Docker in Production | PACKT Books

In this article by **Scott Gallagher**, the author of Mastering Docker, we will be looking at Docker in production, pulling all the pieces together so you can start using Docker in your production environments and feel comfortable doing so. Let's take a peek at what we will be covering in this article:

- Setting up hosts and nodes
- Managing hosts and container
- Using Docker Compose
- Extending to external platforms
- Security

(For more resources related to this topic, see here.*)*

When we start thinking about putting Docker into our production environment, we first need to know where to start. This sometimes can be the hardest part of any project. We first need to start by setting up our Docker hosts and then start running containers on them. So, let's start here!

Setting up hosts

Setting up hosts will require us to tap into our Docker Machine knowledge. We can deploy these hosts to different environments, including cloud hosting. To take a job down memory lane, let's look at how we go about doing this:

```
$ docker-machine create –driver
```

Now, there are two values that we can manipulate: and . The host name can be whatever you want it to be. But I recommend that it should be something that would help you understand its purpose. The driver name on the other hand has to be the location where you want to create the host. If you are looking at doing something locally, then you can use virtualbox or vmwarefusion. If you are looking at deploying to a cloud service, you can use something like Amazon EC2, Azure, or DigitalOcean. Most of these cloud services will require additional details to authenticate who you are and where to place the host:

For example, for AWS, you would use:

```
$ docker-machine create --driver amazonec2
 --amazonec2-access-key  --amazonec2-secret-key
 --amazonec2-subnet-id east-1b amazonhost
```

You can see that you will need the following:

- Amazon access key
- Amazon secret key
- Amazon subnet ID

Setting up nodes

Next, we want to set up the nodes or containers to run on the hosts that we have recently created. Again, using a combination of Docker Machine with the Docker daemon, we can do this. We first must use Docker Machine to point to the Docker host that we want to deploy some containers on:

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

Now we can run our normal Docker commands against this Docker host. To do this, we will simply use the Docker command-line tools. To deploy the containers, we can pull the following images:

```
$ docker pull
```

Or, we can run a container on a host:

```
$ docker run -d -p 80:80 nginx
```

In this section, we will focus on host management, that is, the ways we can manage our hosts, what we should use to manage our hosts, how we can monitor our hosts, and container failover, which is very important when something happens to the host that is running critical containers.

Host monitoring

With host monitoring you can do so via the command line using Docker Machine as also there are some GUI applications out there that can be useful as well. For Machine, you can use the *ls* subcommand:

```
$ docker-machine ls
NAME           ACTIVE   DRIVER       STATE     URL
            SWARM
amazonhost               amazonec2    Error
swarm-master    *        virtualbox   Running
tcp://192.168.99.102:2376   swarm-master(master)
swarm-node1              virtualbox   Running
tcp://192.168.99.103:2376   swarm-master
```

You can use some GUI applications out there as well, such as:

Docker Swarm

Another tool that you can use for node management is that of Docker Swarm. We saw previously how helpful Swarm can be. Remember that you can use Docker Swarm to manage your hosts as well as to create and list them. The most useful commands to remember for Swarm is the *list* subcommand. With the *list* subcommand, you can get a view of all the nodes and their statuses:

Remember that you will need either the discovery service IP or the token number that is used for Swarm:

```
$ docker run swarm list token://
```

Swarm Manager failover

With Docker Swarm, you can set up your manager node to be highly available. That is, if the manage host dies, you can have it failover to another host. If you don’t have it set up, there will be a service interruption, as you won’t be able to communicate to your hosts anymore and will need to reset them up to point to the new Docker Swarm manager. You can set up as many replicas as you want.

To set this up, you will need to use the *—replication* and *—advertise* flags. This

tells Swarm that there will be other managers for failover. It will also tell Swarm what address to advertise on, so the other managers know on what IP address to connect for other Swarm managers.

Now, let's look at container management. This includes questions such as where to store the images that we will be creating, how to use these images, and what commands and GUI applications we can use. It also covers how we can easily monitor our running containers, automatically restart containers upon a failure, and how to roll the updates of our containers.

Container image storage

Remember that there are three major locations to store the images you are creating:

- **Docker Hub**: A location that is run by Docker and can contain public and private repositories
- **Docker Trusted Registry**: A location that is again run by Docker, but provides the ability to get support from Docker
- **The locally run Docker registry**: Locally run by yourself to storage images

You will want to consider where you want your images to be stored. If you are running containers that might contain data that you do not want anybody to be able to access, such as private code, you may want to run your own Docker registry to keep the data locked. If you are testing, then you may only want to use Docker Hub. If you are in an enterprise environment where uptime is necessary, then the second option of having Docker there for support would be immensely beneficial. Again, it all depends on your setup and needs. The best thing is that no matter what you choose at first, you can easily change and push your images to these locations without having to jump through a lot of extra hoops or other configurations.

