Build your Java image

Estimated reading time: 13 minutes

Prerequisites

Work through the orientation and setup in Get started Part 1 (/get-started/) to understand Docker concepts. Refer to the following section for Java prerequisites.

Enable BuildKit

Before we start building images, ensure you have enabled BuildKit on your machine. BuildKit allows you to build Docker images efficiently. For more information, see Building images with BuildKit (/develop/develop-images/build_enhancements/).

BuildKit is enabled by default for all users on Docker Desktop. If you have installed Docker Desktop, you don't have to manually enable BuildKit. If you are running Docker on Linux, you can enable BuildKit either by using an environment variable or by making BuildKit the default setting.

To set the BuildKit environment variable when running the `docker build` command, run:

```
$ DOCKER_BUILDKIT=1 docker build .
```

To enable docker BuildKit by default, set daemon configuration in `/etc/docker/daemon.json` feature to `true` and restart the daemon. If the `daemon.json` file doesn't exist, create new file called `daemon.json` and then add the following to the file.

```
{
  "features":{"buildkit" : true}
}
```

Restart the Docker daemon.

Overview

Now that we have a good overview of containers and the Docker platform, let's take a look at building our first image. An image includes everything needed to run an application - the code or binary, runtime, dependencies, and any other file system objects required.

To complete this tutorial, you need the following:

- Docker running locally. Follow the instructions to download and install Docker (/get-docker/)
- A Git client
- An IDE or a text editor to edit files. We recommend using IntelliJ Community Edition (https://www.jetbrains.com/idea/download/).

Sample application

Let's clone the sample application that we'll be using in this module to our local development machine. Run the following commands in a terminal to clone the repo.

```
$ cd /path/to/working/directory
$ git clone https://github.com/spring-projects/spring-petclinic.git
$ cd spring-petclinic
```

Test the application without Docker (optional)

In this step, we will test the application locally without Docker, before we continue with building and running the application with Docker. This section requires you to have Java OpenJDK version 15 or later installed on your machine. Download and install Java (https://jdk.java.net/)

If you prefer to not install Java on your machine, you can skip this step, and continue straight to the next section, in which we explain how to build and run the application in Docker, which does not require you to have Java installed on your machine.

Let's start our application and make sure it is running properly. Maven will manage all the project processes (compiling, tests, packaging, etc). The **Spring Pets Clinic** project we cloned earlier contains an embedded version of Maven. Therefore, we don't need to install Maven separately on your local machine.

Open your terminal and navigate to the working directory we created and run the following command:

```
$ ./mvnw spring-boot:run
```

This downloads the dependencies, builds the project, and starts it.

To test that the application is working properly, open a new browser and navigate to `http://localhost:8080` .

Switch back to the terminal where our server is running and you should see the following requests in the server logs. The data will be different on your machine.

```
o.s.s.petclinic.PetClinicApplication     : Started
PetClinicApplication in 11.743 seconds (JVM running for 12.364)
```

Great! We verified that the application works. At this stage, you've completed testing the server script locally.

Press `CTRL-c` from within the terminal session where the server is running to stop it.

We will now continue to build and run the application in Docker.

Create a Dockerfile for Java

A Dockerfile is a text document that contains the instructions to assemble a Docker image. When we tell Docker to build our image by executing the `docker build` command, Docker reads these instructions, executes them, and creates a Docker image as a result.

Let's walk through the process of creating a Dockerfile for our application. In the root of your project, create a file named `Dockerfile` and open this file in your text editor.

What to name your Dockerfile?

The default filename to use for a Dockerfile is `Dockerfile` (without a file- extension). Using the default name allows you to run the `docker build` command without having to specify additional command flags.

Some projects may need distinct Dockerfiles for specific purposes. A common convention is to name these `Dockerfile.<something>` or `<something>.Dockerfile` . Such Dockerfiles can then be used through the `--file` (or `-f` shorthand) option on the `docker build` command. Refer to the "Specify a Dockerfile" section (/engine/reference/commandline/build/#specify-a-dockerfile--f) in the `docker build` reference to learn about the `--file` option.

We recommend using the default (`Dockerfile`) for your project's primary Dockerfile, which is what we'll use for most examples in this guide.

The first line to add to a Dockerfile is a `# syntax` parser directive (/engine/reference/builder/#syntax). While *optional*, this directive instructs the Docker builder what syntax to use when parsing the Dockerfile, and allows older Docker versions with BuildKit enabled to upgrade the parser before starting the build. Parser directives (/engine/reference/builder/#parser-directives) must appear before any other comment, whitespace, or Dockerfile instruction in your Dockerfile, and should be the first line in Dockerfiles.

```
# syntax=docker/dockerfile:1
```

We recommend using `docker/dockerfile:1` , which always points to the latest release of the version 1 syntax. BuildKit automatically checks for updates of the syntax before building, making sure you are using the most current version.

Next, we need to add a line in our Dockerfile that tells Docker what base image we would like to use for our application.

```
# syntax=docker/dockerfile:1

FROM openjdk:16-alpine3.13
```

Docker images can be inherited from other images. For this guide, we use the official `openjdk` image from Docker Hub with Java JDK that already has all the tools and packages that we need to run a Java application.

To make things easier when running the rest of our commands, let's set the image's working directory. This instructs Docker to use this path as the default location for all subsequent commands. By doing this, we do not have to type out full file paths but can use relative paths based on the working directory.

```
WORKDIR /app
```

Usually, the very first thing you do once you've downloaded a project written in Java which is using Maven for project management is to install dependencies.

