# Get Started, Part 5: Stacks

*Estimated reading time: 10 minutes*

1: Orientation (https://docs.docker.com/get-started/part1)

2: Containers (https://docs.docker.com/get-started/part2)

3: Services (https://docs.docker.com/get-started/part3)

4: Swarms (https://docs.docker.com/get-started/part4)

5: Stacks (https://docs.docker.com/get-started/part5)

6: Deploy your app (https://docs.docker.com/get-started/part6)

## Prerequisites

- Install Docker version 1.13 or higher (https://docs.docker.com/engine/installation/).
- Get Docker Compose (https://docs.docker.com/compose/overview/) as described in Part 3 prerequisites (https://docs.docker.com/get-started/part3/#prerequisites).
- Get Docker Machine (https://docs.docker.com/machine/overview/) as described in Part 4 prerequisites (https://docs.docker.com/get-started/part4/#prerequisites).
- Read the orientation in Part 1 (https://docs.docker.com/get-started/).
- Learn how to create containers in Part 2 (https://docs.docker.com/get-started/part2/).
- Make sure you have published the `friendlyhello` image you created by pushing it to a registry (https://docs.docker.com/get-started/part2/#share-your-image). We use that shared image here.
- Be sure your image works as a deployed container. Run this command, slotting in your info for `username` , `repo` , and `tag` : `docker run -p 80:80 username/repo:tag` , then visit `http://localhost/` .
- Have a copy of your `docker-compose.yml` from Part 3 (https://docs.docker.com/get-started/part3/) handy.
- Make sure that the machines you set up in part 4 (https://docs.docker.com/get-started/part4/) are running and ready. Run `docker-machine ls` to verify this. If the machines are stopped, run `docker-machine start myvm1` to boot the manager, followed by `docker-machine start myvm2` to boot the worker.

- Have the swarm you created in part 4 (https://docs.docker.com/get-started/part4/) running and ready. Run `docker-machine ssh myvm1 "docker node ls"` to verify this. If the swarm is up, both nodes report a `ready` status. If not, reinitialize the swarm and join the worker as described in Set up your swarm (https://docs.docker.com/get-started/part4/#set-up-your-swarm).

# Introduction

In part 4 (https://docs.docker.com/get-started/part4/), 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 (https://github.com/ManoMarks/docker-swarm-visualizer), 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 (https://docs.docker.com/get-started/part4/#configure-a-docker-machine-shell-to-the-swarm-manager)).

   - 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:

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

On **Windows** the command is:

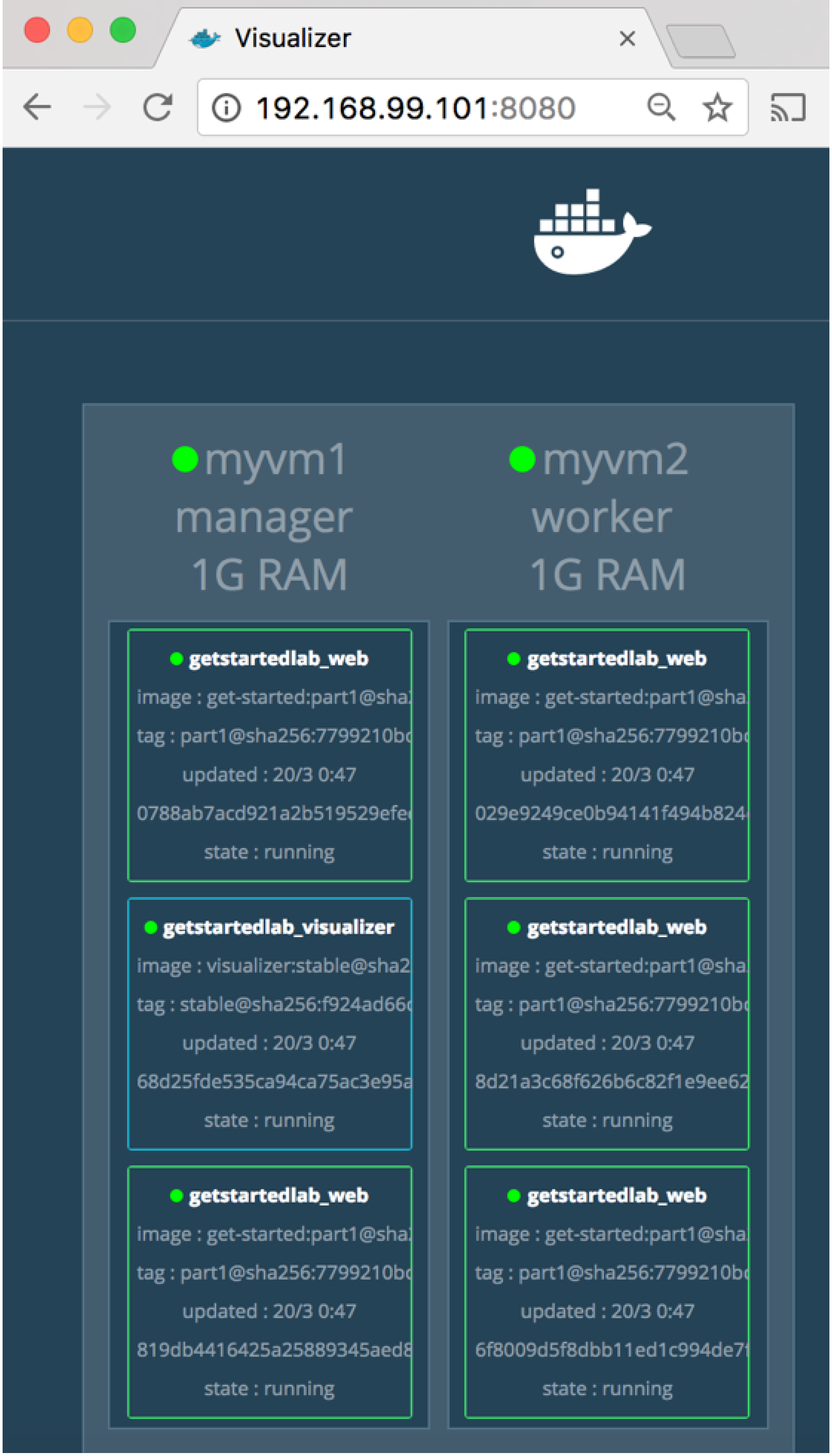
```
& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" en
```

