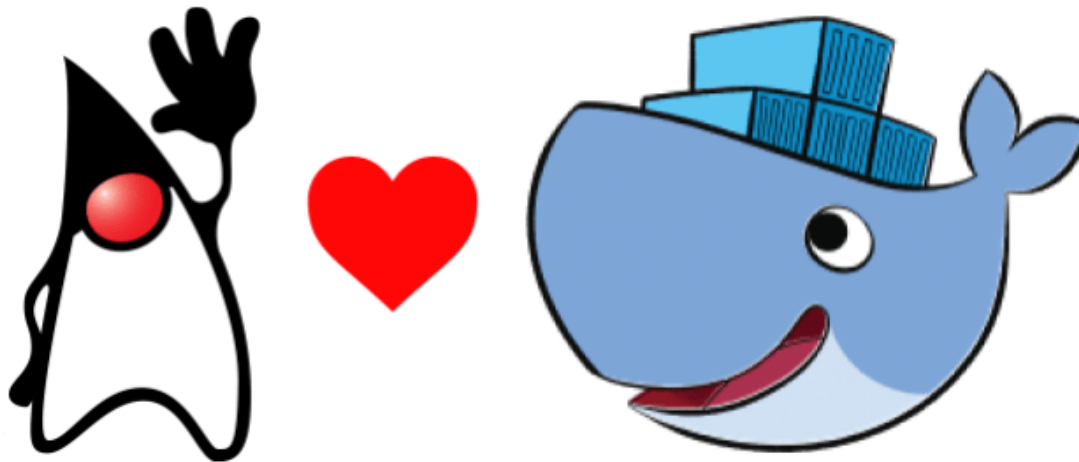


Create your FREE Codefresh account and start making pipelines fast.

Create Account

HOME > BLOG

Search the blog



DOCKER TUTORIALS

Three Ways to Create Docker Images for Java

6 min read



Anna Baker · Mar 05, 2020

Long before Dockerfiles, Java developers worked with single deployment units (WARs, JARs, EARs, etc.). As you likely know by now, it is best practice to work in micro-services, deploying a small number of deployment units per JVM. Instead of one giant, monolithic application, you build your application such that each service can run on its own.

This is where Docker comes in! If you wish to upgrade a service, rather than redeploying your jar/war/ear to a new instance of an application server, you can just build a new Docker image with the upgraded deployment unit.

In this post, I will review 3 different ways to create Docker images for Java applications. If you want to follow along feel free to clone my repository at <https://github.com/annabaker/docker-with-java-demos>.

Prerequisites

- Docker is [installed](#)

- Maven is [installed](#) (for example #1)
- You have a simple Spring Boot application (I used the [Spring Initializr](#) project generator with a Spring Web dependency)

First way: Package-only Build

In a package-only build, we will let Maven (or your build tool of choice) control the build process.

Unzip the Spring Initializr project you generated as part of the prerequisites. In the parent folder of your Spring Boot application, create a Dockerfile. In a terminal, run:

```
1 $ unzip demo.zip
2 $ cd demo
3 $ nano Dockerfile
```

Paste the following and save:

```
1 # we will use openjdk 8 with alpine as it is a very small linux distro
2 FROM openjdk:8-jre-alpine3.9
3
4 # copy the packaged jar file into our docker image
5 COPY target/demo-0.0.1-SNAPSHOT.jar /demo.jar
6
7 # set the startup command to execute the jar
```

```
8 | CMD ["java", "-jar", "/demo.jar"]
```

- The FROM layer denotes which parent image to use for our child image
- The COPY layer will copy the local jar previously built by Maven into our image
- The CMD layer tells Docker the command to run inside the container once the previous steps have been executed

Now, let's package our application into a .jar using Maven:

```
1 | $ mvn clean package
```

...and then build the Docker image. The following command tells Docker to fetch the Dockerfile in the current directory (the period at the end of the command). We build using the username/image name convention, although this is not mandatory. The -t flag denotes a Docker tag, which in this case is 1.0-SNAPSHOT. If you don't provide a tag, Docker will default to the tag :latest.

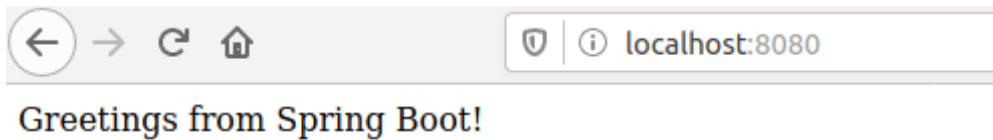
```
1 | $ docker build -t anna/docker-package-only-build-demo:1.0-SNAPSHOT .
```

To run the container from the image we just created:

```
1 | $ docker run -d -p 8080:8080 anna/docker-package-only-build-demo:1.0-SNAPSHOT
```

-d will run the container in the background (detached mode), and -p will map our local port 8080 to the container's port of 8080.

