MATTHIAS SIEBER

Automate your Development Workflow with Docker

CODESHIP

Automate your Development Workflow with Docker.

When you introduce Docker to a team of developers, you can nullify the issue of inconsistent environment setups and the problems that come with them.

But before I show you how you can use Docker for a consistent environment, let me give you a few examples on the next page of why exactly consistent environments are so important.

DID YOU KNOW?

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.

Inconsistent environments come with not-so-small risks

If you were a developer in the pre-Docker world, you probably heard this one quite a few times: **"It worked on my machine!"**

Assuming that this statement is correct, the actual problem then is that a piece of software is apparently not operating the way its developer expected. At least, not on a different machine that's executing the same program.

There are several reasons why this might happen — there might differences in the build process or the environment. While the outcome of a different environment could be so much worse than just an application not running at all, there is still the downside of trying to figure out what's going on and then fixing the issue. It's **time wasted** that could be used for more important things, e.g., developing new features or refactoring code for performance.

But it does get worse than an application simply not running: Imagine a company with $400 million in assets going **bankrupt in 45 minutes** because of a failed deployment. Actually, you don't have to imagine, because you can read about it here. "In 45 minutes, Knight went from being the largest trader in US equities and a major market maker in the NYSE and NASDAQ to bankrupt."

The isolation of the running application also adds a layer of security to the list of benefits for Docker. Docker does this by relying on a technology called "namespaces," which provides the isolated workspace known as the container.

The container of a Docker image contains these components:

- an operating system (e.g., a Debian Linux distribution)
- files added by the user (e.g., application source code)
- configuration (e.g., environment settings and dependencies)
- instructions for what processes to run

The portability of Docker images makes it ideal for use in a continuous integration and deployment process. Any machine that's running Docker can use a Docker image. Setting up a workflow that utilizes Docker is fairly simple, as you'll see in the following example using Ruby.

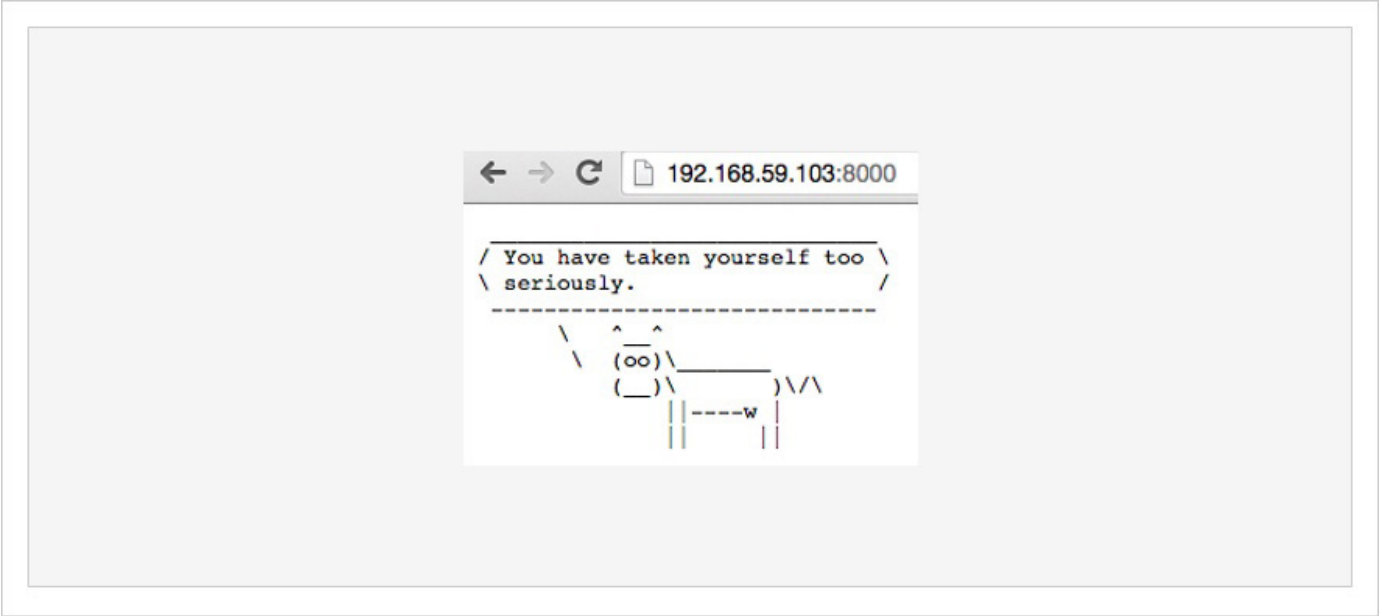
Building an app to run inside a Docker container

