Zero to production-ready Docker packaging: the big picture

Itamar Turner-Trauring

Contents

Legal disclaimer

This book and the information it contains are provided “as is”, without warranties of any kind, including without limitation the implied warranties of merchantability or fitness for any particular purpose. In no event shall the author, Itamar Turner-Trauring, or the copyright owner, Hyphenated Enterprises LLC, or any of their employees, agents or affiliates, be liable for any claim, costs, expenses, damages or other liability, whether under principles of contract, tort or otherwise, arising from, out of or in connection with the use of or reliance on the information contained in this book.

Introduction

You need to package an application with Docker for use in a production environment. But—it's complicated, and there's a lot of details, and you don't know where to start or what to prioritize.

But if you had a good mental model of why you're doing what you're doing, what's important, and how Docker packaging fits in with all the software development processes you're already doing, those details would be a lot less daunting.

This mini-book is focused on giving you that mental model, so to be upfront: this book won't cover any of the specific details you need to get Docker packaging right. At the end of the book I'll point you at some resources, but that's for later.

Instead, what you're going to learn is *why* you should care about Docker packaging at all, how these desired benefits interact with your organizational processes, and how these processes impact the specific Docker image you'll end up building. Lots of theory, very little practice.

This may seem like a distraction from getting your job done. **You've got software to ship, why should you spend time on philosophy, even if it's a short read?**

Here's why:

- **Basic principles change more slowly than software.** Docker is a relatively new technology, from 2013, but people have been writing and packaging software long before that, and will no doubt continue to do so long after Docker is gone. Understanding the underlying principles is a job skill that will last for a long time; for example, many of the same concepts apply to deployments based on virtual machines.

- **Packaging is a process: get the process wrong, and the product will be wrong too.** Consider, for example, security updates: as we will see, there are at least two approaches you can take. If you implement a "best practice" created with the first process in mind, while actually following the second process, you will end up with insecure images.

In short, before you learn the specifics, you need to understand the big picture.

Before we continue, there are some prerequisites. You should already understand the basics of how Docker packaging works: the differences between a container and an image, what an image registry is, the basic structure of a `Dockerfile`, and so on. If you need help with these concepts, consider reading my introductory book *Just Enough Docker Packaging*.

Assuming you know the basics, let's move on to consider packaging for production.

Why packaging? Why Docker images?

The first question we need to answer is why you're packaging your software at all. Can't you just checkout your code from version control on a machine somewhere, and then run it directly as is?

Technically, yes, but even for this simplistic deployment model we have some requirements. To deploy and run your code you will need:

- An operating system installed on the machine.
- A tool like `git` to checkout the source code.
- A Python interpreter to run the code.
- In some cases, a compiler to compile extensions, or some shared libraries your code depends on.

And that's just the starting point. Eventually, time passes and things change:

- Your code changes.
- All your dependencies, from Python to shared libraries, may also change. Sometimes to fix bugs or security problems, sometimes to add new features, sometimes in a backwards compatible way, sometimes incompatibly.

If you start with the simplistic `git pull` approach, you can certainly pull new code, install new dependencies as needed, upgrade Python manually, and so on.

The problem with this approach is that the machine will end up in any one of many arbitrary combinations of versions of your code and its dependencies. At the same time, your development machine may be in some other slightly different combination. **With arbitrary versions of code in different environments, how can you be certain the code that worked on your laptop will also work in production?**

Once you run the same code on two machines, or add additional developers, the problem gets worse: now you might have three, four, ten different combinations of versions and dependencies.

And that's why you want packaging, and that's why Docker images are an excellent form of packaging:

- A Docker image contains *everything* you need, at least on the level of single filesystem, to run an application: the operating system files, the shared libraries, a Python interpreter, your code.
- A Docker image is immutable: you can't change it. Every time you run a process off a Docker image, you know you're starting from scratch with the same identical dependencies and code.
- A Docker image can be shared, so you can trust you're running the same code whether it's on your laptop or on a production server.

The downside of a Docker image is that it is a single snapshot in time, even as the passage of time results in changes to your code and its dependencies. As a result, you will need to create not just one Docker image, but many Docker images over time, driven by both internal processes like code changes, and external events like security updates to your dependencies.

In short, packaging for production is a process, and a process that continues over time. Some parts of this process will be embodied in artifacts like your `Dockerfile`, build scripts, and the like. But other parts will be organizational processes that will require human intervention, for example upgrading to newer but incompatible dependencies.

