spring

Spring Blog

All Posts Engineering Releases News and Events

Creating Docker images with Spring Boot 2.3.0.M1

ENGINEERING | PHIL WEBB | JANUARY 27, 2020 38 COMMENTS

Spring Boot 2.3.0.M1 has just been released and it brings with it some interesting new features that can help you package up your Spring Boot application into Docker images. In this blog post we'll take a look at the typical ways developers create Docker images, and show how they can be improved by using these new features.

Common Docker Techniques

Although it's always been possible to convert the fat jars produced by Spring Boot into Docker images, it's pretty easy to make less than optimal results. If you do a web search for "dockerize spring boot app", the chances are high you'll find an article or blog post suggesting you create a dockerfile that looks something like this:

COPY

```
 FROM openjdk:8-jdk-alpine
EXPOSE 8080
ARG JAR_FILE=target/my-application.jar
ADD ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Whilst this approach works fine, and it's nice and concise, there are a few things that are sub-optimal.

The first problem with above file is that the jar file is not unpacked. There's always a certain amount of overhead when running a fat jar, and in a containerized environment this can be noticeable. It's generally best to unpack your jar and run in an exploded form.

The second issue with the file is that it isn't very efficient if you frequently update your application. Docker images are built in layers, and in this case your application and all its things a bit more. If you put jar files in the layer before your application classes, Docker often only needs to change the very bottom layer and can pick others up from its cache.

Two new features are introduced in Spring Boot 2.3.0.M1 to help improve on these existing techniques: buildpack support and layered jars.

Buildpacks

If you've ever used an application platform such as Cloud Foundry or Heroku then you've probably used a buildpack, perhaps without even realizing it! Buildpacks are the part of the platform that takes your application and converts it into something that the platform can actually run. For example, Cloud Foundry's Java buildpack will notice that you're pushing a `.jar` file and automatically add a relevant JRE.

Until relatively recently buildpacks were tightly coupled to the platform and you couldn't easily use them independently. Thankfully they've now broken free, and with Cloud Native Buildpacks you can use them to create Docker compatible images that you can run anywhere.

Spring Boot 2.3.0.M1 includes buildpack support directly for both Maven and Gradle. This means you can just type a single command and quickly get a sensible image into your locally running Docker daemon. For Maven you can type `mvn spring-boot:build-image`, with Gradle it's `gradle bootBuildImage`. The name of the published image will be your application name and the tag will be the version.

Let's have a look at an example using Maven:

First create a new Spring Boot project using start.spring.io:

```
$ curl https://start.spring.io/starter.zip -d bootVersion=2.3.0.M1 -d dependencies=web -o dem
$ unzip demo.zip
```

Next ensure you have a local Docker installed and running, then type:

```
$ ./mvnw spring-boot:build-image
```

It will take a little time to run the first time around, but subsequent calls will be quicker. You should see something like this in the build log:

```
[INFO]  > Pulling builder image 'docker.io/cloudfoundry/cnb:0.0.43-bionic' 100%
[INFO]  > Pulled builder image 'cloudfoundry/cnb@sha256:c983fb9602a7fb95b07d35ef432c04ad61ae8
[INFO]  > Pulling run image 'docker.io/cloudfoundry/run:base-cnb' 100%
[INFO]  > Pulled run image 'cloudfoundry/run@sha256:ba9998ae4bb32ab43a7966c537aa1be153092ab0c
[INFO]  > Executing lifecycle version v0.5.0
[INFO]  > Using build cache volume 'pack-cache-5cbe5692dbc4.build'
[INFO]
[INFO]  > Running detector
[INFO]     [detector]    6 of 13 buildpacks participating
...
[INFO]
[INFO]  > Running restorer
[INFO]     [restorer]    Restoring cached layer 'org.cloudfoundry.openjdk:2f08c469c9a8adea1b6
...
[INFO]
[INFO]  > Running cacher
[INFO]     [cacher]      Reusing layer 'org.cloudfoundry.openjdk:2f08c469c9a8adea1b6ee3444ba2
[INFO]     [cacher]      Reusing layer 'org.cloudfoundry.jvmapplication:executable-jar'
[INFO]     [cacher]      Caching layer 'org.cloudfoundry.springboot:spring-boot'
[INFO]     [cacher]      Reusing layer 'org.cloudfoundry.springautoreconfiguration:46ab131165
[INFO]
[INFO] Successfully built image 'docker.io/library/demo:0.0.1-SNAPSHOT'
```

That's it! Your application has been compiled, packaged and converted to a Docker image. You can test it using:

