



CODESHIP



CHRIS WARD
TECHNICAL WRITER, SPEAKER, DEVELOPER

Understanding the Docker Ecosystem



About the Author.

Chris Chinch is a Developer, Designer and Writer currently living in Berlin, but originally from London and with many years spent in Melbourne.

After 15 years as a developer Chris realized that his skills lie in helping others understand technical subjects. He achieves this through technical writing, editing and journalism, developer relations and educating people through presentations and workshops.



Understanding the Docker Ecosystem.

You might have heard about Docker and have an idea about what it does and why it can be useful to software teams. But which services and components exactly make Docker what it is? And how do they work together?

In this ebook, we'll present a summary of what's currently (June 2016) available as part of the Docker ecosystem, how it can help you, and how the pieces fit together.

On the following pages you will learn about the Docker Hub, Docker Engine, Kitematic, Docker Machine, Docker Swarm, Docker Compose, Docker Cloud, and Data Center. We'll briefly introduce all of these services and look at what they do and how they help software development teams around the world.



Introduction

Docker builds upon concepts of the past but packages them better. Docker is a tool for creating 'containers' that can hold just what you need for a discrete application or technology stack. Unlike Virtual Machines, these containers share the same resources for managing interactions between the containers and the host machine. This makes Docker containers **quick, light, secure and shareable**.

Personally, as a technology writer and presenter, I have found Docker invaluable for creating presentations and demos. I can assemble a stack of components I need, run them and then destroy them again, keeping my system clean and uncluttered with packages and data I no longer need.

Many developers could see a clear use case for Docker during development and testing but struggled to understand how best it could work in production. A flurry of third-party tools and services emerged to help developers deploy, configure, and manage their Docker workflows from development to production.

Docker has been building their own 'official' toolkit through a series of takeovers and product releases. [Project Orca](#), as it is known, was announced at last year's DockerCon US, though details are a little vague.



In my opinion, Orca is more the strategy behind the consolidation of Dockers growing portfolio of products than an actual project or product.

In this article, I'll present a summary of what's currently available as part of the Docker ecosystem, how it can help you, and how the pieces fit together.



Docker Hub

At the heart of any project using Docker is a [Dockerfile](#). This file contains instructions for Docker on how to build an image. Let's look at a simple example:

DOCKERFILE

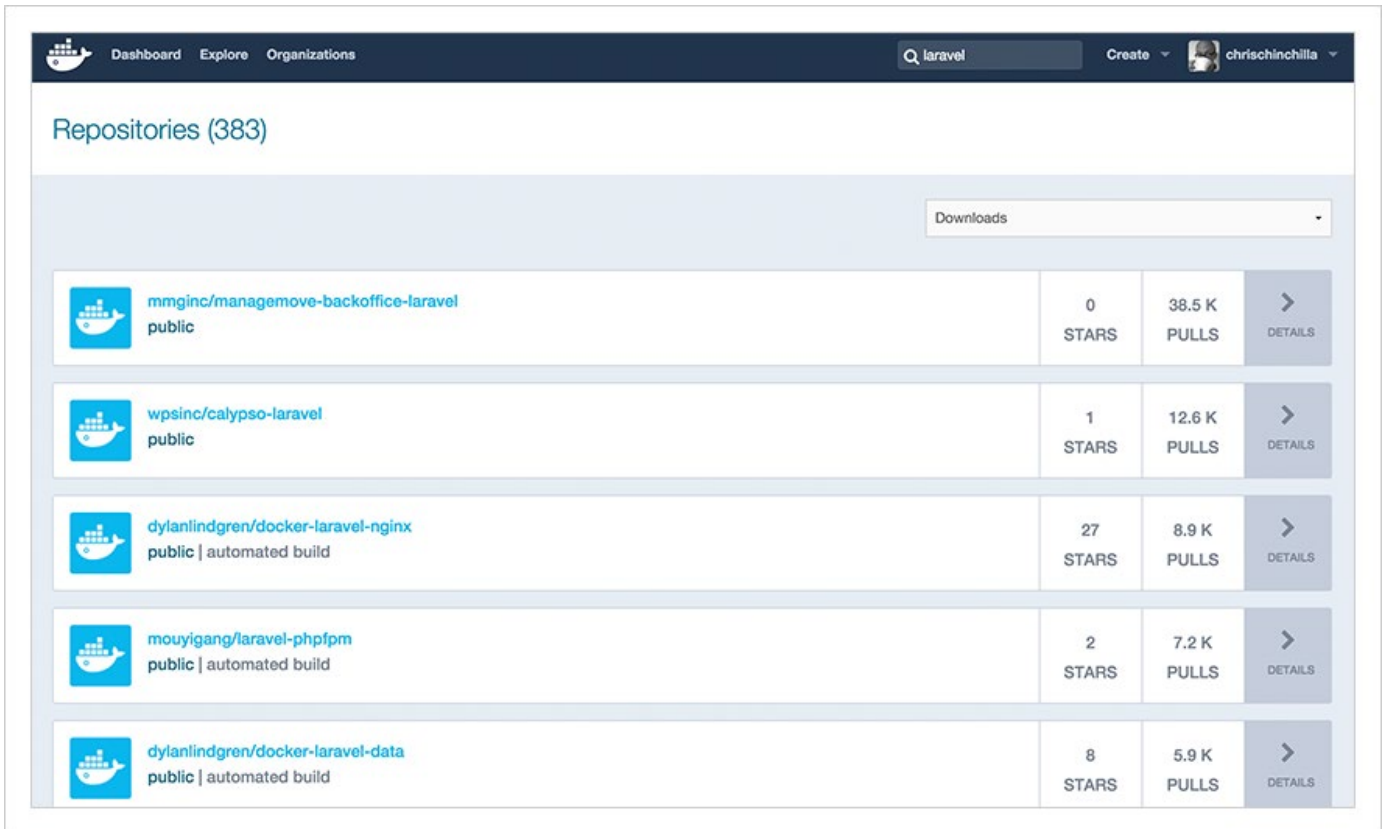
```
FROM python:2.7
ADD . /code
WORKDIR /code
RUN pip install -r requirements.txt
```