Image usage

The most important thing to remember about Docker images is the four Ws:

- **Who**: Who made the image?
- **What**: What is contained in the image?

- **Why**: Why are these things created?
- **Where**: Where are the items such as the Dockerfile or the other code for the image?

The Docker commands and GUIs

Remember that there are many commands that you can use to control your containers. With tools such as the Docker daemon, Docker Machine, Docker Compose, and Docker Swarm, there is almost nothing that can stop you from achieving the goal you want. Remember, however, that some of these tools are not available on all the platforms yet. I stress yet as I assume that Docker will eventually have their tools available for all the environments. Be sure to use the *—help* flag on all the commands to see the additional switches that might be available. I myself am always finding new switches to use every day on various commands.

There are also many GUI applications out there; they can be beneficial to your container's management needs. One that has been on the forefront of this since the beginning is Panamax. Panamax provides the ability to set up your environments in a GUI-based application for you to deploy, monitor, and manipulate your container environments. With the popularity of Docker growing each day, there will be many, many, many others that you can use to help set up and tune your environment.

Container monitoring

We can also monitor our containers using methods similar to monitoring hosts: using Docker commands as well as GUIs that are built by others.

First, the Docker commands that you can use:

- *docker stats*
- *docker port*
- *docker logs*
- *docker inspect*
- *docker events*

In the *Host monitoring* section, you can see that the same GUI applications can

monitor both your Docker hosts and your containers. It is a double bonus as you don't need separate applications to monitor each service.

Automatic restarts

Another great thing you can do with your Docker images is you can set them to automatically restart upon a failure or a reboot of a Docker host. There is a flag that can be set at runtime: the *—restart* flag. There are three options you can set, one of which is set by default by not setting the flag.

These three options are:

- *no*: The default by not using the flag.
- *on-failure:max_retires*: Sets the container to restart, but not indefinitely if there is a major problem. It will try to restart the container a number of times based on the value set for *max_retires*. After it has passed that value, it will not try to restart anymore.
- *always*: Will always restart the container. It could cause a looping issue if the container continues to just restart.

Rolling updates

One of the benefits I have learned to love about Docker is the ability to control it the same way I control the code that I write. Just like Git, remember that your Docker images are version-controlled as well. This being said, you can do things such a rolling updates to them. There are two ways you can go about doing it. You can keep your images as a hosted code on something like GitHub. You can then update your code, build your image, and deploy your containers. If something goes wrong, you can simply use another version of that image to redeploy. There is also another way you can do this. You can get the new image up and running; when you are ready, stop the old container from running and then start up the new one. If you use items such as discovery services, it becomes even easier; you can roll your newly updated images into the discovery service while rolling out the old images. This makes for seamless upgrades and a great peace of mind for zero downtime.

One of the more useful tools, and one I find myself using a lot, is Docker Compose. Compose has a lot of powerful usage, which in turn is great for you. In this section, we will look at two of its usages:

- Developer environments
- Scaling environments

Developer environments

You can use Docker Compose to set up your development environments. How is this any different from setting up a virtual machine for them to use or letting them use their own setup? With Docker Compose, you control the setup, you control what is linked to what, and you know how the environment is set up. So, there is no more “well it works on my system” or need to troubleshoot error messages that are appearing on one system setup but not another.

Scaling environments

Docker Compose also allows you to scale containers that are located in the *docker-compose.yml* file. For example, let’s say our Compose file looks as follows:

```
varnish:
  image: jacksoncage/varnish
  ports:
    - "82:80"
  links:
    - web
environment:
    VARNISH_BACKEND_PORT: 80
    VARNISH_BACKEND_IP: web
    VARNISH_PORT: 80
web:
   image: scottpgallagher/php5-mysql-apache2
   volumes:
     - .:/var/www/html/
```

With the Compose setup, you can easily scale the containers from your *docker-compose.yml* file. For instance, if you need more web containers to help with the backend load, you can do so with Docker Compose. Be sure that you are in the folder where your *docker-compose.yml* file is located:

```
$ docker-compose scale web=3
```

This will add three extra web containers and do all the linking as well as the traffic forwarding from the varnish server that is necessary. This can be immensely helpful if you are looking at figuring out how many instances you might need to help scale for load or service usage.

We looked at how we can extend to some other external platforms such as cloud services like AWS, Microsoft Azure, and DigitalOcean before. In this section, we will focus on extending Docker to the Heroku platform. Heroku is more a little different than those cloud services; it is a considered a **Platform as a Service** (**PaaS**). Instead of deploying containers to it, you can link your containers to the Heroku platform from which it is running a service, such as PHP, Java, Node.js, Python, or many others. So, you can run your rails application on Heroku and then attach your Docker container to that platform.