```
$ docker run -it -p8080:8080 demo:0.0.1-SNAPSHOT
```

Note Unfortunately `M1` does not support Windows but it should work fine on a Mac or in a Linux VM. If you're using Windows, please use `2.3.0.BUILD-SNAPSHOT` for the time being.

The built-in support provided by Spring Boot provides a great way to get started with buildpacks. Since it's an implementation of the buildpack platform specification, it's also easy to migrate to more powerful buildpack tools such as `pack` or `kpack` with confidence that the same image will be produced.

Layered Jars

It's possible that you might not want to use buildpacks to create your images. Perhaps you have existing tools that are built around dockerfiles, or perhaps you just prefer them. Either way, we wanted to make it also easier to create optimized Docker images that can be built with a regular dockerfile so we've added support for "layered jars".

Spring Boot has always supported its own "fat jar" format that allows you to create an archive that you can run using `java -jar`. If you've ever looked into the contents of that jar, you'd see a structure that looks like this:

```
org/
  springframework/
    boot/
      loader/
        ...
BOOT-INF/
  classes/
    ...
  lib/
    ...
```

The jar is organized into three main parts:

- Classes used to bootstrap jar loading
- Your application classes in `BOOT-INF/classes`
- Dependencies in `BOOT-INF/lib`

Since this format is unique to Spring Boot, it's possible for us to evolve it in interesting ways. With Spring Boot `2.3.0.M1` we're providing a new `layout` type called `LAYERED_JAR`.

If you opt-in to the layered format and peek at the jar structure, you'll see something like this:

```
META-INF/
  MANIFEST.MF
org/
  springframework/
    boot/
      loader/
        ...
BOOT-INF/
  layers/
    <name>/
      classes/
        ...
      lib/
        ...
    <name>/
      classes/
        ...
      lib/
        ...
  layers.idx
```

You still see the bootstrap loader classes (you can still run `java -jar`) but now the `lib` and `classes` folders have been split up and categorized into layers. There's also a new `layers.idx` file that provides the order in which layers should be added.

Initially, we're providing the following layers out-of-the box:

- `dependencies` (for regular released dependencies)

- `resources` (for static resources)
- `application` (for application classes and resources)

This layering is designed to separate code based on how likely it is to change between application builds. Library code is less likely to change between builds, so it is placed in its own layers to allow tooling to re-use the layers from cache. Application code is more likely to change between builds so it is isolated in a separate layer.

Extracting layers

Even with the new format, there are still a few hoops you need to jump though in order to extract the files so that they can be copied by your `dockerfile`. Those loader classes need to be in the root of your jar, but you probably want them in an actual layer when you build the image. Of course, you can do this with some combination of `unzip` and `mv`, but we've tried to make it even easier by introducing the idea of "jar modes".

A `jarmode` is a special system property that you can set when you launch the jar. It allows the bootstrap code to run something entirely different from your application. For example, something that extracts the layers.

Here's how you can launch your jar with a `layertools` jar mode:

```
$ java -Djarmode=layertools -jar my-app.jar
```

This will provide the following output:

```
Usage:
  java -Djarmode=layertools -jar my-app.jar

Available commands:
  list     List layers from the jar that can be extracted
  extract  Extracts layers from the jar for image creation
  help     Help about any command
```

In this mode you can either `list` or `extract` layers.

Writing the `dockerfile`

Let's continue with the sample application that we generated above and add a `dockerfile` to it.

Start by editing the `pom.xml` and add the following:

```
                <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                        <configuration>
                                <layout>LAYERED_JAR</layout>
                        </configuration>
                </plugin>
        </plugins>
</build>
```

Then rebuild the jar:

```
$ mvn clean package
```

All being well, we should now have a layered jar with `jarmode` support. Test it with the following:

```
$ java -Djarmode=layertools -jar target/demo-0.0.1-SNAPSHOT.jar list
```

You should see the following output which tells us the layers and the order that they should be added:

```
dependencies
snapshot-dependencies
resources
application
```

We can now craft a `dockerfile` that extracts and copies each layer. Here's an example:

```
FROM adoptopenjdk:11-jre-hotspot as builder
WORKDIR application
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layertools -jar application.jar extract

FROM adoptopenjdk:11-jre-hotspot
WORKDIR application
COPY --from=builder application/dependencies/ ./
COPY --from=builder application/snapshot-dependencies/ ./
COPY --from=builder application/resources/ ./
COPY --from=builder application/application/ ./
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

This is a multi-stage dockerfile. The `builder` stage extracts the folders that are needed later. Each of the `COPY` commands relates to the layers that we listed earlier.