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:



myvm1
manager
1G RAM

● getstartedlab_web
image : get-started:part1@sha2
tag : part1@sha256:7799210bc
updated : 20/3 0:47
0788ab7acd921a2b519529efe
state : running

● getstartedlab_visualizer
image : visualizer:stable@sha2
tag : stable@sha256:f924ad66c
updated : 20/3 0:47
68d25fde535ca94ca75ac3e95a
state : running

● getstartedlab_web
image : get-started:part1@sha2
tag : part1@sha256:7799210bc
updated : 20/3 0:47
819db4416425a25889345aed8
state : running

myvm2
worker
1G RAM

● getstartedlab_web
image : get-started:part1@sha2
tag : part1@sha256:7799210bc
updated : 20/3 0:47
029e9249ce0b94141f494b824
state : running

● getstartedlab_web
image : get-started:part1@sha2
tag : part1@sha256:7799210bc
updated : 20/3 0:47
8d21a3c68f626b6c82f1e9ee62
state : running

● getstartedlab_web
image : get-started:part1@sha2
tag : part1@sha256:7799210bc
updated : 20/3 0:47
6f8009d5f8dbb11ed1c994de7f
state : 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.

```
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
  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 (https://docs.docker.com/get-started/part4/#configure-a-docker-machine-shell-to-the-swarm-manager)).

   - 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:

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

     On **Windows** the command is:

     ```
     & "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" en
     ```

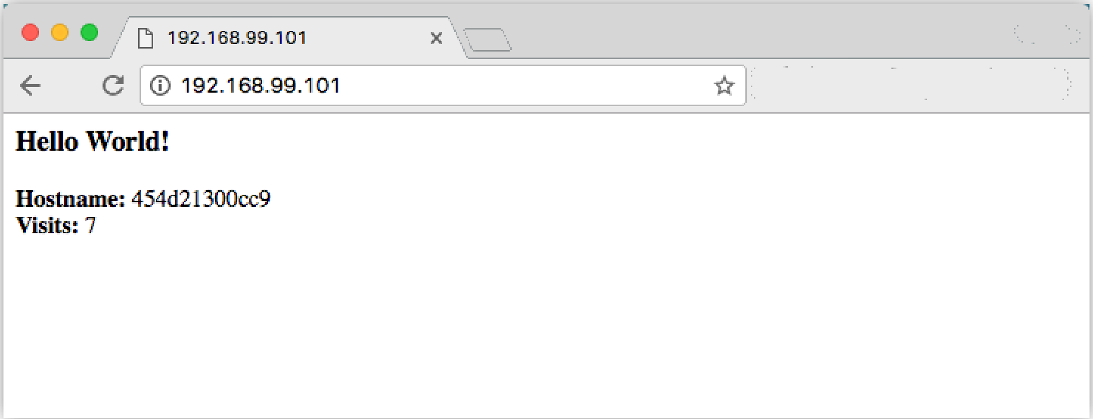
4. Run `docker stack deploy` one more time.