Navigate to localhost:8080, and you should see the following:



Spring inside docker

Once you are satisfied with your testing, stop the container.

```
1 | $ docker stop <container_id>
```

Pros to this approach:

- Results in a light-weight Docker image
- Does not require Maven to be included in the Docker image
- Does not require any of our application's dependencies to be packaged into the image
- You can still utilize your local Maven cache upon application layer changes, as opposed to methods 2 and 3 which we will discuss later

Cons to this approach:

- Requires Maven and JDK to be installed on the host machine
- The Docker build will fail if the Maven build fails/is not executed beforehand — this becomes a problem when you want to integrate with services that automatically “just build” using the present Dockerfile

Second way: Normal Docker Build

In a “normal” Docker build, Docker will control the build process.

Modify the previous Dockerfile to contain the following:

```
1 # select parent image
2 FROM maven:3.6.3-jdk-8
3
4 # copy the source tree and the pom.xml to our new container
5 COPY ./ ./
6
7 # package our application code
8 RUN mvn clean package
9
10 # set the startup command to execute the jar
11 CMD ["java", "-jar", "target/demo-0.0.1-SNAPSHOT.jar"]
```

Now, let's build a new image as we did in Step 1:

```
1 | $ docker build -t anna/docker-normal-build-demo:1.0-SNAPSHOT .
```

And run the container:

```
1 | $ docker run -d -p 8080:8080 anna/docker-normal-build-demo:1.0-SNAPSHOT
```

Again, to test your container, navigate to localhost:8080. Stop the container once you are finished testing.

Pros to this approach:

- Docker controls the build process, therefore this method does not require the build tool or the JDK to be installed on the host machine beforehand
- Integrates well with services that automatically “just build” using the present Dockerfile

Cons to this approach:

- Results in the largest Docker image of our 3 methods
- This build method not only packaged our app, but all of its dependencies and the build tool itself, which is not necessary to run the executable
- If the application layer is rebuilt, the mvn package command will force all Maven dependencies to be pulled from the remote repository all over again (you lose the local Maven cache)

Multi-stage Build (The ideal way)

With multi-stage Docker builds, we use multiple `FROM` statements for each build stage. Every `FROM` statement creates a new base layer, and discards everything we don't need from the

previous FROM stage.

Modify your Dockerfile to contain the following:

```
1 # the first stage of our build will use a maven 3.6.1 parent image
2 FROM maven:3.6.1-jdk-8-alpine AS MAVEN_BUILD
3
4 # copy the pom and src code to the container
5 COPY ./ ./
6
7 # package our application code
8 RUN mvn clean package
9
10 # the second stage of our build will use open jdk 8 on alpine 3.9
11 FROM openjdk:8-jre-alpine3.9
12
13 # copy only the artifacts we need from the first stage and discard the rest
14 COPY --from=MAVEN_BUILD /docker-multi-stage-build-demo/target/demo-0.0.1-SNAPSHOT.jar /demo.jar
15
16 # set the startup command to execute the jar
17 CMD ["java", "-jar", "/demo.jar"]
```

Build the image:

```
1 |$ docker build -t anna/docker-multi-stage-build-demo:1.0-SNAPSHOT .
```

And then run the container:

```
1 |$ docker run -d -p 8080:8080 anna/docker-multi-stage-build-demo:1.0-SNAPSHOT
```

Pros to this approach:

- Results in a light-weight Docker image
- Does not require the build tool or JDK to be installed on the host machine beforehand (Docker controls the build process)
- Integrates well with services that automatically “just build” using the present Dockerfile
- Only artifacts we need are copied from one stage to the next (i.e., our application’s dependencies are not packaged into the final image as in the previous method)
- Create as many build stages as you need
- Stop at any particular stage on an image build using the `--target` flag, i.e.

```
1 | docker build --target MAVEN_BUILD -t anna/docker-multi-stage-build-demo:1.0-SNAPSHOT .
```

Cons to this approach:

If the application layer is rebuilt, the `mvn package` command will force all Maven dependencies to be pulled from the remote repository all over again (you lose the local Maven cache)

Verification: How big are the images?

In a terminal, run:

```
1 | docker image ls
```

You should see something like the following:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
anna/docker-package-only-build-demo	1.0-SNAPSHOT	b2d98be350a3	About a minute ago	122MB
anna/docker-normal-build-demo	1.0-SNAPSHOT	33c923390af7	31 minutes ago	590MB
anna/docker-multi-stage-demo	1.0-SNAPSHOT	535134039856	2 days ago	102MB

Docker image size comparison

As you can see, the multi-stage build resulted in our smallest image, whereas the normal build resulted in our largest image. This should be expected since the normal build included our application code, all of its dependencies, and our build tooling, and our multi-stage build contained only what we needed.

Conclusion

Of the three Docker image build methods we covered, Multi-stage builds are the way to go. You get the best of both worlds when packaging your application code — Docker controls the build, but you extract only the artifacts you need. This becomes particularly important when storing containers on the cloud.

- You spend less time building and transferring your containers on the cloud, as your image is much smaller
- Cost — the smaller your image, the cheaper it will be to store
- Smaller surface area, aka removing additional dependencies from our image makes it less prone to attacks

Thanks for following along, and I hope this helps! You can find the source code for all three examples at <https://github.com/annabaker/docker-with-java-demos>

In a future post I will show you how to deal with caching issues specifically for Java builds.