In this example, the Dockerfile pulls a particular version of an existing image, copies the current local directory into the file system of the container, sets it as the working directory, and then installs Python dependencies from a text file via the `pip` command.

The [Docker Hub](#) is the official source of pre-written Dockerfiles, providing public (for free) and private (paid)



repositories for images. If you're looking for a Dockerfile to suit your needs, search the Hub first, using project documentation, downloads, and stars to help guide you.



The screenshot shows the Docker Hub interface with a search for 'laravel'. The results list several repositories, including 'mmginc/managemove-backoffice-laravel', 'wpsinc/calypso-laravel', 'dylanlindgren/docker-laravel-nginx', 'mouyigang/laravel-phpfpm', and 'dylanlindgren/docker-laravel-data'. Each entry shows the repository name, its visibility (public), and statistics for stars and pulls.

Repository	Stars	Pulls	Details
mmginc/managemove-backoffice-laravel public	0	38.5 K	> DETAILS
wpsinc/calypso-laravel public	1	12.6 K	> DETAILS
dylanlindgren/docker-laravel-nginx public automated build	27	8.9 K	> DETAILS
mouyigang/laravel-phpfpm public automated build	2	7.2 K	> DETAILS
dylanlindgren/docker-laravel-data public automated build	8	5.9 K	> DETAILS



Docker Engine

The Docker Engine builds Dockerfiles and turns them into usable containers. It is the core of Docker and nothing else will run without it. There are several options for installing the Docker Engine depending on your operating system, which you can [find more details about here](#).



To start a container based upon an image on the Docker Hub, pull its image and run it. Continuing the Python example:

CODE

```
docker pull python
docker run -it --rm --name script-name -v "$PWD":/usr/src/appname -w /usr/src/appname python:3 python app.py
```

This pulls the latest Python image and then starts a container that runs a Python script and exits when complete. There are some other options set, and the **run** command provides many more; you can [read a complete guide here](#).