To build the image we can run:

```
$ docker build . --tag demo
$ docker run -it -p8080:8080 demo:latest
```

Summary

With buildpacks, dockerfiles and existing plugins such as jib, there’s certainly no shortage of ways to create Docker images. Each approach has pros and cons, but hopefully the new features we’re shipping in Spring Boot 2.3 will be helpful no matter which approach you choose.

Spring Boot 2.3 is currently scheduled to be released at the end of April and we’re very interested in feedback on the Docker images before then (raise issues, comment here or chat on Gitter).

Happy containerizing!

Featured Comment

Andy Wilkinson ➔ Francois Marot • 18 days ago

Layered jars and jib are largely equivalent and achieve a similar end goal. Any major pros and cons are likely to be down to personal preference. Without knowing what you like and dislike about jib, it’s difficult to answer.

The buildpack support is a different take on things. It allows you to build an image in the same way a Platform would. One advantage of this is that changes can be made in a centralised place (the builder) and all applications using the builder will then pick up that change.

2 ^ | ˅ • Share ›

Comments **Community** 1 **Login**

♡ **Recommend** 21 Tweet f Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Francois Marot • 19 days ago

explain the pros & cons of the SpringBoot method VS using Jib plugin as I currently do ? Thanks for the good work !

7 ⌃ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Francois Marot • 18 days ago

🏆 Featured by spring.io

Layered jars and jib are largely equivalent and achieve a similar end goal. Any major pros and cons are likely to be down to personal preference. Without knowing what you like and dislike about jib, it’s difficult to answer.

The buildpack support is a different take on things. It allows you to build an image in the same way a Platform would. One advantage of this is that changes can be made in a centralised place (the builder) and all applications using the builder will then pick up that change.

2 ⌃ | ⌄ • Reply • Share ›

Lukas Cardot ➦ Andy Wilkinson • 18 days ago

I would add that an important aspect and advantage of jib is that the image does not require Docker to be built.

1 ⌃ | ⌄ • Reply • Share ›

Arpan Sharma • 19 days ago

Good Article. I have been using Jib.
Query - Running a fat jar and not unpacking it will have overhead in terms of memory or performance?

2 ⌃ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Arpan Sharma
• 18 days ago • edited

A little, yes. In a Docker environment unpacking is recommended so that its contents can be split into separate image layers. Using the launcher with the unpacked jar then ensures that classpath ordering is unchanged.

2 ⌃ | ⌄ • Reply • Share ›

Balázs Mária Németh ➦ Andy Wilkinson
• 18 days ago • edited

it’s totally logical but could you please point me to a more detailed explanation? recommended by whom?

ok, so you edited the answer I was replying to not to use fat jars with docker (runtime) and what is the performance penalty you were talking about?

^ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Balázs Mária Németh • 18 days ago

The edit was a single-character change to correct a typo.

We, the Spring Boot team, recommend not using fat jars with Docker. By layering and exploding the jar, an update to your application's code only changes the layer that contains that code. The layer that contains all your dependencies can be reused, saving disk space and bandwidth.

There's a small performance penalty when running a fat jar with java -jar. It ever so slightly slows down class and resource loading. The more CPU-constrained your environment is, the more likely your are to notice. If you haven't noticed, it isn't something to worry about.

2 ^ | ⌄ • Reply • Share ›

Balázs Mária Németh ➦ Andy Wilkinson • 18 days ago • edited

the first point on its own justifies using this layered approach, and I also kind of understand the reasoning behind the fat jar class loading, what I don't see how docker is a problem here. I mean a normal VM or bare metal can be CPU constrained.
OK, I get it now. it's not docker per se, you're only suggesting that a containerized environment is not so wasteful usually.

^ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Balázs Mária Németh • 18 days ago

No one intended to claim that the problem was specific to Docker. However, I do think it's more likely to be

2 ⌃ | ⌄ • Reply • Share ›

Christian Tzolov • 10 days ago • edited

How the suggested new layering (e.g. dependencies, snapshot-dependencies, resources, application) improves the already existing JIB layering: https://github.com/GoogleCo... ?

1 ⌃ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Christian Tzolov • 10 days ago

A similar question was asked and answered already.