Heroku

The way you can use Docker and Heroku together is by creating your application on the Heroku platform. Then, in your code, you will have something similar to the following:

```
{
  "name": "Application Name",
  "description": "Application to run code in a Docker
 container",
  "image": ":",
  "addons": [ "heroku-postgresql" ]
}
```

To take a step back, we first need to install a plugin to be able to get this functionality working. To install it, we will simply run:

```
$ heroku plugins:install heroku-docker
```

Now, if you are wondering what image you can or should be using from Docker Hub, Heroku maintains a lot of images you can use in the preceding code. They are as follows:

```
heroku/nodejs
heroku/ruby
heroku/jruby
heroku/python
heroku/scala
heroku/clojure
heroku/gradle
heroku/java
heroku/go
heroku/go-gb
```

Lastly, let's take a look at the security aspect of putting Docker into production. This is probably one of the most talked about aspect of not only Docker, but any technology out there. What security risks exist? What security advantages exist? We will take a look at both of these aspects as well as cover the best practices for your overall Docker setup.

Security best practices

These are the things to keep in mind when you are setting up your production environment:

- Whoever has access to your Docker host has access to every single Docker container that is running on that host and has the ability to stop them, delete them, or even start up new containers.
- Remember that you can run Docker containers or attach containers to Docker volumes using the read-only modes. This can be done by adding the :ro option to the volume:

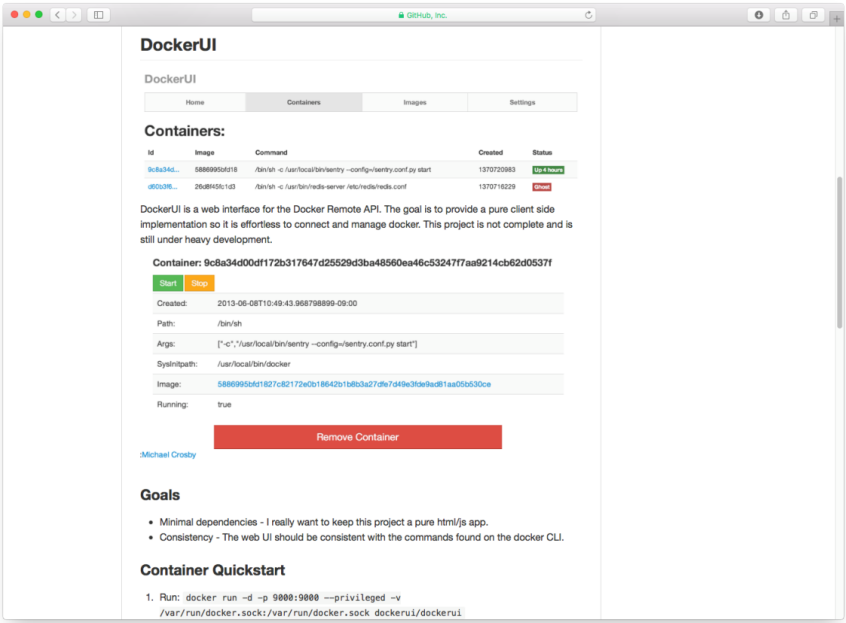
  ```
  $ docker run -d -v /opt/uploads:ro nginx
  $ docker run -d —volumes-from data:ro nginx
  ```

- Remember to utilize the Docker security benchmark application to help tune your environments.
- Utilize the Docker command-line tools to your capability to see what has changed in a particular image:

  ```
  $ docker diff
  ```

```
$ docker inspect
$ docker history
```

DockerUI is a tool written by Michael Crosby, who at the time of writing this book worked for Docker. DockerUI is a simple way to view what is going on inside your Docker host.



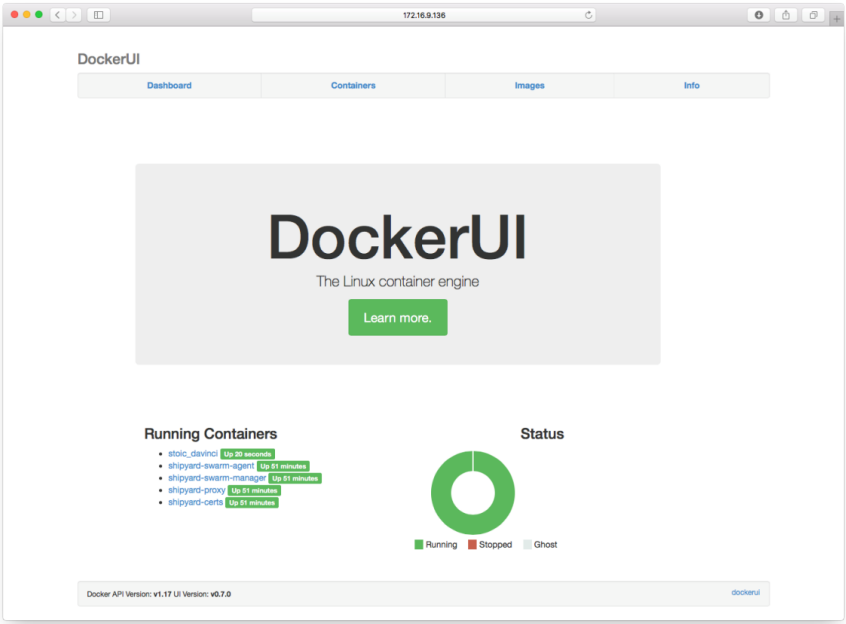
This is a screenshot of the GitHub repository, where the code for DockerUI is kept. You can view the content yourself by navigating to https://github.com/crosbymichael/dockerui.

