Top of Form

Bottom of Form

Top of Form

Bottom of Form

**A hands-on guide to images and containers for developers**

Understand the key concepts behind images and containers. Then try a lab that demonstrates building and running images and containers.

By [Evan "Hippy" Slatis](https://opensource.com/users/hippyod)

May 23, 2022 | [3 Comments](https://opensource.com/article/22/5/guide-containers-images#comments) | %t min read

Top of Form

[Register](https://opensource.com/user/register) or [Login](https://opensource.com/user/login?destination=/article/22/5/guide-containers-images) to like

Bottom of Form



Image by:

Lucarelli via Wikimedia Commons. CC-BY-SA 3.0

Containers and Open Container Initiative (OCI) images are important open source application packaging and delivery technologies made popular by projects like Docker and Kubernetes. The better you understand them, the more able you will be to use them to enhance the consistency and scalability of your projects.

In this article, I will describe this technology in simple terms, highlight the essential aspects of images and containers for a developer to understand, then wrap up by discussing some best practices developers can follow to make their containers portable. I will also walk you through a simple lab that demonstrates building and running images and containers.

**What are images?**

Images are nothing more than a packaging format for software. A great analogy is Java's JAR file or a Python wheel. JAR (or EAR or WAR) files are simply ZIP files with a different extension, and Python wheels are distributed as gzipped tarballs. All of them conform to a standard directory structure internally.

Images are packaged as tar.gz (gzipped tarballs), and they include the software you're building and/or distributing, but this is where the analogy to JARs and wheels ends. For one thing, images package not just your software but all supporting dependencies needed to run your software, up to and including a complete operating system. Whereas wheels and JARs are usually built as dependencies but can be executable, images are almost always built to be executed and more rarely as a dependency.

Knowing the details of what's in the images isn't necessary to understand how to use images or to write and design software for them (if you're interested, read ["What is a container image?"](https://opensource.com/article/21/8/container-image)). From your perspective, and especially from the perspective of your software, what's important to understand is that the images you create will contain a *complete operating system*. Because images are packaged as if they're a complete operating system from the perspective of the software you wish to run, they are necessarily much larger than software packaged in a more traditional fashion.

Note that *images are immutable*. They cannot be changed once they are built. If you modify the software running on the image, you must build an entirely new image and replace the old one.

[Skip to content](https://opensource.com/article/22/5/guide-containers-images)

Linux Containers

[What are Linux containers?](https://www.redhat.com/en/topics/containers/whats-a-linux-container?intcmp=70160000000h1s6AAA)

[What is Kubernetes?](https://www.redhat.com/en/topics/containers/what-is-kubernetes?intcmp=70160000000h1s6AAA)

[Free online course: Deploy containerized applications](https://www.redhat.com/en/services/training/do080-deploying-containerized-applications-technical-overview?intcmp=70160000000h1s6AAA)

[eBook: A guide to Kubernetes for SREs and sysadmins](https://opensource.com/downloads/kubernetes-sysadmin?intcmp=70160000000h1s6AAA)

[Free online course: Running containers with Red Hat technical overview](https://www.redhat.com/en/services/training/rh065-running-containers-red-hat-technical-overview?intcmp=70160000000h1s6AAA)

[Podman cheat sheet](https://developers.redhat.com/cheat-sheets/podman-cheat-sheet?intcmp=70160000000h1s6AAA)

[The latest articles on Linux containers](https://opensource.com/tags/containers?intcmp=70160000000h1s6AAA)

**Tags**

When images are created, they are created with a unique hash, but they are typically identified with a human-readable name such as ubi, ubi-minimal, openjdk11, and so on. However, there can be different versions of the image for each of their names, and those are typically differentiated by tags. For example, the openjdk11 image might be tagged as jre-11.0.14.1\_1-ubi and jre-11.0.14.1\_1-ubi-minimal, denoting image builds of the openjdk11 software package version 11.0.14.1\_1 installed on a Red Hat ubi and ubi minimal image, respectively.

**What are containers?**

Containers are images that have been realized and executed on a host system. Running a container from an image is a two-step process: create and start. Create takes the image and gives it its own ID and filesystem. Create (as in docker create, for example) can be repeated many times in order to create many instances of a running image, each with its own ID and filesystem. Starting the container will launch an isolated process on the host machine in which the software running inside the container will behave as if it is running in its very own virtual machine. A container is thus an isolated process on the host machine, with its own ID and independent filesystem.