Various languages and frameworks incorporate different package management systems. Even when these are being used, two developer machines are usually not the same. One developer might have installed a dependency, not via an application's package or dependency management system, but globally. That dependency may not be on another developer's machine, or perhaps it's not in a more public-facing environment, yet the application needs it. By running an application inside a Docker container, we can avoid these kinds of problems rather easily.

The steps for creating an application, and containerizing it will vary based on the language and framework. In this example we'll create a simple Ruby application, however if you wish to use another language, you should be able to follow this process

Our app will be a simple HTTP server that will utilize two Ruby gems modeled after their command line counterparts, namely cowsay and fortune. A user can access the server via any HTTP request (for example, a browser, postman extension, or curl from the command line). The result is a text response that resembles a cow saying a short somewhat random phrase.

Below is a sample screenshot of the application running inside a Docker container which we'll access via Chrome. The source code is also available on https://github.com/codeship/cowserver.



Building the App

GEMFILE

```
source "https://rubygems.org"

gem "ruby_cowsay"
gem "fortune_gem"
```

APP.RB

```
require "socket"
require "ruby_cowsay"
require "fortune_gem"

server = TCPServer.new('0.0.0.0', 8000)

loop do
  socket = server.accept
  request = socket.gets
  response = Cow.new.say FortuneGem.give_fortune({:max_length => 80})
  socket.print "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n" + "Content-
Length: #{response.bytesize}\r\nConnection: close\r\n\r\n"
  socket.print response
  socket.close
end
```

As you can see, the whole application is composed of just two files. In the Gemfile, we specify which external packages we're going to use in our `app.rb`. The `app.rb` has all the logic of our Ruby application.

Besides the external gems, we're also going to use Ruby's socket library, which provides easy access to the operating system's underlying socket implementations. The reason we're using this library in this example is its simplicity. In just a few lines of code, we get up and running and can accept and respond to HTTP requests.

The response that the server is sending back is a combination of the `ruby_cowsay` and the `fortune_gem` gems.

Creating a Dockerfile

There are several language stacks amongst the official repositories of the Docker Hub registry. Ruby is amongst these repositories, which makes it easy and convenient to set up the environment via Dockerfile. You can also find most of the major languages such as Python, PHP and Golang.

But for this section, I'm going to be creating a Dockerfile from top to bottom. You can make adjustments to your Dockerfile while I go through every step.

Using a language stack as a base image

Using an official repository of a language stack with a version number ensures that the same language features are available on all machines (including development, testing, and production).

We'll be using the **official Ruby Docker image in version 2.2**, let's start by writing this on top of our Dockerfile:

DOCKERFILE

```
FROM ruby:2.2
```

If you're planning on making your application public, you might want to add author information like this:

DOCKERFILE

```
MAINTAINER Matthias Sieber <matthiasksieber@gmail.com>
```

We also need to instruct Docker to listen to the port 8000 at runtime, since that's where our Ruby application will accept connections. This should be the configured port of your application, which may not represent the final routable port of the container.

DOCKERFILE

```
EXPOSE 8000
```

Copying the source code to the Docker image

On the next page, we'll create a directory within the Docker image. We'll copy the files in our project directory there. This changes the working directory to the newly created directory, which now hosts the source code of the application.

