JAROSLAV HOLUB

Continuous Integration and Continuous Delivery with Docker

About the Author.

Jaroslav Holub is Senior Engineer at Container Solutions in Amsterdam. He mainly works on research and development of programmable infrastructure projects. He contributes to open source projects and writes about containers, continuous delivery and devops.

Container Solutions is a premium software consultancy that focuses on programmable infrastructure to help their customers innovate at speed and scale.

They provide support and consulting to technical and management teams who are carrying out complex technical projects and transitions as well as training for a range of technologies, including Docker and Mesos.

Learn more about Container Solutions here.

Continuous Integration and Continuous Delivery with Docker.

Continuous delivery is all about reducing risk and delivering value faster by producing reliable software in short iterations. As Martin Fowler says, you actually do continuous delivery if:

- Your software is deployable throughout its lifecycle.
- Your team prioritizes keeping the software deployable over working on new features.
- Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them.
- You can perform push-button deployments of any version of the software to any environment on demand.

ABOUT DOCKER

Docker started as a side project by CTO Solomon Hykes who worked on a Platform as a Service solution called dotCloud. With growth of his PaaS slowing down, Hykes and today's Docker CEO Ben Golub started to build a product around the underlying technology that made dotCloud so fast. This product was Docker.

Containerization of software allows us to further improve on this process. The biggest improvements are in speed and in the level of abstraction used as a cornerstone for further innovations in this field.

In this post, I'll show you how to set up a continuous delivery pipeline using Docker. We'll see how using this tool for Linux containers as part of the continuous delivery pipeline lets us nicely encapsulate the build process of a service. It also lets us deploy any revision with a few simple steps.

We'll mainly use the term continuous delivery here, because it stands for the full circle of steps leading to our ultimate goal. However, continuous integration is the most substantial part of continuous delivery.

Continuous Integration with Docker

Let's take a Hello World web server written in Go as an example service. You can find all the code used in this GitHub repository.

Downloaded the file? Good. Let's look what a continuous integration setup consists of on the next page.

The continuous integration setup consists of:

- running unit tests
- building the Docker image we use to build our service
- running the build container and compiling our service
- building the Docker image that we run and deploy
- pushing the final image to a Docker registry

Automated testing

Running tests in this example is as trivial as it should be:

TEST.SH

```
#!/bin/sh
go test
```

Building Docker image

The core of a single service integration is making the end artifact — Docker image in our case.

Because we've deliberately chosen the compiled language Go in this example, we need to build an executable file as part of our integration process. We'll eventually place the executable file inside this Docker image.

Now one might think that we would build our web server executable file using build tools installed on the host dedicated to continuous integration and then somehow copy the binary to the Docker image. But this is a no-no in the containerized world. Let's do it all in containers. That way, we won't rely on any build tools installed on hosts, and it'll make the whole setup easily reproducible and encapsulated.

Building an executable file can be part of a single Docker image build process together with runtime environment setup. Or we can separate the two. Having everything in a single build process, we would end up with extra content (build process leftovers) in our Docker image filesystem, even if we clean it afterwards in separate `RUN` commands within the Dockerfile.

Some people use tricks to create, manipulate, and remove unwanted stuff in a single `RUN` command. Although it's sometimes handy, I can't generally recommend it; in my opinion this adds to Dockerfile complexity. Of course, there are situations where you might want to retain your sources and all in the end artifact.

The approach I recommend, however, is to create separate "build" and "distribution" Dockerfiles. Use `Dockerfile.build` to do the heavy lifting during building the software, and use `Dockerfile.dist` to create the distributable Docker image, as light and clean as possible.

The following is `Dockerfile.build`. As you can see, once we run the build file, we create the container from a golang image, compile our example service, and output the binary.

DOCKERFILE.BUILD

```
FROM golang:1.4

RUN mkdir -p /tmp/build
ADD hello-world.go /tmp/build/
WORKDIR /tmp/build
RUN go build hello-world.go
CMD tar -czf - hello-world
```

In `Dockerfile.dist`, we only use this binary and run it on runtime:

DOCKERFILE.DIST

```
FROM debian:jessie

RUN mkdir /app
ADD build.tar.gz /app/
ENTRYPOINT /app/hello-world
```

Our `build.sh` script — the essential part of our continuous integration pipeline — then looks like this:

BUILD.SH

```
# !/bin/sh
docker build -t hello-world-build -f Dockerfile.build .
docker run hello-world-build > build.tar.gz
docker build -t hello-world -f Dockerfile.dist .
```

As you can see, these three simple Docker commands get us a clean, small `Hello-World` Docker image that's ready to be deployed and run on demand. Once both images used in the `FROM` clauses are pulled and cached locally, our build process will be a matter of milliseconds or at most a few seconds, with a very small resources footprint.