When a Docker **run** command starts to become more complex, it may be a better idea to create your own custom Dockerfile. To start a container based on a local Dockerfile, run the following inside the directory containing the file:

CODE

```
docker build -t my_image .
```

This will create an image named **my_image**. Start a container based on the image by running:

CODE

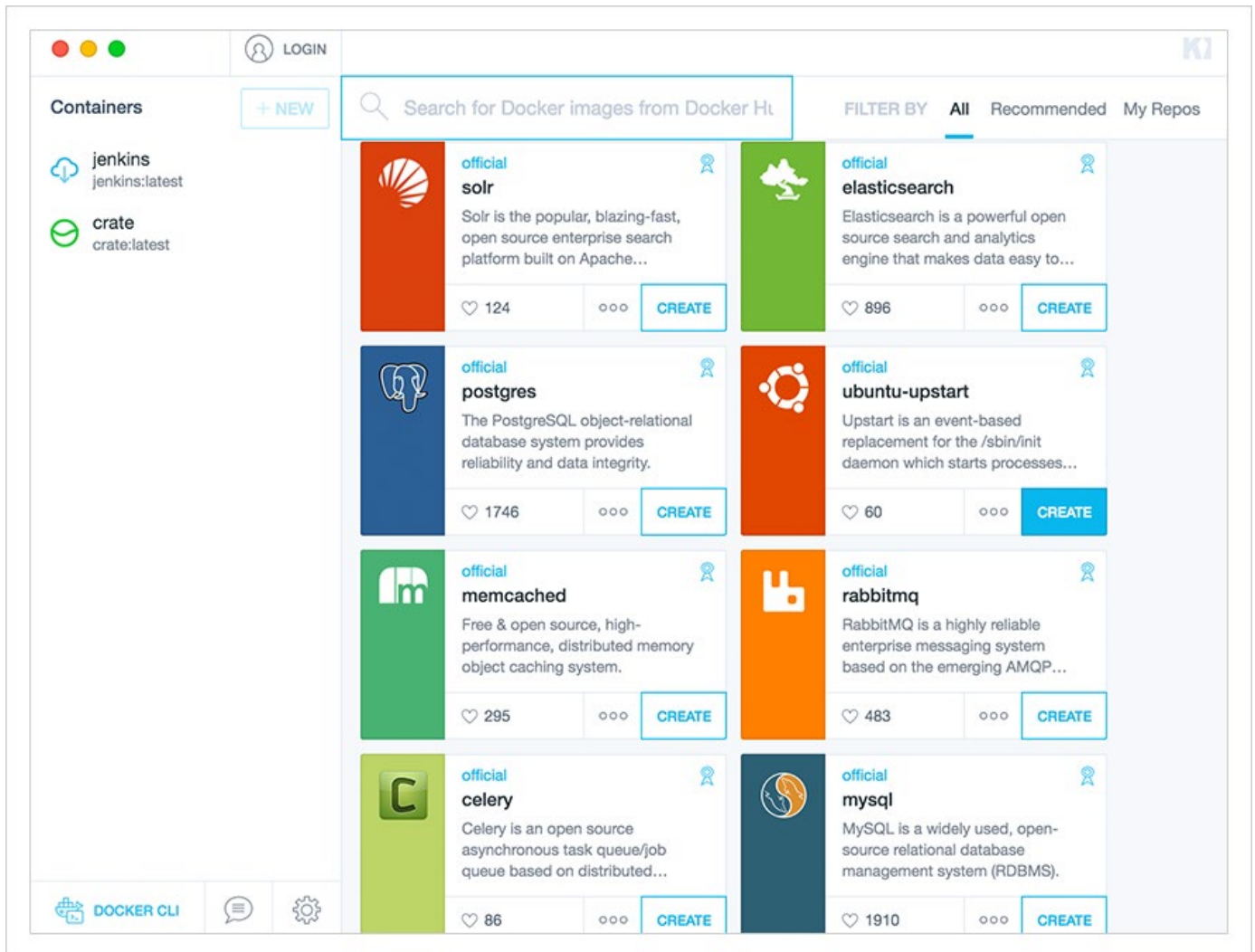
```
docker run --name my_image_container -i -t my_image
```

This starts a container named **my_image_container** based upon your custom **my_image**.



Kitematic

For those of you who would rather avoid the command line, [Kitematic](#) is a great GUI tool for Mac OS X and Windows. Search for the image you need, create a container, and you're good to go. Kitematic offers basic configuration options, but for more advanced settings, you may need to dive into the command line.





Your containers appear on the left hand side where they can be started, stopped, restarted, and most usefully, you can find container logs and direct SSH (the **exec** button) access.

crate

RUNNING

STOP

RESTART

EXEC

DOCS

CONTAINER LOGS

```

-12 15:37:38,609][INFO ][node
[Firelord] version[1.7.5], pid[1], build[${build/NA}
[2016-03-12 15:37:38,610][INFO ][node
[Firelord] initializing ...