This page will include screenshots of DockerUI in action as well as the current features of DockerUI that are available. You can create pull requests against the code if you have ideas you would like to see in DockerUI and would like to help contribute to the code. You can also submit issues that you might find with DockerUI.

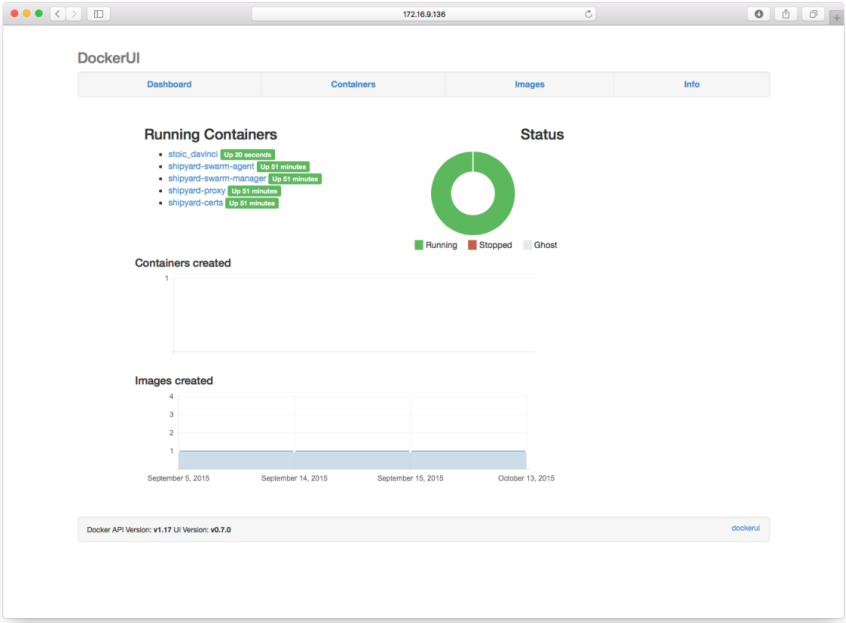
The installation of DockerUI is very straightforward with you just running a

simple Docker run command to get started:

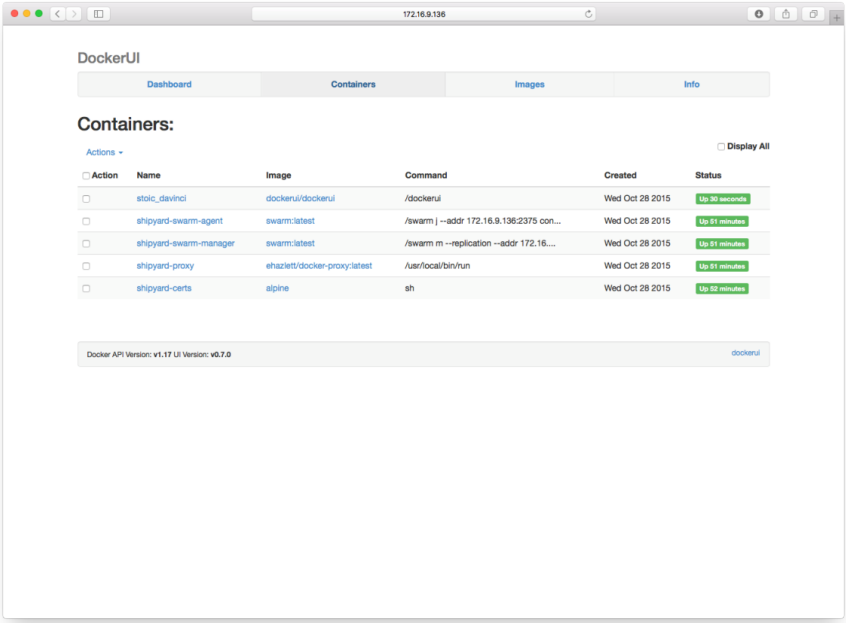
```
$ docker run -d -p 9000:9000 --privileged -v
 /var/run/docker.sock:/var/run/docker.sock dockerui/dockerui
```