Storing Docker image

Once our build process artifact is created, we want to push it to Docker Registry, where it will be available for deployments.

Please note that tagging images properly is very important. Docker ecosystems suffer from the usage of "latest" tag. If you use a unique tag for every new image,

then all your image versions will be easily accessible for deployment in the future.

We can choose whether we want to use our own Docker Registry or rely on Docker Hub. On Docker Hub, you can store public or private repositories of images. It's also the first place people would look for your images (if you want anyone to look for them).

Your own Docker Registry on the other hand gives you full control over your images storage, performance, and security. More advanced setups might combine both approaches.

This way you can tag the new image with an appropriate tag and push it to a public hub. Replace `your_username` and `your_tag` with actual values:

PUSH.SH

```
# !/bin/sh
docker tag hello-world:latest your_username/hello-world:your_tag
docker push your_username/hello-world:your_tag
```

Continuously Delivered Containers

Once we have our Docker images building pipeline working and images nicely stashed in a repository, we definitely want to get our service deployed.

How you deploy your applications depends on your infrastructure or cloud provider. A few cloud providers support Docker images in their APIs these days (e.g., Amazon EC2 Container Service, Digital Ocean, or Giant Swarm). You can further leverage the power of containerized applications with resource abstraction tools like Apache Mesos (read more about running containers on Mesos) or Google Kubernetes that let you deploy and manage containers in their own ways.

In case of our `Hello World` example, deploying remotely means running the following command remotely on a target machine with Docker installed on it:

DEPLOY.SH

```
# !/bin/sh
docker stop hello-production
docker run --rm -p 8000:80 --name hello-production hello-world
```

Beyond Continuous Delivery with Docker

Using containerized software does not inherently mean one is implementing microservices. However, containers enable this architectural pattern because they encourage developers to split their monoliths based on separation of concerns.

Microservices also promote communication between containerized components over a plain network using standardized and easily replaceable tubes. To learn more about microservices and why they might be a good architectural pattern for your software project, I recommend "Building Microservices" by Sam Newman.

A continuous delivery pipeline with containerized software also allows you to set up a new kind of testing environment; subsets of (micro)services are deployed in small clusters that represent the system under test running with some parts intentionally disabled or disconnected.

Creation of such a matrix of deployments and programming against it has little to no additional costs in terms of a continuous integration time. It does have a dramatic impact on the stability and resilience of software in production. Such a testing system allows teams to get ready to deal with any kind of Chaos Monkey.

Further Reading

DOCKER

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

CONTINUOUS DELIVERY

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

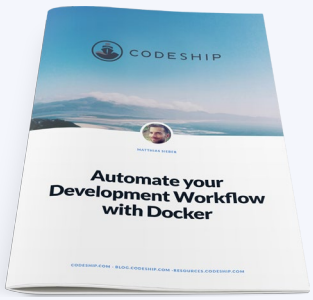
More Codeship Resources.

EBOOK

Automate your Development Workflow with Docker.

Learn how to use Docker to nullify the issue of inconsistent environment setups.

Download this eBook

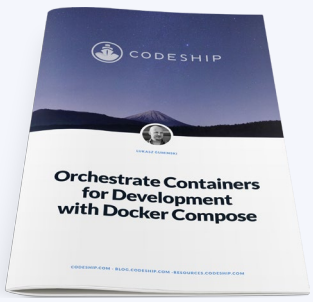


EBOOK

Orchestrate Containers with Docker Compose.

Learn how to use Docker Compose. We focus on how to orchestrate containers in development.

Download this eBook

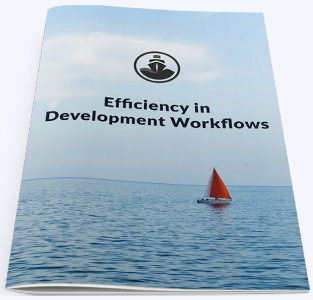


EBOOK

Efficiency in Development Workflows.

Learn about Software Development for distributed teams and how to make them code efficiently.

Download this eBook



CODESHIP

Share this Codeship Guide

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