```



Docker Machine and Swarm

The first steps toward using Docker in production are understanding [Machine](#) and Swarm, providing a simple set of tools for moving and scaling your local projects to a variety of virtualization and cloud providers.

For example, to create a Docker instance on Azure:

CODE

```
docker-machine create -d azure --azure-subscription-id="XXX"
--azure-subscription-cert="/mycert.pem" ecodemo
```



This command creates an Ubuntu 12.04-based VM named **ecodemo** with Docker preinstalled. Each provider requires different parameters and authentication methods, and default settings can be overridden. Read more details in the [documentation](#) here.

When combined with [Swarm](#), Machine can create clusters of Docker instances that can be treated as one single, large Docker instance. Every Swarm cluster needs a master instance, which is created with the following command:

CODE

```
docker-machine create
  -d virtualbox
  --swarm
  --swarm-master
  --swarm-discovery token://TOKEN_ID
  swarm-master
```

This creates a Docker instance in VirtualBox and sets it as a master node in a Swarm cluster. The **TOKEN_ID** is important as it helps all nodes in a cluster identify each other. Aside from creating a token manually, there are [discovery systems](#) you can use to help manage this process.

Add Docker instances to the Swarm cluster, using the same **TOKEN_ID**:



CODE

```
docker-machine create
  -d virtualbox
  --swarm
  --swarm-discovery token://TOKEN_ID
  swarm-node-n
```

swarm-node-n is a unique name for each node in the cluster.

Now instead of starting containers on individual VMs, you can start containers on the cluster, and the master node will allocate it to the most available and capable node.



Docker Compose

[Compose](#) makes assembling applications consisting of multiple components (and thus containers) simpler; you can declare all of them in a single configuration file started with one command.

On the following page is an example of a Compose file (named **docker-compose.yml**) that creates three instances of the [Crate](#) database and an instance of the PHP framework [Laravel](#) (with some extra configuration). Crucially, the containers are linked with the **links** configuration option.



```
crate1:
  image: crate
  ports:
    - "4200:4200"
    - "4300:4300"
crate2:
  image: crate
crate3:
  image: crate
  volumes:
    - ./data:/importdata
laravelcomposer:
  image: dylanlindgren/docker-laravel-composer
  volumes:
    - /laravel-application:/var/www
  command: --working-dir=/var/www install
  links:
    - crate1
laravelartisan:
  image: dylanlindgren/docker-laravel-artisan
  links:
    - crate1
  volumes_from:
    - laravelcomposer
  working_dir: /var/www
  command: serve --host=0.0.0.0:8080
  ports:
    - "8080:8080"
    - "8080:8080"
```

All these instances and their configuration can now be started by running the following command in the same directory as the **docker-compose.yml** file:

```
docker-compose up
```



```
bash-3.2$ docker-compose up -d
Creating githubhistory_crate3_1...
Creating githubhistory_crate1_1...
Creating githubhistory_laravelcomposer_1...
Creating githubhistory_laravelartisan_1...
Creating githubhistory_crate2_1...
bash-3.2$ docker ps
```

CONTAINER ID	IMAGE NAMES	COMMAND	CREATED	STATUS	PORTS
cf4a8e803594	crate githubhistory_crate2_1	"crate"	4 seconds ago	Up 3 seconds	4200/tcp, 4300/tcp
c21843f626f9	dylanlindgren/docker-laravel-artisan githubhistory_laravelartisan_1	"php artisan serve --"	4 seconds ago	Up 3 seconds	0.0.0.0:8080->8080/tcp, 0.0.
a88e01446b36	dylanlindgren/docker-laravel-composer githubhistory_laravelcomposer_1	"composer --working-d"	4 seconds ago	Up 3 seconds	
dc8bdc6a63bb	crate githubhistory_crate1_1	"crate"	4 seconds ago	Up 3 seconds	0.0.0.0:4200->4200/tcp, 0.0.
0aa59668de12	crate githubhistory_crate3_1	"crate"	4 seconds ago	Up 4 seconds	4200/tcp, 4300/tcp

```
bash-3.2$
```

You can use similar subcommands as the **docker** command to affect all containers started with **docker-compose**. For example, **docker-compose stop** will stop all containers started with **docker-compose**.