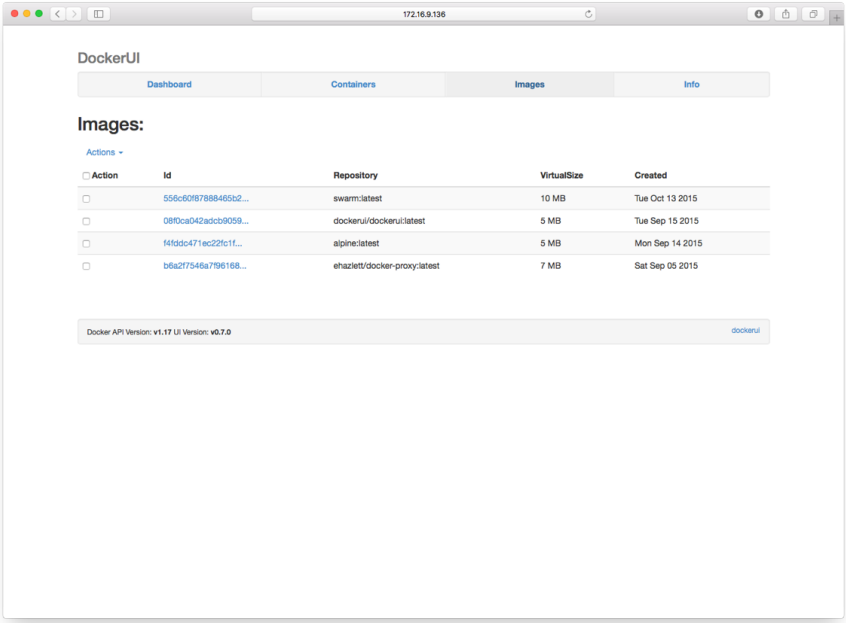
After you have run the previous command, you will be able to navigate to the DockerUI web interface. You should be able to easily breakdown the run command and see what it is doing and where you need to go to get to the dashboard. However, in case you are stumped, here is what the command is doing: it is running the DockerUI container on your Docker host and exposing port 9000 from the host to the container. So, simply launching a web browser and pointing to the IP address of the Docker host and then port 9000 will give you to a screen similar to the previous one. This is the DockerUI web dashboard.



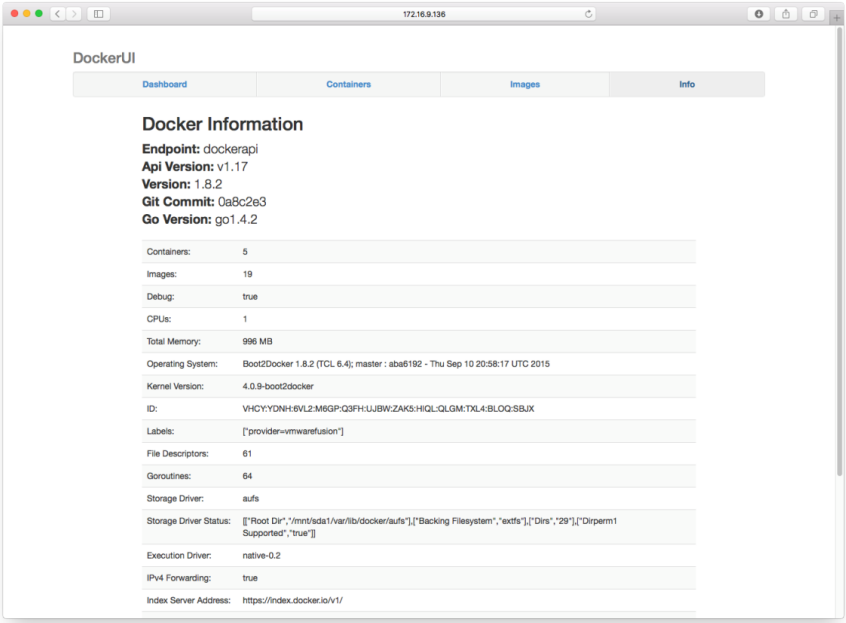
This is another view of the dashboard shortly after you have launched the container and visited the web interface. You can see information such as what containers are currently running on your Docker host and what their statuses are; some could be stopped as well. It will also show you the containers that are created and a timeline for when the images were created.



At the top of the web interface, you will see a navigation bar. When you click on the **Containers** item, you will be brought to a page that provides you information on all the containers running on your host. You will see their name, the images used to run the containers, what command is being executed inside each container, when they were created, and their statuses. You can take actions against these containers from here as well. These actions are start, stop, restart, kill, pause, unpause, and remove.

