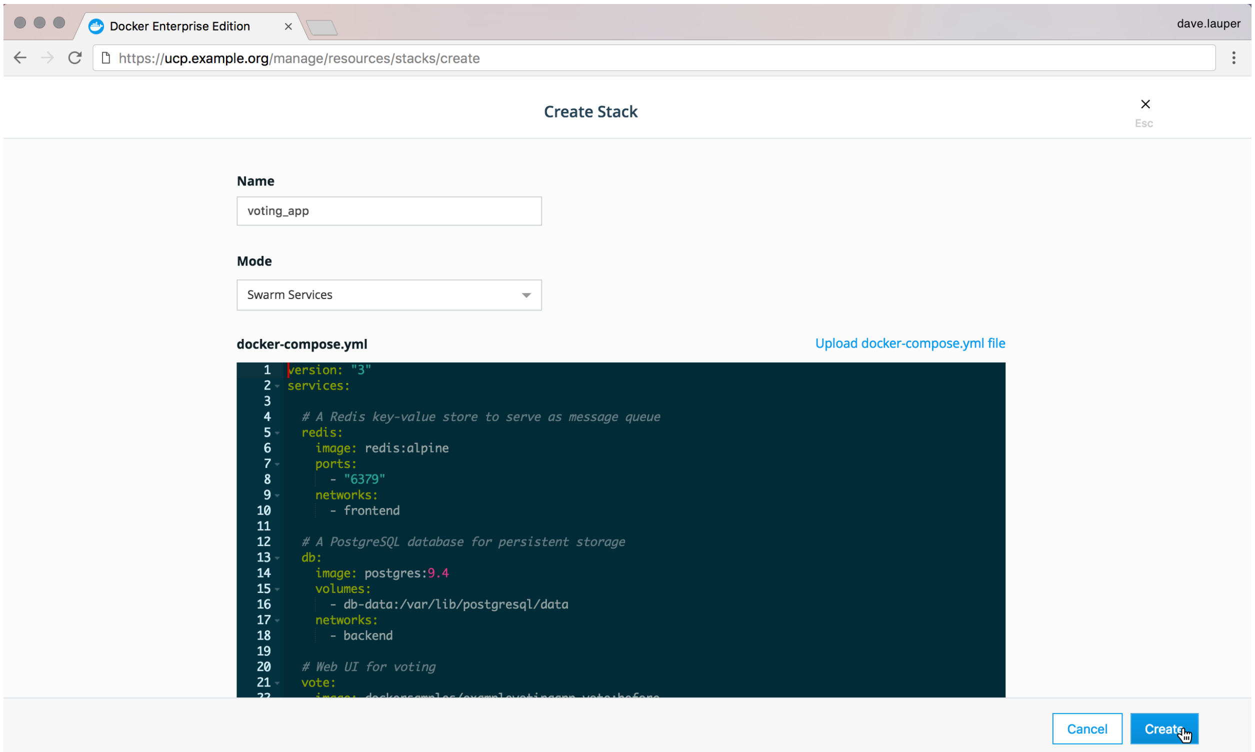

Figure 11: Deploy an app on Docker Enterprise

After that, you can see it running, and can change any aspect of the application you choose, or even edit the Compose file itself.

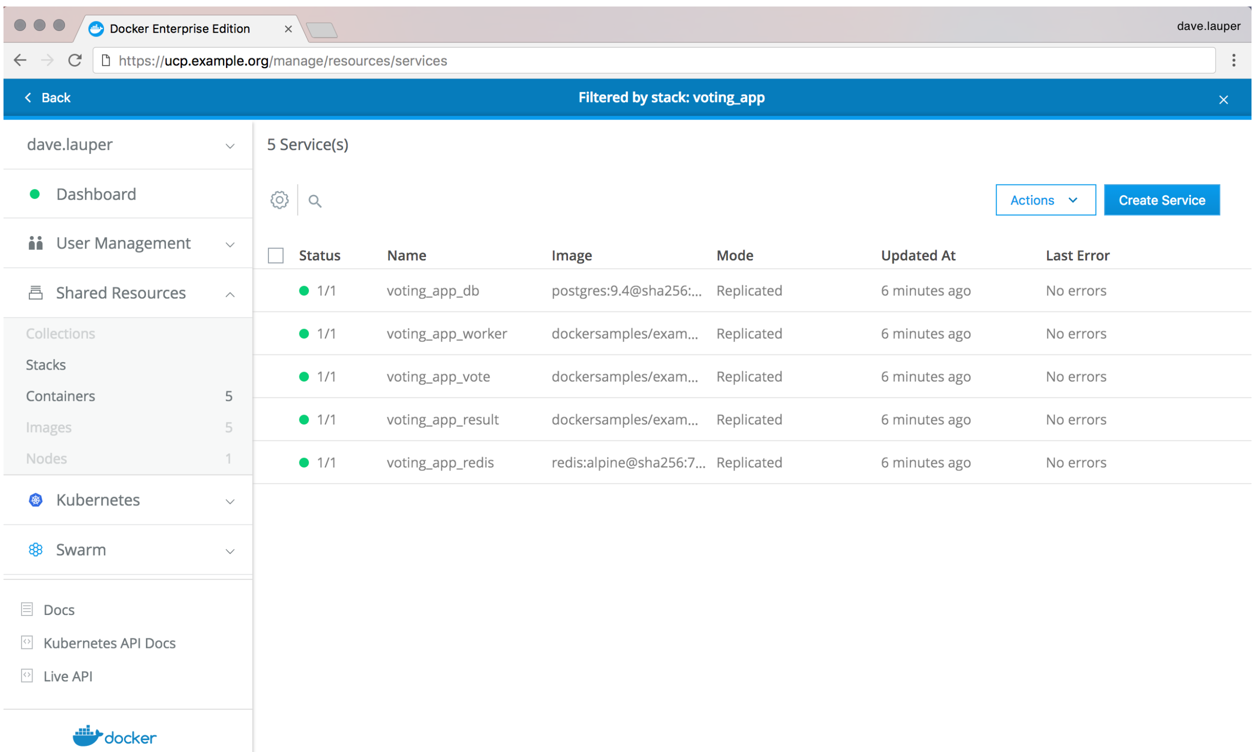

Figure 12: Managing app on Docker Enterprise