Docker Cloud

Automated management and orchestration of containers has been the main piece of the Docker puzzle filled by third-party services until Docker acquired [Tutum \(which underpins Docker Cloud\)](#) last year. While there is no integrated command line tool (at least at the time when this book was written), the Docker Cloud service accepts Docker Compose files to set up application stacks, so it isn't a large diversion from the rest of the ecosystem.

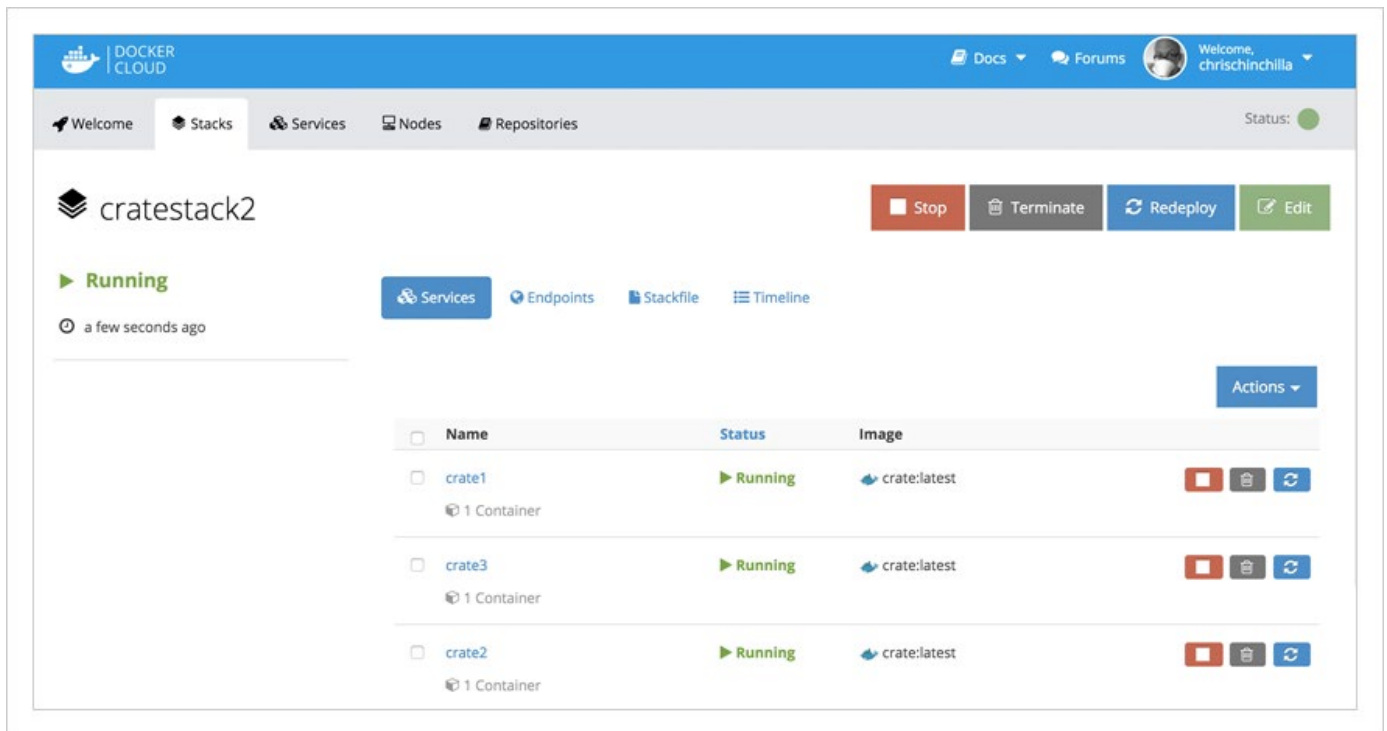
For example:

DOCKER-COMPOSE.YML

```
crate1:
  image: crate
  ports:
    - "4200:4200"
    - "4300:4300"
  command: crate -Des.network.publish_host=_ethwe:ipv4_
crate2:
  image: crate
  command: crate -Des.network.publish_host=_ethwe:ipv4_
crate3:
  image: crate
  command: crate -Des.network.publish_host=_ethwe:ipv4_
```

This creates three instances of the same image, one with port allocation between the host machine and Docker manually set and automatically set on the others. I will revisit **command** soon.