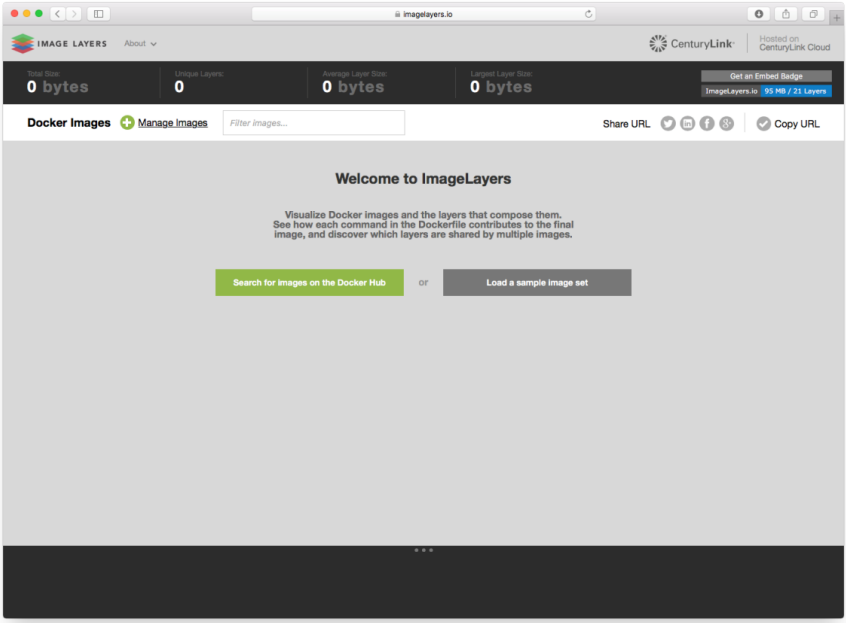


Next up in the navigation bar is **Images**. Again, like **Containers**, you can get all the information on all the images being used on your Docker host here. Information such as their IDs, what repositories they are from, their virtual sizes, and when they were created will be displayed here. Again, you can take some actions on your images. But for images, the only option you have is to remove them from your Docker host.



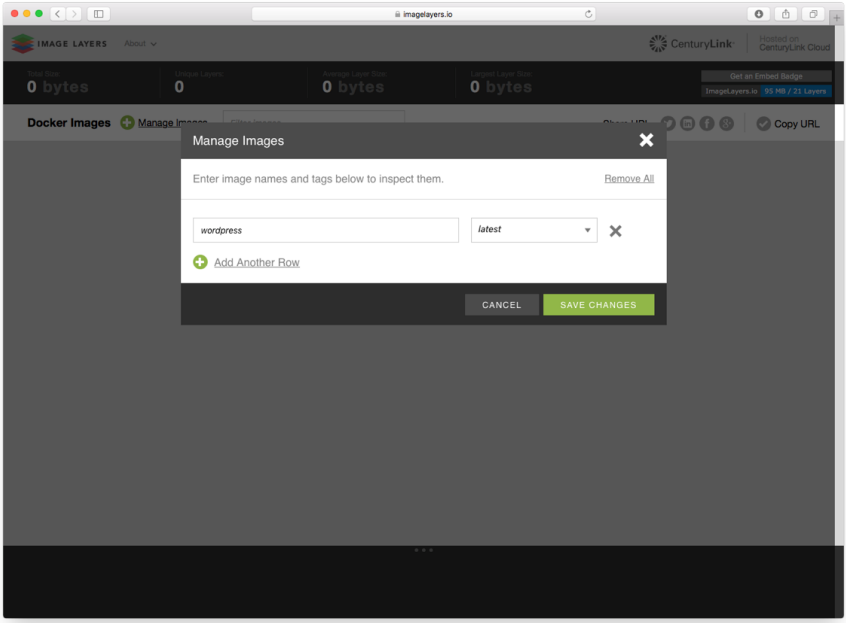
The last item in the navigation menu is **Info**. The **Info** section gives you a general overview of your Docker host, such as what Docker version it is running and how many containers and images are there. It also provides system information on the hardware that is available.

ImageLayers is a great tool, when you are looking at shipping your containers or images around. It will take into account everything that is going on in every single layer of a particular Docker image and give you an output of how much weight it has in terms of actual size or the amount of disk space it will take up.