Anna Baker



Anna Baker is a Software Engineer/Technical Writer. She previously worked at Red Hat and is passionate about the open source community. In her free time, she enjoys drawing and cooking dishes from all over the world.

Enjoy this article? Don't forget to share.



15 responses to “Three Ways to Create Docker Images for Java”

JD

at

Outstanding article!

Reply

NRH

at

I like this article so much

Reply

CAT

at

The simplest yet most informative of the java/docker how-to articles I have read.

Reply

RS

at

Wonderfully clear article. I learned so much in this short read.

Reply

Ahmad

at

Very useful information !

Reply

Vishal Trivedi

at

Excellent. To-The-Point with real time practical approach.

Reply

Don

at

Perfect, just what I was looking for

Reply

Yash

at

Hi, how we attach volume to the container . Kindly suggest solution.

Reply

Nisanth

at

Hi ..

When you are running the

```
$ docker build -t anna/docker-multi-stage-build-demo:1.0-SNAPSHOT .
```

command for the first time, the "Target" folder is not yet available in the workspace.

And the below error message is displayed while executing the COPY step..

```
#8 192.3 [INFO] BUILD SUCCESS
```

```
#8 192.3 [INFO] -----
```

```
#8 192.3 [INFO] Total time: 03:11 min
```

```
#8 192.3 [INFO] Finished at: 2020-12-23T07:20:04Z
```

```
#8 192.3 [INFO] -----
```

```
#8 DONE 192.4s
```

```
#9 [stage-1 2/2] COPY --from=MAVEN_BUILD /docker-multi-stage-build-demo/tar...
```

```
#9 ERROR: failed to walk /var/lib/docker/tmp/buildkit-mount448700145/docker-  
multi-stage-build-demo/target: lstat /var/lib/docker/tmp/buildkit-  
mount448700145/docker-multi-stage-build-demo/target: no such file or directory
```


Reply

Nagesh Kekan

at

modify COPY command like given below.

Do not use like /target/*.jar because you are already running a command inside your project folder.

use like this:

`COPY -from=MAVEN_BUILD target/*`

what is given in the blog in 3rd way/step:

`COPY -from=MAVEN_BUILD /docker-multi-stage-build-demo/target/*`

Reply

khaled

at

Thanks so much for this comparison and listing the pros and cons.
the article was really useful.

Reply

omurs

at

very nice and simplistic

Reply

HGs

at

Very helpful, concise and informative, which is what's needed here!

Reply

Arun

at

Hi,

Do we copy the class files, html files, images required by the project in the docker

image?

Reply

Sumit

at

Thanks for writing this

Reply

[Leave a Reply](#)

* All fields are required. Your email address will not be published.

Comment...

Name

Email

Post Comment

Related Posts

DOCKER TUTORIALS

**Using Docker Compose for
Local Code Testing and**

CONTINUOUS DEPLOYMENT/DELIVERY

**Minimize failed deployments
with Argo Rollouts and Smoke**

Development



David Widen

tests



Kostis Kapelonis

CATEGORIES

Helm Tutorials

Deployment Verification Testing

Serverless

Continuous Deployment/Delivery

Devops

Kubecon

Design

GitOps

Uncategorized

Docker Tutorials

Kubernetes Tutorials

Containers

Codefresh News

Continuous Integration

DevOps Tutorials

Docker Registry

Webinars


Security Testing

How Tos

LATEST TWEETS

FOLLOWERS: 6465

Codefresh @codefresh


With Codefresh, waiting for available workers to start a pipeline is a thing of the past. Download [📄](#) the full study to see how GoodRx  benefited from this feature.

<https://t.co/76ohLILF8N>

#GitOps #Kubernetes #CI/CD #pipelines



Codefresh @codefresh

There's still time to join us in SF  this December. Register to join us at ArgoCon '21 now! [👉 https://t.co/4jxNB3zG3h](https://t.co/4jxNB3zG3h)

#Argo #GitOps #Kubernetes #DevOps <https://t.co/a4AvBFTiTW>



Codefresh @codefresh

🧠 "Increased build speeds, a game-changing debugger, a UI that is easy to navigate - Codefresh has it all!"

See how Codefresh helped @SamaAI create pipelines that cover any and all of their need 🙌 <https://t.co/852z6fo0Mz>

Get a head start on
building better
pipelines!

Create Account

Schedule Demo >



Product

Kubernetes
Deployment

Codefresh Pricing

Status

Docker CI

Continuous Delivery
for Kubernetes

Helm Release
Management



Resources

Codefresh Live Events

Codefresh Plugins

Documentation

Case Studies

Kubernetes Guides

Docker Guides

How-to Videos

Containers Academy

GitHub

Company

About Codefresh

Contact Us

Careers

Blog

© 2021 Codefresh.
[Terms of Service.](#)

Google Cloud Platform
Technology Partner

Microsoft | AZURE
PARTNER



SOC 2 Type II
COMPLIANT

Microsoft
for Startups
CHOICE AWARD