From a software developer's perspective, there are two primary reasons to use containers: consistency and scalability. These are related to each other, and together they allow projects to use one of the most promising innovations to come to software development in recent years, the principle of *"Build once, deploy many."*

**Consistency**

Because images are immutable and include all of the dependencies needed to run your software from the OS on up, you gain consistency wherever you choose to deploy it. This means whether you launch an image as a container in a development, test, or any number of production environments, the container will run exactly the same way. As a software developer, you won't have to worry about whether any of those environments are running on a different host operating system or version, because the container is running the same operating system every time. That's the benefit of packaging your software along with its complete runtime environment, rather than just your software without the complete set of dependencies needed to run it.

This consistency means that in almost all cases, when an issue is found in one environment (for example, production), you can be confident that you'll be able to reproduce that issue in development or some other environment, so you can confirm the behavior and focus on fixing it. Your project should never get mired in and stumped by the dreaded "But it works on my machine" problem again.

**Scalability**

Images contain not only your software but also all the dependencies needed to run your software, including the underlying operating system. This means all processes running inside the container view the container as the host system, the host system is invisible to processes running inside the container, and, from the host system's point of view, the container is just another process it manages. Of course, virtual machines do almost the same thing, which raises a valid question: Why use container technology instead of a virtual machine? The answer lies in both speed and size.

Containers run only the software required to support an independent host without the overhead of having to mimic the hardware. Virtual machines must contain a complete operating system and mimic the underlying hardware. The latter is a very heavyweight solution, which also results in much larger files. Because containers are treated as just another running process from the host system's perspective, they can be spun up in seconds rather than minutes. When your application needs to scale quickly, containers will beat a virtual machine in resources and speed every time. Containers are also easier to scale back down.

Scaling is outside the scope of this article from a functional standpoint, so the lab will not be demonstrating this feature, but it's important to understand the principle in order to understand why container technology represents such a significant advance in the packaging and deployment of software.

Note: While it is possible to [run a container that does not include a complete operating system](https://opensource.com/article/22/2/build-your-own-container-linux-buildah), this is rarely done because the minimal images available are usually an insufficient starting point.

**How to find and store images**

Like every other type of software packaging technology, containers need a place where packages can be shared, found, and reused. These are called *image registries*, analogous to Java Maven and Python wheel repositories or npm registries.

These are a sampling of different image registries available on the internet:

* [**Docker Hub:**](http://hub.docker.com/) The original Docker registry, which hosts many Docker official images used widely among projects worldwide and provides opportunities for individuals to host their own images. One of the organizations that hosts images on Docker Hub is *adoptopenjdk*; view their repository for examples of images and tags for the [openjdk11](https://hub.docker.com/r/adoptopenjdk/openjdk11/tags?page=1&name=jre-11.0.14.1_1-ubi) project.
* [**Red Hat Image Registry:**](http://registry.redhat.io/) Red Hat's official image registry provides images to those with valid Red Hat subscriptions.
* [**Quay**](http://quay.io/): Red Hat's public image registry hosts many of Red Hat's publicly available images and provides providing opportunities for individuals to host their own images.

**Using images and containers**

There are two utilities whose purpose is to manage images and containers: [Docker](https://opensource.com/resources/what-docker) and [Podman](https://www.redhat.com/sysadmin/podman-guides-2020). They are available for Windows, Linux, and Mac workstations. From a developer's point of view, they are completely equivalent when executing commands. They can be considered aliases of one another. You can even install a package on many systems that will automatically change Docker into a Podman alias. Wherever Podman is mentioned in this document, Docker can be safely substituted with no change in outcome.

**[ Read next:**[**5 underused Podman features to try now**](https://www.redhat.com/sysadmin/podman-features-2?intcmp=7013a000002qLH8AAM) **]**

You'll immediately notice these utilities are very similar to [Git](https://git-scm.com/)) in that they perform tagging, pushing, and pulling. You will use or refer to this functionality regularly. They should not be confused with Git, however, since Git also manages version control, whereas images are immutable and their management utilities and registry have no concept of change management. If you push two images with the same name and tag to the same repository, the second image will overwrite the first with no way to see or understand what has changed.