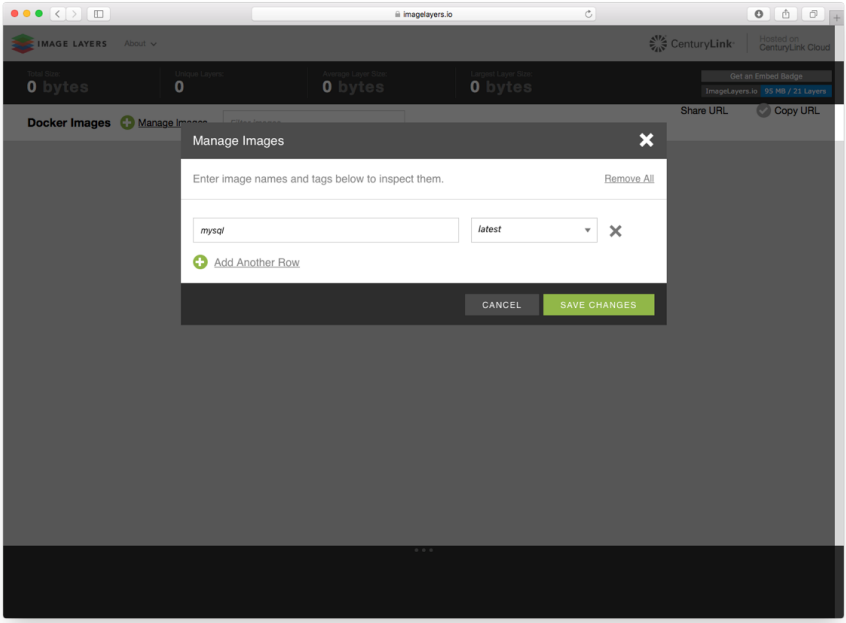


This screenshot is what you will be presented with while navigating to the ImageLayers website: https://imagelayers.io.

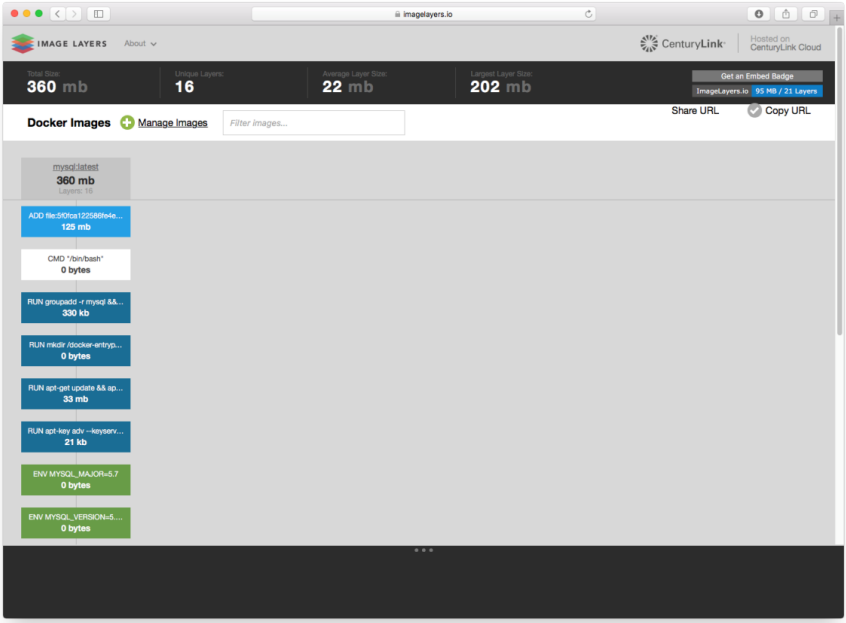
You can search for images that are on Docker Hub to have ImageLayers provide information on the image back to you. Or, you can load up a sample image set if you are looking at providing some sample sets or seeing some more complex setups.



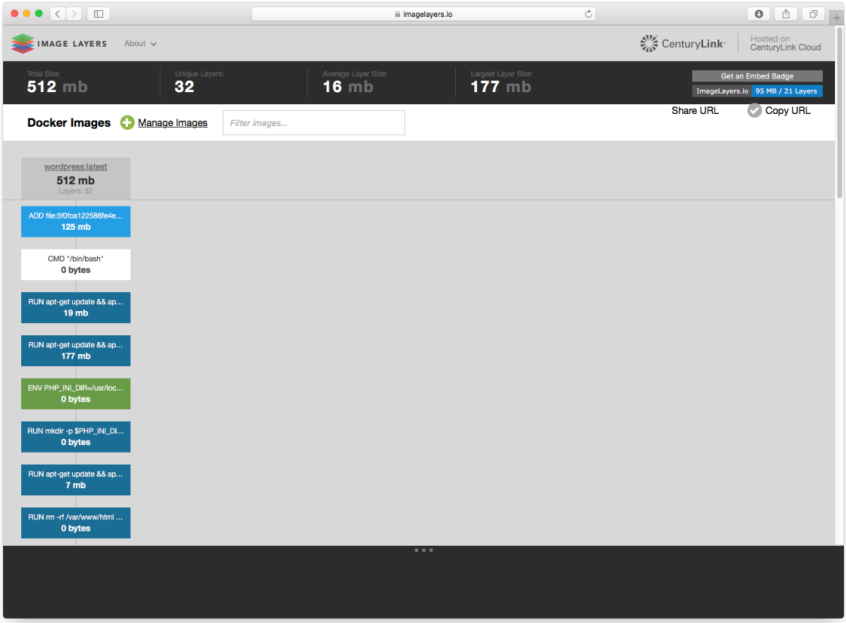
In this example, we are going to search for the *wordpress* image and select the **latest** tag. Now, you can search for any image and it does do autofill and then you can select any tag as well. This could be useful if you have, say, a staging tag and are thinking of pushing a new image to your latest tag, but you want to see what impact it has on the size of the image.