If you want to scale an application beyond one node (which can run as many containers as it can manage) and one private repository, Docker Cloud is a paid service. This is enough for experimentation purposes. Bear in mind that by default Docker Cloud manages containers hosted on third-party hosting services, so you will need to also pay their costs. It's possible to get the Docker Cloud agent running on any Linux host you may manage; you can [find instructions here](#).



The above screenshot shows three Docker containers running across two Digital Ocean instances using a preconfigured rule that allocates containers to hosts based on parameters you set. It will automatically ensure that the quantity of containers you specify are always running.

In the Docker Compose example earlier, you might have noticed `_ethwe:ipv4_`. This is one other great feature of the Docker Cloud. Many distributed applications and services rely on 'Service Discovery' to find other instances of the same service and communicate. When spreading services across data centers and physical machines, this has often required manual declaration of the instances or another way of them finding each other.

Docker Cloud includes support for [Weave](#) to create a 'soft' network across your real network; all containers and applications can discover each other, no matter where they are hosted. In the example above, we override the default command issued to the container to make sure it receives the information it needs to make use of this feature.



Data Center

So far, most of the tools covered in this article have been tools you install, host, and support yourself. For enterprise users looking for a higher guarantee of security, performance, and support, Docker offers [Data Center](#).

It uses much of the same toolkit covered here but adds a private registry for your images, a private cloud, premium support, and third-party integrations with providers likely to appeal to enterprise users. These include user management with LDAP and Active Directory, container monitoring, and logging.



Conclusion

As you will see from my screenshots and your own experiments with these tools, they still feel like a set of connected but loosely coupled products, not a cohesive 'suite'. Project Orca seems to be trying to focus on building consistency between all these projects, making each one a logical stepping stone to the next, all from one GUI or CLI. It aims to not only answer the question "why should I use Docker?" but also "why would I not use Docker?"



Further Reading

DOCKER

- ▶ The Future is Containerized
- ▶ Container Operating Systems Comparison
- ▶ Building a Minimal Docker Container for Ruby Apps

CONTINUOUS DELIVERY

- ▶ Running a MEAN web application in Docker containers on AWS
- ▶ Running a Rails Development Environment in Docker
- ▶ Testing your Rails Application with Docker



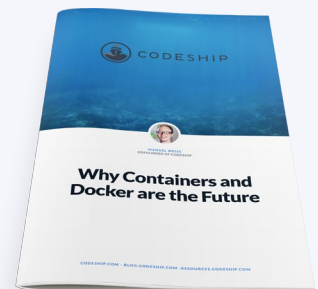
More Codeship Resources.

EBOOK

Why Containers and Docker are the Future.

Learn about the rise of the Container Stack and why Docker and its ecosystem play such a big part in it.

[Download this eBook](#)



EBOOK

Automate your Development Workflow with Docker.

Use Docker to nullify inconsistent environment set ups and the problems that come with them.

[Download this eBook](#)

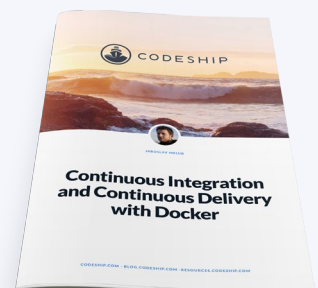


EBOOK

Continuous Integration and Delivery with Docker.

In this eBook you will learn how to set up a Continuous Delivery pipeline with Docker.

[Download this eBook](#)





About Codeship.

Codeship is a hosted Continuous Integration service that fits all your needs.

[Codeship Basic](#) provides pre-installed dependencies and a simple setup UI that let you incorporate CI and CD in only minutes. [Codeship Pro](#) has native Docker support and gives you full control of your CI and CD setup while providing the convenience of a hosted solution.

Codeship Basic

A simple out-of-the-box Continuous Integration service that just works.

Starting at \$0/month.



Works out of the box



Preinstalled CI dependencies



Optimized hosted infrastructure



Quick & simple setup

LEARN MORE

Codeship Pro

A fully customizable hosted Continuous Integration service.

Starting at \$0/month.



Customizability & Full Autonomy



Local CLI tool



Dedicated single-tenant instances



Deploy anywhere

LEARN MORE