```
$ 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
x7uij6xb4foj    getstartedlab_redis         replicated    1/1
n5rvhm52ykq7    getstartedlab_visualizer    replicated    1/1
mifd433bti1d    getstartedlab_web           replicated    5/5
```

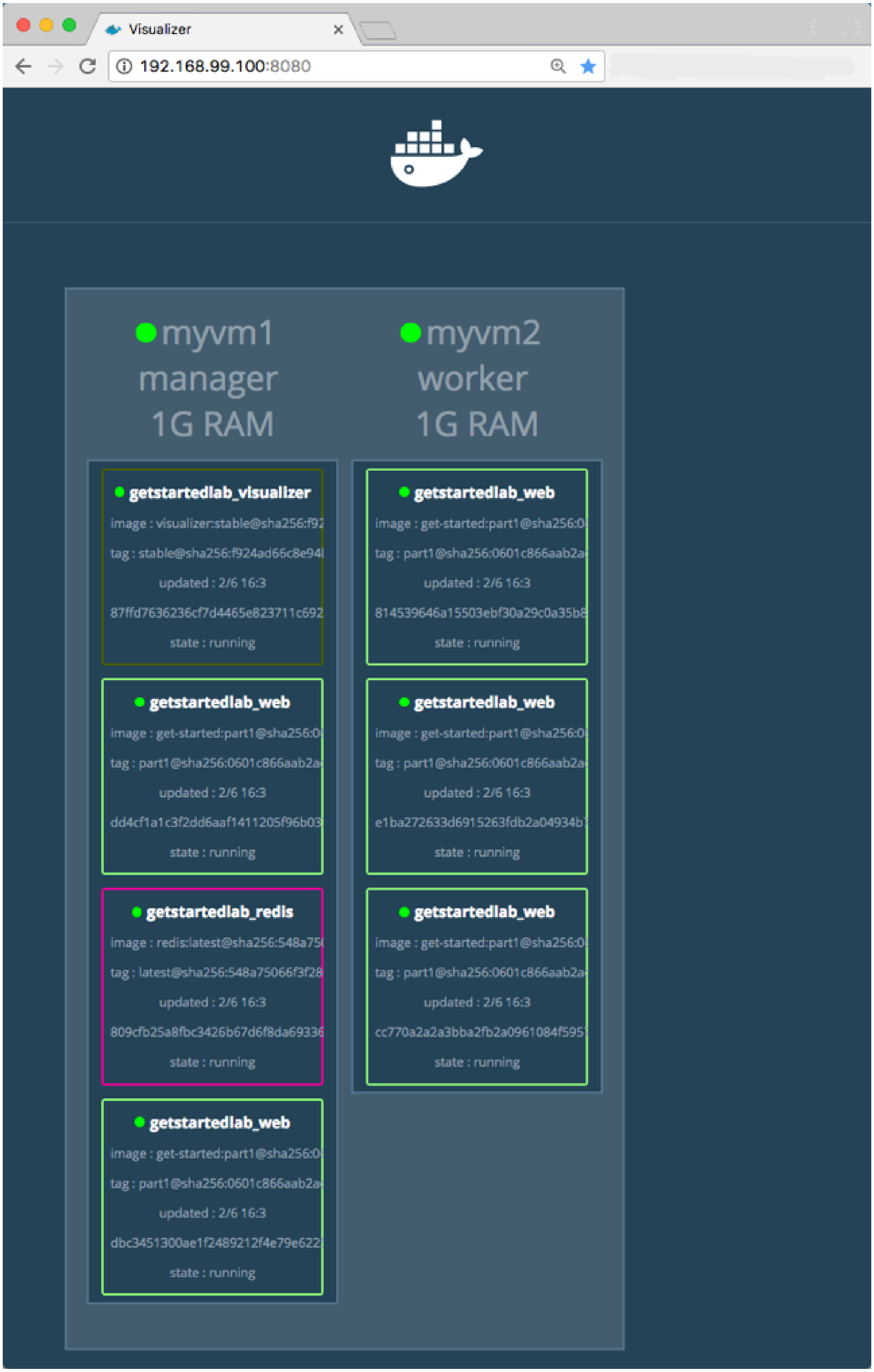
6. Check the web page at one of your nodes, such as `http://192.168.99.101`, and take a look at the results of the visitor counter, which is now live and storing information on Redis.



**Hello World!**

**Hostname:** 454d21300cc9
**Visits:** 7

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



● myvm1
manager
1G RAM

● getstartedlab_visualizer
image : visualizer:stable@sha256:f92
tag : stable@sha256:f924ad66c8e94l
updated : 2/6 16:3
87ffd7636236cf7d4465e823711c692
state : running

● getstartedlab_web
image : get-started:part1@sha256:0
tag : part1@sha256:0601c866aab2a
updated : 2/6 16:3
dd4cf1a1c3f2dd6aaf1411205f96b03
state : running

● getstartedlab_redis
image : redis:latest@sha256:548a75
tag : latest@sha256:548a75066f3f28
updated : 2/6 16:3
809cfb25a8fbc3426b67d6f8da69336
state : running

● getstartedlab_web
image : get-started:part1@sha256:0
tag : part1@sha256:0601c866aab2a
updated : 2/6 16:3
dbc3451300ae1f2489212f4e79e622
state : running

● myvm2
worker
1G RAM

● getstartedlab_web
image : get-started:part1@sha256:0
