Docker
Imagenes eficientes

Raúl Estrada

Octubre 2020

1. Overview

During the past few years, Docker has become the de facto standard for containerization on Linux. Docker is easy to use and provides lightweight virtualization, making it ideal for building applications and microservices as more and more services run in the cloud.

Although creating our first images can be relatively easy, building an efficient image requires forethought. In this tutorial, we'll see examples of how to write efficient Docker images and the reasons behind each recommendation.

Let's start with the use of official images.

2. Base Your Image on an Official One

2.1. What Are Official Images?

Official Docker images **are those created and maintained by a team sponsored by Docker, or at least approved by them**. They manage the Docker images publicly on GitHub projects. They also make changes when vulnerabilities are discovered and ensure that the image is up-to-date and follows best practices.

Let's see this more clearly with an example that uses the Nginx official image. The creators of the webserver maintain this image.

Let's say we want to use Nginx to host our static website. We can create our Dockerfile and base it on the official image:

```
FROM nginx:1.19.2
COPY my-static-website/ /usr/share/nginx/html
```

Then we can build our image:

```
$ docker build -t my-static-website .
```

And lastly, run it:

```
$ docker run -p 8080:80 -d my-static-website
```

Our Dockerfile is only two lines long. The base official image took care of all the details of an Nginx server, like a default configuration file and ports that should be exposed.

More specifically, the base image stops Nginx from becoming a daemon and ending the initial process. This behavior is expected in other environments, but in Docker, this is interpreted as the end of the application, so the container terminates. The solution is to configure Nginx not to become a daemon. This is the configuration in the official image:

```
CMD ["nginx", "-g", "daemon off;"]
```

When we base our images on official ones, we avoid unexpected errors that are hard to debug. The official image maintainers are specialists in Docker and in the software we want to use, so we benefit from all their knowledge and potentially save time, too.

2.2. Images Maintained by Their Creators

Although not official in the sense explained before, there are other images in the Docker Hub that are also maintained by the creators of the application.

Let's illustrate this with an example. EMQX is an MQTT Message broker. Let's say we want to use this broker as one of the microservices in our application. We could base our image on theirs and add our configuration file. Or even better, we could use their provision for configuring EMQX through environment variables.

For example, to change the default port where EMQX is listening, we can add the *EMQX_LISTENER__TCP__EXTERNAL* environment variable:

```
$ docker run -d -e EMQX_LISTENER__TCP__EXTERNAL=9999 -p 9999:9999 emqx/emqx:v4.1.3
```

As the community behind a particular software, they are in the best position to provide a Docker image with their software.

In some cases, we won't find an official image of any kind, for the application we want to use. Even in those situations, we'll benefit from searching the Docker Hub for images we can use as a reference.

Let's take H2 as an example. H2 is a lightweight relational database written in Java. Although there are no official images for H2, a third party created one and documented it well. We can use their GitHub project to learn how to use H2 as a standalone server, and even collaborate to keep the project up-to-date.

Even if we use a Docker image project only as a starting point to build our image, we may learn more than starting from scratch.

3. Avoid Building New Images When Possible

When getting up-to-speed with Docker, we may get in the habit of always creating new images, even when there is little change compared to the base image. For those cases, **we may consider adding our configuration directly to the running container instead of building an image**.

Custom Docker images need to be re-built every time there is a change. They also need to be uploaded to a registry afterward. If the image contains sensitive information, we may need to store it on a private repository. In some cases, we may get more benefit by using base images and configuring them dynamically, instead of building a custom image every time.

Let's use HAProxy as an example to illustrate this. Like Nginx, HAProxy can be used as a reverse proxy. The Docker community maintains its official image.

Suppose we need to configure HAProxy to redirect requests to the appropriate microservices in our application. All that logic can be written in one configuration file, let's say *my-config.cfg*. The Docker image requires us to place that configuration on a specific path.