Before we can run `mvnw dependency` , we need to get the Maven wrapper and our `pom.xml` file into our image. We'll use the `COPY` command to do this. The `COPY` command takes two parameters. The first parameter tells Docker what file(s) you would like to copy into the image. The second parameter tells Docker where you want that file(s) to be copied to. We'll copy all those files and directories into our working directory - `/app` .

```
COPY .mvn/ .mvn
COPY mvnw pom.xml ./
```

Once we have our `pom.xml` file inside the image, we can use the `RUN` command to execute the command `mvnw dependency:go-offline`. This works exactly the same way as if we were running `mvnw` (or `mvn`) dependency locally on our machine, but this time the dependencies will be installed into the image.

```
RUN ./mvnw dependency:go-offline
```

At this point, we have an Alpine version 3.13 image that is based on OpenJDK version 16, and we have also installed our dependencies. The next thing we need to do is to add our source code into the image. We'll use the `COPY` command just like we did with our `pom.xml` file above.

```
COPY src ./src
```

This `COPY` command takes all the files located in the current directory and copies them into the image. Now, all we have to do is to tell Docker what command we want to run when our image is executed inside a container. We do this using the `CMD` command.

```
CMD ["./mvnw", "spring-boot:run"]
```

Here's the complete Dockerfile.

```
# syntax=docker/dockerfile:1

FROM openjdk:16-alpine3.13

WORKDIR /app

COPY .mvn/ .mvn
COPY mvnw pom.xml ./
RUN ./mvnw dependency:go-offline

COPY src ./src

CMD ["./mvnw", "spring-boot:run"]
```

Create a `.dockerignore` file

To use a file in the build context, the Dockerfile refers to the file specified in an instruction, for example, a `COPY` instruction. To increase the performance of the build, and to exclude files and directories, we recommend that you create a `.dockerignore` file to the context directory. To improve the context load time, add a `target` directory within the `.dockerignore` file.

Build an image

Now that we've created our Dockerfile, let's build our image. To do this, we use the `docker build` command. The `docker build` command builds Docker images from a Dockerfile and a "context". A build's context is the set of files located in the specified PATH or URL. The Docker build process can access any of the files located in this context.

The build command optionally takes a `--tag` flag. The tag is used to set the name of the image and an optional tag in the format `name:tag`. We'll leave off the optional `tag` for now to help simplify things. If we do not pass a tag, Docker uses "latest" as its default tag. You can see this in the last line of the build output.

Let's build our first Docker image.

```
$ docker build --tag java-docker .
```

```
Sending build context to Docker daemon  5.632kB
Step 1/7 : FROM java:3.7-alpine
Step 2/7 : WORKDIR /app
...
Successfully built a0bb458aabd0
Successfully tagged java-docker:latest
```

View local images

To see a list of images we have on our local machine, we have two options. One is to use the CLI and the other is to use Docker Desktop (/desktop/dashboard/#explore-your-images). As we are currently working in the terminal let's take a look at listing images using the CLI.

To list images, simply run the `docker images` command.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
java-docker         latest              b1b5f29f74f0        47 minutes ago      567MB
```

You should see at least the we just built `java-docker:latest` .

Tag images

An image name is made up of slash-separated name components. Name components may contain lowercase letters, digits, and separators. A separator is defined as a period, one or two underscores, or one or more dashes. A name component may not start or end with a separator.

An image is made up of a manifest and a list of layers. Do not worry too much about manifests and layers at this point other than a "tag" points to a combination of these artifacts. You can have multiple tags for an image. Let's create a second tag for the image we built and take a look at its layers.

To create a new tag for the image we've built above, run the following command:

```
$ docker tag java-docker:latest java-docker:v1.0.0
```

The `docker tag` command creates a new tag for an image. It does not create a new image. The tag points to the same image and is just another way to reference the image.

Now, run the `docker images` command to see a list of our local images.

```
$ docker images
REPOSITORY    TAG      IMAGE ID          CREATED                  SIZE
java-docker   latest   b1b5f29f74f0      59 minutes ago         567MB
java-docker   v1.0.0   b1b5f29f74f0      59 minutes ago         567MB
```

You can see that we have two images that start with `java-docker`. We know they are the same image because if you take a look at the `IMAGE ID` column, you can see that the values are the same for the two images.

Let's remove the tag that we just created. To do this, we'll use the `rmi` command. The `rmi` command stands for "remove image".

```
$ docker rmi java-docker:v1.0.0
Untagged: java-docker:v1.0.0
```

Note that the response from Docker tells us that the image has not been removed but only "untagged". You can check this by running the `docker images` command.

```
$ docker images
REPOSITORY      TAG     IMAGE ID       CREATED               SIZE
java-docker     latest  b1b5f29f74f0   59 minutes ago        567MB
```

Our image that was tagged with `:v1.0.0` has been removed, but we still have the `java-docker:latest` tag available on our machine.

Next steps

In this module, we took a look at setting up our example Java application that we'll use for the rest of the tutorial. We also created a Dockerfile that we used to build our Docker image. Then, we took a look at tagging our images and removing images. In the next module, we'll take a look at how to:

Run your image as a container (/language/java/run-containers/)

Feedback

Help us improve this topic by providing your feedback. Let us know what you think by creating an issue in the Docker Docs (https://github.com/docker/docker.github.io/issues/new?title=[Java%20docs%20feedback]) GitHub repository. Alternatively, create a PR (https://github.com/docker/docker.github.io/pulls) to suggest updates.

Java (/search/?q=Java), build (/search/?q=build), images (/search/?q=images), dockerfile (/search/?q=dockerfile)