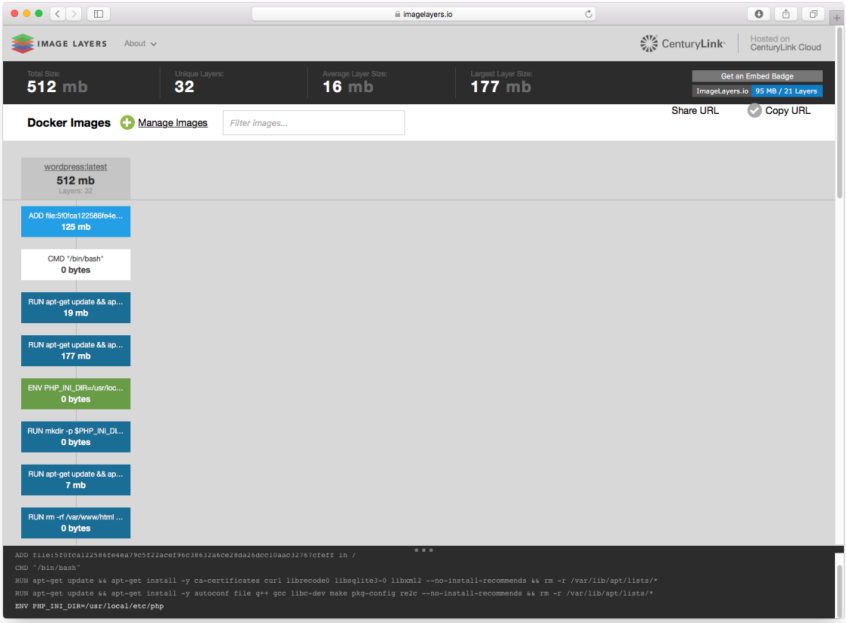
So, let's walk through an example. In this example, we are going to select a *mysql* image and the **latest** tag. We will use this since it is a common image that most people will use at some point in their Docker experience.



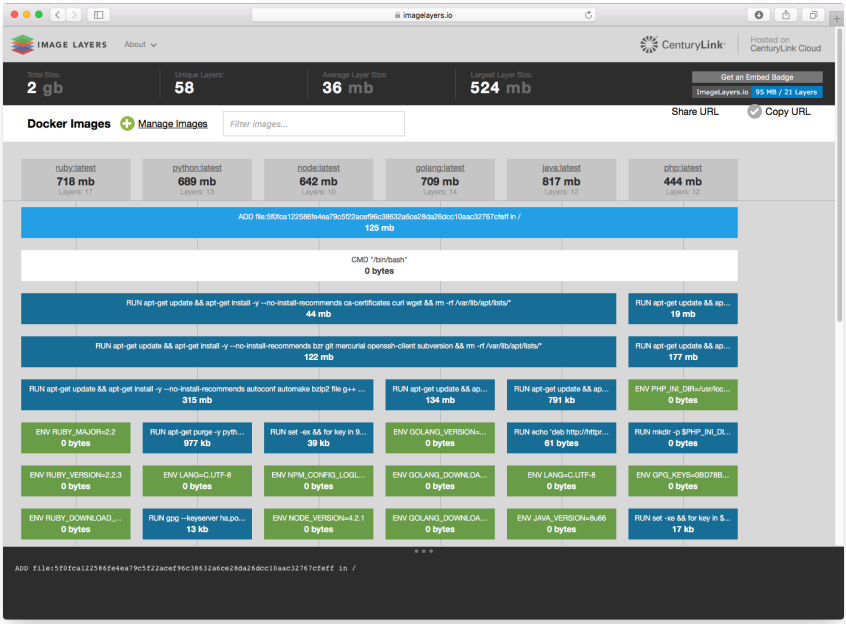
Once we click on **Save Changes** from the previous item, we will be shown something similar to the preceding screenshot (now, this will vary depending upon the image you have selected in your search). This displays some information at the top, such as the total image size, unique layers, the average layer size, and the largest layer size. This will help you hone in on a particular layer that might have grown wildly.



The layers are broken down on the left-hand side of the previous screenshot. We can see what action is being done at each level as the size that it adds to the overall image per layer.



Upon hovering on a particular layer, you will be given information on it at the bottom of the screen in a black box. This will show how each action is layered one after the other so as to help see the command structure of the image.



The preceding screenshot is an example of you might see if you were to click on the sample image set from the main screen. As you can see, this one is quite complex; not only does it have a lot of layers, but it also has a lot of images that are being used. This could be something you would see while adding multiple images to see your desired output.

In this article, you have learned how to use Docker in a production environment as well as the key considerations to keep an eye on during the times of and before implementation.