DOCKERFILE

```
RUN mkdir -p /usr/src/app
COPY . /usr/src/app
WORKDIR /usr/src/app
```

By copying the source code and installing dependencies, we make sure that the application will run uniformly across all deployed target machines.

Installing dependencies

Since our Ruby application has a Gemfile defined, we can use Bundler to install our dependencies. If you are using another language, just execute your dependency management tool and install any dependencies required. You should also install any other system packages you require. Installing them before any `COPY` commands are executed will speed up image build time, however making them the last step executed ensures your image has the latest version of the dependency.

DOCKERFILE

```
RUN bundle install
```

Ready to launch

So far, we've specified the base image which includes all the tools necessary to build and run a Ruby application. We've also copied the application's source code into the Docker image. All that's left to do is provide a default command that will launch our application. This can be achieved with the CMD instruction. The preferred form is `CMD ["executable", "param"]`. In our case, this is the command we should be running by default:

DOCKERFILE

```
CMD ["ruby","app.rb"]
```

Building the Docker image

There are a couple of ways you can create a Docker image from your source code. In many cases, you probably want to have an automated build process whenever you push your source code to a git repository. This Docker hub repository reflects the state of the GitHub repository where I've pushed the project's code, including the Dockerfile.

But before we consider going that route, we want to make sure that our application is actually working. So let's create a Docker image from the command line.

Creating a Docker image from the command line

To build your project's Docker image, run this command in your project directory:

COMMAND LINE

```
docker build -t <your_user_name>/cowserver .
```

Now you should have a functional Docker image. In order to test it, start the container with this command:

COMMAND LINE

```
docker run -p 8000:8000 <your_user_name>/cowserver
```

The application is now listening on port 8000. Since we're already in the terminal anyway, let's curl that HTTP server:

COMMAND LINE

```
curl http://192.168.59.103:8000/
```

As with all Docker containers running on a Mac OS X, be aware to use the **Docker-specific host IP address**. The default is **192.168.59.103**. Hit the server a couple times to see the ASCII cow with different fortunes.

Running an application in a Docker container isolates the application from the host machine (except where we've explicitly said to share resources), so there's less opportunity for disaster on the host machine.

Conclusion

By creating a Docker image from a simple text file, the problems of inconsistent environments on varying deployment targets are gone. The shaky statement "it worked on my machine" can now be replaced by a reassuring **"it runs in Docker."** The packaged application can be executed and will behave exactly the way it was intended to on all deployment targets.

Resources

STARTER CONTAINERS

- **Python:** https://github.com/ehazlett/docker-py-helloworld
- **Nginx:** https://hub.docker.com/r/tutum/hello-world/
- **Php:** https://github.com/tutumcloud/apache-php
- **Golang:** https://github.com/GoogleCloudPlatform/golang-docker/tree/master/hello

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

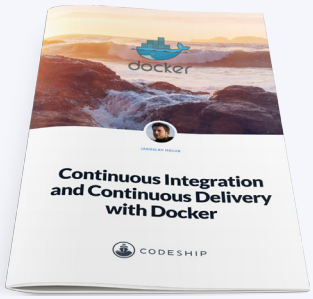
More Codeship Resources.

EBOOK

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

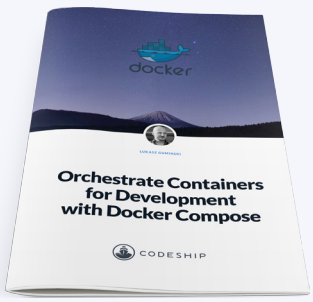


EBOOK

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

About Codeship.

Codeship is a hosted Continuous Integration and Delivery Platform that focuses on speed, security and customizability.

Connect your GitHub and Bitbucket projects and Codeship will test and deploy your apps at every push. Create **organizations**, teams and handle permissions to ensure smooth collaboration and **speed up your testing 10x** with Codeship's ParallelCI pipelines.

You can learn more about Codeship here.

https://codeship.com