**Subcommands**

The following are a sampling of Podman and Docker subcommands you will commonly use or refer to:

* build: build an image
  + Example: podman build -t org/some-image-repo -f Dockerfile
* image: manage images locally
  + Example: podman image rm -a will remove all local images.
* images:list images stored locally
* tag*:* tag an image
* container:manage containers
  + Example: podman container rm -a will remove all stopped local containers.
* run:create and start a container
  + also stop and restart
* pull/push: pull/push and image from/to a repository on a registry

**Dockerfiles**

Dockerfiles are the source files that define images and are processed with the build subcommand. They will define a parent or base image, copy in or install any extra software you want to have available to run in your image, define any extra metadata to be used during the build and/or runtime, and potentially specify a command to run when a container defined by your image is run. A more detailed description of the anatomy of a Dockerfile and some of the more common commands used in them is in the lab below. A link to the complete Dockerfile reference appears at the end of this article.

**Fundamental differences between Docker and Podman**

Docker is a daemon in Unix-like systems and a service in Windows. This means it runs in the background all the time, and it runs with root or administrator privileges. Podman is binary. This means it runs only on demand, and can run as an unprivileged user.

This makes Podman more secure and more efficient with system resources (why run all the time if you don't have to?). Running anything with root privileges is, by definition, less secure. When using images on the cloud, the cloud that will host your containers can manage images and containers more securely.

**Skopeo and Buildah**

While Docker is a singular utility, Podman has two other related utilities maintained by the Containers organization on GitHub: [Skopeo](https://github.com/containers/skopeo/blob/main/install.md) and [Buildah](https://github.com/containers/buildah). Both provide functionality that Podman and Docker do not, and both are part of the container-tools package group with Podman for installation on the Red Hat family of Linux distributions.

For the most part, builds can be executed through Docker and Podman, but Buildah exists in case more complicated builds of images are required. The details of these more complicated builds are far outside the scope of this article, and you'll rarely, if ever, encounter the need for it, but I include mention of this utility here for completeness.

Skopeo provides two utility functions that Docker does not: the ability to copy images from one registry to another and to delete an image from a remote registry. Again, this functionality is outside the scope of this discussion, but the functionality could eventually be of use to you, especially if you need to write some DevOps scripts.

**Dockerfiles lab**

The following is a very short lab (about 10 minutes) that will teach you how to build images using Dockerfiles and run those images as containers. It will also demonstrate how to externalize your container's configuration to realize the full benefits of container development and "Build once, deploy many."

**Installation**

The following lab was created and tested locally running Fedora and in a [Red Hat sandbox environment](https://developers.redhat.com/courses/red-hat-enterprise-linux/deploy-containers-podman) with Podman and Git already installed. I believe you'll get the most out of this lab running it in the Red Hat sandbox environment, but running it locally is perfectly acceptable.

You can also install Docker or Podman on your own workstation and work locally. As a reminder, if you install Docker, podman and docker are completely interchangeable for this lab.

**Building Images**

**1. Clone the Git repository from GitHub:**

$ git clone https://github.com/hippyod/hello-world-container-lab

**2. Open the Dockerfile:**

$ cd hello-world-container-lab

$ vim Dockerfile

1 FROM Docker.io/adoptopenjdk/openjdk11:x86\_64-ubi-minimal-jre-11.0.14.1\_1

2

3 USER root

4

5 ARG ARG\_MESSAGE\_WELCOME='Hello, World'

6 ENV MESSAGE\_WELCOME=${ARG\_MESSAGE\_WELCOME}

7

8 ARG JAR\_FILE=target/\*.jar

9 COPY ${JAR\_FILE} app.jar

10

11 USER 1001

12

13 ENTRYPOINT ["java", "-jar", "/app.jar"]

This Dockerfile has the following features:

* The FROM statement (line 1) defines the base (or parent) image this new image will be built from.
* The USER statements (lines 3 and 11) define which user is running during the build and at execution. At first, root is running in the build process. In more complicated Dockerfiles I would need to be root to install any extra software, change file permissions, and so forth, to complete the new image. At the end of the Dockerfile, I switch to the user with UID 1001 so that, whenever the image is realized as a container and executes, the user will not be root, and therefore more secure. I use the UID rather than a username so that the host can recognize which user is running in the container in case the host has enhanced security measures that prevent containers from running as the root user.
* The ARG statements (lines 5 and 8) define variables that can be used during the build process only.
* The ENV statement (line 6) defines an environment variable and value that can be used during the build process but will also be available whenever the image is run as a container. Note how it obtains its value by referencing the variable defined by the previous ARG statement.
* The COPY statement (line 9) copies the JAR file created by the Spring Boot Maven build into the image. For the convenience of users running in the Red Hat sandbox, which doesn't have Java or Maven installed, I have pre-built the JAR file and pushed it to the hello-world-container-lab repo. There is no need to do a Maven build in this lab. (Note: There is also an add command that can be substituted for COPY. Because the add command can have unpredictable behavior, COPY is preferable.)
* Finally, the ENTRYPOINT statement defines the command and arguments that should be executed in the container when the container starts up. If this image ever becomes a base image for a subsequent image definition and a new ENTRYPOINT is defined, it will override this one. (Note: There is also a cmd command that can be substituted for ENTRYPOINT. The difference between the two is irrelevant in this context and outside the scope of this article.)