tag : part1@sha256:0601c866aab2a
updated : 2/6 16:3
814539646a15503ebf30a29c0a35b8
state : running

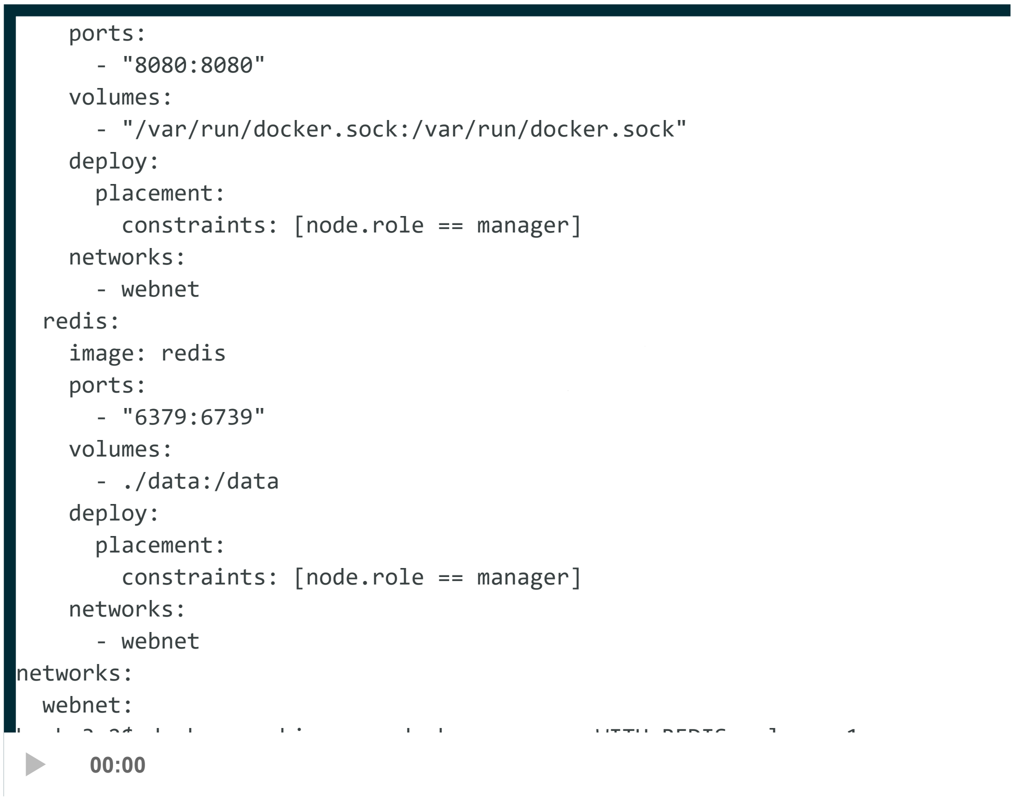
● getstartedlab_web
image : get-started:part1@sha256:0
tag : part1@sha256:0601c866aab2a
updated : 2/6 16:3
e1ba272633d6915263fdb2a04934b
state : running

● getstartedlab_web
image : get-started:part1@sha256:0
tag : part1@sha256:0601c866aab2a
updated : 2/6 16:3
cc770a2a2a3bba2fb2a0961084f595
state : running

On to Part 6 >> (https://docs.docker.com/get-started/part6/)

# Recap (optional)

Here’s a terminal recording of what was covered on this page (https://asciinema.org/a/113840):



```
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
  redis:
    image: redis
    ports:
      - "6379:6739"
    volumes:
      - ./data:/data
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
networks:
  webnet:
```

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.

stack (https://docs.docker.com/glossary/?term=stack), data (https://docs.docker.com/glossary/?term=data), persist (https://docs.docker.com/glossary/?term=persist), dependencies (https://docs.docker.com/glossary/?term=dependencies), redis (https://docs.docker.com/glossary/?term=redis), storage (https://docs.docker.com/glossary/?term=storage), volume (https://docs.docker.com/glossary/?term=volume), port (https://docs.docker.com/glossary/?term=port)