ETHAN JONES
SUPPORT ENGINEER AT CODESHIP

Best Practices for Building Minimal Docker Images

About the Author.

Ethan Jones is a support engineer at Codeship. You might know Ethan from Codeship webinars or eBooks. If you're a Codeship user you most probably interacted with Ethan before.

Codeship is a fully customizable hosted Continuous Integration and Delivery platform that helps you build, test, and deploy web applications fast and with confidence.

Learn more about Codeship here.

Best Practices for Building Minimal Docker Images

When architecting Docker applications, keeping your images as lightweight as possible has a lot of practical benefits. It makes things faster, more portable, and less prone to breaks.

Lightweight images also make it easier to use services like Codeship Pro, Codeship's customizable CI platform with Docker support; they're less likely to present complex problems that are hard to troubleshoot, and it takes less time to share them between builds.

I hosted a webinar about the best ways to reduce Docker image sizes. Feel free to watch a re-run here.

ABOUT THIS EBOOK

Keeping your images lightweight has a lot of practical benefits; in this book we will look at some ways to streamline your Docker image as small as possible.

Use as few layers as required

Generally, the fewer the layers, the simpler the Dockerfile. This means you should combine related commands as much as you can, but of course don't try to combine unrelated commands just for the sake of producing a tiny image. Especially if you're new to Docker.

It's better to break up layers when adding files (to increase granularity and cacheability) but to combine layers when running related commands. For instance, run `apt get update && apt-get install` so that a dependent command is always executed with the latest version of the parent command and so that any cleanup from a command is done within the same layer.

There are a few tools to inspect the composition of your Docker images to see which layers might be contributing to bloat. Try looking at your image's layers on MicroBadger.

Clean up right away

When running commands, execute the simplest chain of commands possible to get things working. Plus, as I mentioned earlier, always try to clean up in the same layer where you run your original commands. It's very common to download an archive, extract it, and then

edit the file or move it into place — and then forget to remove the original archive afterward.

Deleting these kinds of files, as well as other temporary logs and cache directories, can reduce a lot of space in your final image. However, if you don't do this within the same Dockerfile line, the deleted file will still exist in a previous image layer. This type of cleanup applies to any interactions on the filesystem during a Docker image build.

For example:

CODE

```
1  RUN wget http://mysite.com/app && tar -xzf app.tar.gz && rm app.tar.gz
```

Instead of:

CODE

```
1  RUN wget http://mysite.com/app
2  RUN tar -xzf app.tar.gz
3  RUN rm app.tar.gz
```

Use a stripped-down base image

Using an appropriate base image for your project can make a huge difference. It makes sense to use the official Ruby image for your Rails project, but if you're just executing a binary in a container, do you need the full

Ubuntu image? Maybe you can use a smaller image, like Alpine, or even just run a Scratch container?

One possibility would be to create a custom base image with only the components you need. Don't be afraid to take stripped down images and add just your required components. Sometimes a smaller image customized to your needs is preferable to an out-of-the-box image.

Use the right image for the right service

Wherever it makes sense, go ahead and share images between services. But if one service needs Ruby but not Rails, while all your other services require Rails, you can probably separate those out to create multiple, contextually streamlined images.

Do use:

- Use an existing similar image or a more complex image
  - when that image is being built anyway;
  - and the extra time to wait for the required image does not slow down the overall build process.
- Use a service with the simplest base image in all other cases.

Do not:

- Use a more complex image as a build artifact.

Optimize dockerignore

This one can make a big difference: Add as much as possible to your `.dockerignore`.

Depending on your application, you probably want to ignore the `.git`, `log`, and `tmp` folders at a minimum (although in some cases `.git` may need to be included). You can also update other parts of your pipeline to avoid dumping large binaries into directories that are not required in the build context. This will help keep your images from ballooning as an unintended result of build artifacts being erroneously added into the build environment.

One way to help figure out what to remove is to run your image while overriding the entrypoint to an interactive shell. This way, you can take a look at all the files that were added and note any that you don't actually need.

CODE

```
$ docker build -t myapp ./
$ docker run --entrypoint /bin/bash -it myapp
root # ls
mycode tmp logs
^D
$ echo "tmp" >> .dockerignore
$ echo "logs" >> .dockerignore
```

You can poke around inside the running container to see if your image left behind any unnecessary files.

One great side effect of this optimization is that your image builds will be faster, since your image layers will have fewer files that could invalidate them.

Build versus bootstrap

A decision that will impact a lot of your other choices and have a big impact on your final results is whether to build assets into an image or to prepare them at runtime instead.

While you could compile assets at build time and include them in your images, you can also generate them at run time instead. You could even pull them from an external source where they were generated either during the build, during a previous build, or from an altogether out-of-band process. You can also add your assets to your image in a compressed format and uncompress them at runtime.

The right path for you depends on your architecture and your goals, but you should factor in how heavy or light you ultimately want your images to be.

Conclusion

As you can see, there are a lot of great ways to reduce your image size. Some of them go beyond mere efficiency and rely on decisions that will permeate the rest of your application architecture.

Just remember: Efficient images are ultimately a tradeoff of size balanced against what you need to reliably and easily support your application.

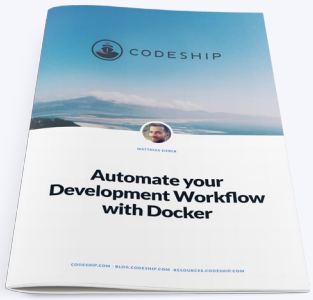
More Codeship Resources.

EBOOKS

Automate your Development Workflow with Docker.

In this eBook we will learn how to use Docker to solve the problems of inconsistent environments on varying deployment targets.

Download this eBook



EBOOKS

Breaking up your Monolith into Microservices.

In this eBook you will learn about the basics of "decomposing" a monolith into microservices.

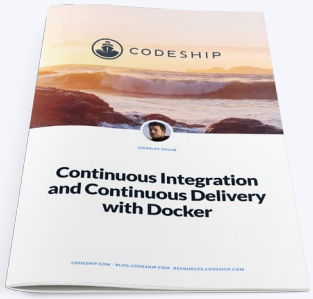
Download this eBook

EBOOKS

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



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