Type :q and hit **Enter** to quit the Dockerfile and return to the shell.

**3. Build the image:**

$ podman build --squash -t test/hello-world -f Dockerfile

You should see:

STEP 1: FROM docker.io/adoptopenjdk/openjdk11:x86\_64-ubi-minimal-jre-11.0.14.1\_1

Getting image source signatures

Copying blob d46336f50433 done

Copying blob be961ec68663 done

...

STEP 7/8: USER 1001

STEP 8/8: ENTRYPOINT ["java", "-jar", "/app.jar"]

COMMIT test/hello-world

...

Successfully tagged localhost/test/hello-world:latest

5482c3b153c44ea8502552c6bd7ca285a69070d037156b6627f53293d6b05fd7

In addition to building the image the commands provide the following instructions:

The --squash flag will reduce image size by ensuring that only one layer is added to the base image when the image build completes. Excess layers will inflate the size of the resulting image. FROM, RUN, and COPY/ADD statements add layers, and best practices are to concatenate these statements when possible, for example:

RUN dnf -y --refresh update && \

dnf install -y --nodocs podman skopeo buildah && \

dnf clean all

The above RUN statement will not only run each statement to create only a single layer but will also fail the build should any one of them fail.

The -t flag is for naming the image. Because I did not explicitly define a tag for the name (such as test/hello-world:1.0), the image will be tagged as latest by default. I also did not define a registry (such as quay.io/test/hello-world), so the default registry will be localhost.

The -f flag is for explicitly declaring the Dockerfile to be built.

When running the build, Podman will track the downloading of "blobs." These are the image layers your image will be built upon. They are initially pulled from the remote registry, and they will be cached locally to speed up future builds.

Copying blob d46336f50433 done

Copying blob be961ec68663 done

...

Copying blob 744c86b54390 skipped: already exists

Copying blob 1323ffbff4dd skipped: already exists

**4. When the build completes, list the image to confirm it was successfully built:**

$ podman images

You should see:

REPOSITORY TAG IMAGE ID CREATED SIZE

localhost/test/hello-world latest 140c09fc9d1d 7 seconds ago 454 MB

docker.io/adoptopenjdk/openjdk11 x86\_64-ubi-minimal-jre-11.0.14.1\_1 5b0423ba7bec 22 hours ago 445 MB

**Running containers**

**5. Run the image:**

$ podman run test/hello-world

You should see:

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v2.5.4)

...

GREETING: Hello, world

GREETING: Hello, world

The output will continue printing "Hello, world" every three seconds until you exit:

crtl-c

**6. Prove that Java is installed only in the container:**

$ java -version

The Spring Boot application running inside the container requires Java to run, which is why I chose the base image. If you're running in the Red Hat sandbox environment for the lab, this prove sthat Java is installed only in the container, and not on the host:

-bash: java: command not found...

**Externalize your configuration**

The image is now built, but what happens when I want the "Hello, world" message to be different for each environment I deploy the image to? For example, I might want to change it because the environment is for a different phase of development or a different locale. If I change the value in the Dockerfile, I'm required to build a new image to see the message, which breaks one of the most fundamental benefits of containers—"Build *once*, deploy many." So how do I make my image truly portable so it can be deployed wherever I need it? The answer lies in externalizing the configuration.

7. Run the image with a new, external welcome message:

$ podman run -e 'MESSAGE\_WELCOME=Hello, world DIT' test/hello-world

You should see:

Output:

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v2.5.4)

...

GREETING: Hello, world DIT

GREETING: Hello, world DIT

Stop using by using crtl-c and adapt the message:

$ podman run -e 'MESSAGE\_WELCOME=Hola Mundo' test/hello-world

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v2.5.4)

...