Let's see how we can run HAProxy with our custom configuration mounted on the running container:

```
$ docker run -d -v my-config.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro haproxy:2.2.2
```

This way, even upgrading HAProxy becomes more straightforward, as we only need to change the tag. Of course, we also need to confirm that our configuration still works for the new version.

If we're building a solution composed of many containers, we may already be using an orchestrator, like Docker Swarm or Kubernetes. They provide the means to store configuration and then link it to running containers. Swarm calls them Configs, and Kubernetes calls them ConfigMaps.

Orchestration tools already consider that we may store some configuration outside the images we use. Keeping our configuration outside the image may be the best compromise in some cases.

4. Create Slimmed Down Images

Image size is important for two reasons. First, **lighter images are transferred faster**. It may not look like a game-changer when we're building the image in our development machine. Still, when we build several images on a CI/CD pipeline and deploy maybe to several servers, the total time saved on each deployment may be perceptible.

Second, to achieve a slimmer version of our image, we need to **remove extra packages that the image isn't using. This will help us decrease the attack surface** and, therefore, increase the security of the images.

Let's see two easy ways to reduce the size of a Docker image.

4.1. Use Slim Versions When Available

Here, we have two main options: the slim version of Debian and the Alpine Linux distribution.

The slim version is an effort of the Debian community to prune unnecessary files from the standard image. **Many Docker images already are slimmed-down versions based on Debian**.

For example, the HAProxy and Nginx images are based on the slim version of the Debian distribution *debian:buster-slim*. Thanks to that, these images went from hundreds of MB to only a few dozen MB.

In some other cases, the image offers a slim version along with the standard full-size version. For example, the latest Python image provides a slim version, currently *python:3.7.9-slim*, which is almost ten times smaller than the standard image.

On the other hand, many images offer an Alpine version, like the Python image we mentioned before. **Images based on Alpine are usually around 10 MB in size**.

Alpine Linux was designed from the beginning with resource efficiency and security in mind. This makes it a perfect fit for base Docker images.

A point to keep in mind is that Alpine Linux chose some years ago to change system libraries from the more common *glibc* to *musl*. Although most software will work without issues, we do well to test our application thoroughly if we choose Alpine as our base image.

4.2. Use Multi-Stage Builds

The multi-stage build feature **allows building images in more than one stage in the same Dockerfile, typically using the result of a previous stage in the next one**. Let's see how this can be useful.

Let's say we want to use HAProxy and configure it dynamically with its REST API, the Data Plane API. Because this API binary is not available in the base HAProxy image, we need to download it during build time.

We can download the HAProxy API binary in one stage and make it available to the next:

```
FROM haproxy:2.2.2-alpine AS downloadapi
RUN apk add --no-cache curl
RUN curl -L
https://github.com/haproxytech/dataplaneapi/releases/download/v2.1.0/dataplaneapi_2.1.0_Linux_x86_64.tar.gz -
-output api.tar.gz
RUN tar -xf api.tar.gz
RUN cp build/dataplaneapi /usr/local/bin/

FROM haproxy:2.2.2-alpine
COPY --from=downloadapi /usr/local/bin/dataplaneapi /usr/local/bin/dataplaneapi
...
```

The first stage, *downloadapi*, downloads the latest API and uncompresses the tar file. The second stage copies the binary so that HAProxy can use it later. We don't need to uninstall *curl* or delete the downloaded tar file, as the first stage is completely discarded and won't be present in the final image.

There are some other situations where the benefit of multi-stage images is even clearer. For example, if we need to build from source code, the final image won't need any build tools. A first stage can install all building tools and build the binaries, and the next stage will only copy those binaries.

Even if we don't always use this feature, it's good to know that it exists. In some cases, it may be the best choice to slim down our image.

5. Conclusion

Containerization is here to stay, and Docker is the easiest way to start using it. Its simplicity helps us be productive quickly, although some lessons are only learned through experience.

In this tutorial, we reviewed some tips to build more robust and secure images. As containers are the building blocks of modern applications, the more reliable they are, the stronger our application will be.