An iterative process for Docker packaging

So how can you create these artifacts and corresponding processes?

Instead of trying to build up all the necessary artifacts in one go, and think through all the different organizational processes you need all at the same time, you can approach Docker packaging in an iterative manner. The idea is that:

- After each step in this process, you will have something useful, even if not perfect.
- You implement the most important parts first, so that if you're pulled away to work on something else you're always at a good stopping point.

Here are the steps I suggest you follow:

1. Get your application running in Docker.
2. Implement basic security.
3. Automate the builds.
4. Ensure easier debugging, and operational correctness.
5. Make the builds reproducible.
6. Speed up the build time, and make images smaller.

In practice you might choose to do the steps in a different order, for example doing reproducibility earlier, but the given order is a reasonable starting point for most situations.

Next, we'll go through each step, explaining:

- Why it's the logical next step.
- The decisions you'll be making.
- The ongoing processes you will need to launch.

Again, I won't be going into specific implementation details about how to do any of these steps—that's a 100-page book, and this isn't it. But after we go through the whole process I'll point you at resources to help you do the actual work.

Step 1: Your application runs

Having your application run inside a Docker image is your first step: if your Docker image can't run your application, it's not particularly useful. This won't be perfect packaging, since it's only the first step, but you will improve the packaging with each additional iteration.

The real world is more complex than a simplified model. So in practice, some of the work you'll be doing at this stage matches the goals of later steps. For example, choosing a stable base image is part of reproducibility, but you should be doing it here.

The main focus, however, should be on doing just the minimum necessary to get something working.

Deciding the image interface

Since you're starting from scratch, you will need to decide how your application will be configured: will you use environment variables, configuration files, or both?

For network servers you'll want to think about which ports are public and which are private.

For batch jobs that process data you'll want to think about how input files will be passed in, for example a volume mount at a known directory, and how output will be stored.

You should of course document all these decisions, so that people using the image know how to use it.

Step 2: Security

The next step is making your image more secure. If your image is insecure you won't want to run it anywhere public, so it's worth doing this as early as possible so you can start using your image outside your development machine.

This includes not running as root, installing security updates, and other best practices.

Dealing with security updates

The key process you'll have to implement here is handling security updates for your dependencies. Given Docker images are immutable, security updates require rebuilding a new image with the newly updated dependency, and then redeploying if necessary.

To simplify somewhat, there are two basic approaches you can take:

- Always use latest versions: Whenever you rebuild the image, you use the latest versions of your dependencies to ensure you get the latest security fixes. In order to ensure updates are applied in a timely fashion, you need to either rebuild from scratch nightly or weekly, or whenever a security update becomes available.
- Use pinned versions: Your dependencies are pinned to specific versions, for example you only install v1.1.1 of a particular package. When a security update—or critical bug fix—becomes available you will need to regenerate your list of pinned dependencies to include the version with the fix, and then rebuild the image.

Either way you will be rebuilding and redeploying your image on a regular basis, as an ongoing process.

My general advice is to use the latest version approach for system packages: a stable long-term-support Linux distribution will make sure security updates are backwards compatible. For Python dependencies I recommend the pinning approach, because upgrades are much more likely to introduce incompatibilities with your code.

Step 3: Automated builds

Now that you have an image that runs your application and hopefully is somewhat secure, the next step is automating builds. Instead of building images manually, you want to build images automatically in your build or CI system. Whenever new code is pushed to version control, you'll build a new image. This means it's easier for multiple team members to get images built, and means you can also automate deploys if relevant.

You will likely want to push your images to an image registry, for later access. If you don't already have one you will need to set one up, either on your own servers or via a hosted service like your cloud provider.

If you've chosen the path of automated daily or weekly rebuilds for security updates, you'll want to implement that at this point.

Integrating with your development process

Now that it's not just you manually building one-off images, you need to think about how your image building integrates with your development process.

For example, a common way to use version control is feature branches: each feature gets its own branch. Eventually you open a pull request back on to the main branch. Tests get run on the pull request, there might be a code review, and eventually the pull request gets merged in to the main branch.

One approach to automating image building is to only build a Docker image off the main branch, after each branch is merged.

Alternatively, you might want to have tests that use the Docker image, which means you are going to end up building Docker images as part of

the pull request. You will then need to some way to differentiate between images from different branches—a common way is to base the Docker image tag on the branch name, so you might have `yourorg/yourimage:main` and `yourorg/yourimage:branch-12345`.

Looking further out, you might want to start thinking about how new releases of the image can be deployed automatically.