GREETING: Hola Mundo

GREETING: Hola Mundo

The -e flag defines an environment variable and value to *inject* into the container at startup. As you can see, even if the variable was built into the original image (the ENV MESSAGE\_WELCOME=${ARG\_MESSAGE\_WELCOME} statement in your Dockerfile), it will be overridden. You've now externalized data that needed to change based on where it was to be deployed (for example, in a DIT environment or for Spanish speakers) and thus made your images portable.

**8. Run the image with a new message defined in a file:**

$ echo 'Hello, world from a file' > greetings.txt

$ podman run -v "$(pwd):/mnt/data:Z" \

-e 'MESSAGE\_FILE=/mnt/data/greetings.txt' test/hello-world

In this case you should see:

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v2.5.4)

...

GREETING: Hello, world from a file

GREETING: Hello, world from a file

Repeat until you hit crtl-c to stop

The -e flag in this case defines a path to the file at /mnt/data/greetings.txt that was mounted from the host's local file system with the -v flag at $(pwd)/greetings.txt (pwd is a bash utility that outputs the absolute path of the current directory, which in your case should be the hello-world-container-lab). You've now externalized data that needed to change based on where it was to be deployed, but this time your data was defined in an external file you mounted into the container. Environment variable settings are OK for a limited number of settings, but when you have several settings to apply, a file is a more efficient way of injecting the values into your containers.

Note: The :Z flag at the end of the volume definition above is for systems using [SELinux](https://www.redhat.com/en/topics/linux/what-is-selinux). SELinux manages security on many Linux distributions, and the flag allows the container access to the directory. Without the flag, SELinux would prevent the reading of the file, and an exception would be thrown in the container. Try running the command above again after removing the :Z to see a demonstration.

This concludes the lab.

**Developing for containers: externalize the configuration**

"Build once, deploy many" works because the immutable containers running in different environments don't have to worry about differences in the hardware or software required to support your particular software project. This principle makes software development, debugging, deployment, and ongoing maintenance much faster and easier. It also isn't perfect, and some minor changes have to be made in how you code to make your container truly portable.

The most important design principle when writing software for containerization is deciding what to externalize. These decisions ultimately make your images portable so they can fully realize the "Build once, deploy many" paradigm. Although this may seem complicated, there are some easy-to-remember factors to consider when deciding whether the configuration data should be injectable into your running container:

* **Is the data environment-specific?** This includes any data that needs to be configured based on where the container is running, whether the environment is a production, non-production, or development environment. Data of this sort includes internationalization configuration, datastore information, and the specific testing profile(s) you want your application to run under.
* **Is the data release independent?**Data of this sort can run the gamut from feature flags to internationalization files to log levels—basically, any data you might want or need to change between releases without a build and new deployment.
* **Is the data a secret?** Credentials should never be hard coded or stored in an image. Credentials typically need to be refreshed on schedules that don't match release schedules, and embedding a secret in an image stored in an image registry is a security risk.

The best practice is to choose where your configuration data should be externalized (that is, in an environment variable or a file) and only externalize those pieces that meet the above criteria. If it doesn't meet the above criteria, it is best to leave it as part of the immutable image. Following these guidelines will make your images truly portable and keep your external configuration reasonably sized and manageable.

**[ Free online course:**[**Containers, Kubernetes and Red Hat OpenShift Technical Overview**](https://www.redhat.com/en/services/training/do080-deploying-containerized-applications-technical-overview?intcmp=7013a000002qLH8AAM) **]**

**Summary**

This article introduces four key ideas for software developers new to images and containers:

1. **Images are immutable binaries:** Images are a means of packaging software for later reuse or deployment.
2. **Containers are isolated processes:** When they are created, containers are a runtime instantiation of an image. When containers are started, they become processes in memory on a host machine, which is much lighter and faster than a virtual machine. For the most part, developers only need to know the latter, but understanding the former is helpful.
3. **"Build once, deploy many":** This principle is what makes container technology so useful. Images and containers provide consistency in deployments and independence from the host machine, allowing you to deploy with confidence across many different environments. Containers are also easily scalable because of this principle.
4. **Externalize the configuration:** If your image has configuration data that is environment-specific, release-independent, or secret, consider making that data external to the image and containers. You can inject this data into your running image by injecting an environment variable or mounting an external file into the container.