Step 4: Improving debugging and operational correctness

If you're running a server in production, you start having to worry about operational issues: how do you upgrade your software with minimal downtime? How can you detect broken processes automatically? How can you easily debug problems?

Improving debugging

Once builds are automated, your whole team will start generating Docker images just as a side-effect of normal software development. And so now you need to start thinking about how to support the debugging process your team uses.

Imagine an image is running in production, and a bug report comes in. Can you tell what version of the code the image running, so a developer can reproduce the problem locally? Are you getting sufficient logging to help debug the problem?

The next step then is making images easier to debug, for example by additional logging. You'll also want to add metadata to the image to keep track of where, how, and when it was created.

Operational correctness

In addition to debugging, you also have to think about minimizing downtime for servers, issues like how upgrading works, and how to notice wedged servers.

A lot of this comes down to application logic, and to integrating with your runtime environment—Heroku or Kubernetes, for example—but packaging does come into this. For example, health checks are typically part of your packaging, or at least adjacent to it. Likewise, how database schema upgrades work is closely tied to how your image starts up images, and how you deploy the images.

Step 5: Reproducible builds

In the first week or so of packaging and development, it's unlikely that any major changes will happen to your Python dependencies. As time goes on, the third-party software you depend is more and more likely to change. And that can break your build, or break your application.

We talked about this issue in step 2, security, but more from the perspective of security updates. Here we're considering the problem of changing dependencies from a different perspective: the tension between stability and stagnation.

Ensuring stability via reproducibility

In general, you want reproducible builds: when you rebuild your image with the same version of the source code, you want to get the same image. If you're always installing the latest version of third-party dependencies your application will end up breaking in unexpected ways, even if your code has only minor changes, because you'll end up installing incompatible dependencies.

For system packages like `glibc` my tendency is install the latest version automatically, and rely on a stable operating system to ensure backwards compatibility. But if you're depending on the Django web framework, you probably want to have the same version of Django used whenever you package your software.

What you need, then, is tooling that takes your logical dependencies—the packages you import—and turns them into transitively pinned dependencies. Pinning means you specify specific versions, for example Flask 1.1.1. Transitive meaning you also want to pin the dependencies of your dependencies, and their dependencies, and so on.

Preventing stagnation via an update process

At the same time, freezing your dependencies in place for too long is a bad idea. If you only upgrade your dependencies every three years, you might find yourself updating your code to deal with multiple major API changes at once. This makes debugging harder, and problems more likely.

You therefore need an ongoing organizational process for updating to newer dependencies. For example, you might upgrade every 3 months or so, which means chances are you'll only be facing one major dependency upgrade at a time.

You can of course automate some parts of this, by using services like GitHub's Dependabot that updates dependencies, but that only solves the mechanical part of changing the version, you may still need to manually upgrade your code.

Step 6: Faster builds and smaller images

At this point you should have packaging that from an operational perspective is in great shape. So next it's time to improve another part of the development process: the developer feedback loop.

If building a Docker image is a key part of your development cycle, then slow builds can become an expensive bottleneck. Instead of getting test results from CI in 2 minutes, you might have to wait another 10 minutes for the Docker image to build.

If you're doing continuous deployment and the newly built images get immediately deployed, slow builds also make it harder to get feedback about whether code works in production. And that can impede developers ability to feel secure about deploying at any time.

Similarly, large images can slow down deploys and testing, and can cause higher cloud computing costs due to disk and bandwidth charges.

The final step in the packaging process is therefore optimizing both build time and image size. This can involve relying on Docker's layer caching, multi-stage builds, and nitty-gritty configuration options of your package manager.

Implementing the process

At this point you should have a sense of the scope of Docker packaging for production, and some of the processes you'll be thinking about: security updates, dependency feature updates, deployments, pull requests. Actually implementing all this requires getting the details right, though, and Docker packaging has plenty of those. So how can you learn these details?

First, you can read the free articles on my website. There's a huge amount of tutorial-style content demonstrating specific best practices for Python and how you can use them.

Second, if you'd like to get going faster, consider reading my *Python on Docker Production Handbook*. It's a streamlined reference, covering 70+ best practices specifically for Python in production, and it's organized using the exact same process I describe in this mini-book[1].

Finally, if you have any questions, suggestions, or ideas you'd like share, please email me—I always love hearing from readers.

[1] Given the number of best practices to cover, I split up two of the steps for organizational purposes, so it actually uses an 8-step process. But essentially it's the same structure.