To expand on that answer a little bit, a benefit of the jar itself understanding the layers is that it will allow a buildpack to produce a layered image from that jar. It also positions things nicely for us to make Spring Boot-specific enhancements to the layering in future milestones and releases.

1 ⌃ | ⌄ • Reply • Share ›

Steven Gibbs • 17 days ago

Builder having a problem downloading the JRE via the proxy? I've set proxy settings in my maven settings, where else needs to be aware of the proxy? Odd that is says it succeed but i don't see the image on my machine after this runs.

```
> Running builder
[builder]
[builder] Cloud Foundry OpenJDK Buildpack v1.0.80
[builder] OpenJDK JRE 11.0.5: Contributing to layer
[builder] Downloading from
https://github.com/AdoptOpenJDK/openjdk11-
binaries/releases/download/jdk-11.0.5%2B10/OpenJDK11U-
jre_x64_linux_hotspot_11.0.5_10.tar.gz
[builder]
[builder] Cloud Foundry OpenJDK Buildpack v1.0.80
[builder] Get https://github.com/AdoptOpenJDK/openjdk11-
```

see more

1 ⌃ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Steven Gibbs • 17 days ago

The build continuing when a phase in the builder's lifecycle failed is a bug. Please subscribe to https://github.com/spring-p... for updates.

You should be able to configure the use of a proxy by

HTTPS_PROXY environment variables. There's an example of environment variable configuration in the Maven plugin's reference documentation: https://docs.spring.io/spri...

2 ^ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Andy Wilkinson • 17 days ago

I've opened an issue to see if we can provide sensible defaults for the proxy environment variables based on what Gradle and Maven are using.

3 ^ | ⌄ • Reply • Share ›

Dimitris Klimis • 18 days ago

That's good stuff. Do you surface anywhere which buildpack you're using? Can I control the image on which the build image is based? Finally, does this work offline?

1 ^ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Dimitris Klimis • 18 days ago

The default builder is cloudfoundry/cnb:0.0.43-bionic. The base image is determined by the builder. Offline usage isn't supported as it will be unable to pull the builder. Assuming that a builder with hot caches can work offline this may be something that we can improve upon. Please open an issue if it's something that you'd like us to try to support.

1 ^ | ⌄ • Reply • Share ›

Jack b • 3 days ago • edited

When setting the port in the applicaiton.yml or .properties this is not getting reflected in the container and is defaulting to 8080. Any reason why this might be?

^ | ⌄ • Reply • Share ›

Stéphane Nicoll ➦ Jack b • 2 days ago

I didn't experience that. Changing the port and running the container with `-p` and the proper port works for me. Feel free to join our gitter channel or ask further questions on StackOverflow.

^ | ⌄ • Reply • Share ›

ttddyy • 11 days ago

Hi,
This is great addition to 2.3.
I started recommending exploded with layering approach to our applications.

performance difference is ultimately the usage of `LaunchedURLClassLoader` when embedded layout is used vs `URLClassLoader` for exploded when `[Jar|War]Launcher` is used.
Since jars are embedded, reading(loading) classes from jar incurs costs; and, also adds overhead of security-manager, access-control, etc for dynamically loading them.

Do you have any numbers that shows the impact for exploded vs embedded(fat jar) ?
That would be helpful to promote other teams to move to use exploded layout with layering approach.

Thanks,

⌃ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ ttddyy • 10 days ago

The cost varies from application to application and is largely a function of the numbers of classes and resources that they load. For a small app that starts in under 1 second, the cost of java -jar is a few 10s of milliseconds. The real benefit of layering is in the size of the images and I suspect that's probably more compelling to most teams.

1 ⌃ | ⌄ • Reply • Share ›

Alexander Satek • 12 days ago

How to configure the output of "mvn spring-boot:build-image"?
I mean, how do I pass all the options I would normally have in a Dockerfile? e.g. arguments for the JVM running the application

Can and should this be used for build pipelines? How?

⌃ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Alexander Satek • 10 days ago

You should be able to specify a JAVA_OPTS environment variable on the plugin configuration (Maven) or the bootBuildImage task (Gradle) and this will be picked up by the appropriate buildpack and used when its defining the command to launch the JVM in the image.

I think there's some room for improvement in the Cloud Native Buildpacks documentation. It would be useful if it covered all the environment variables that are understood by a specific builder and its

⌃ | ⌄ • Reply • Share ›

Bahadir Tasdemir • 14 days ago

Very handy feature for Docker image optimisation, thank you very much 🙏

⌃ | ⌄ • Reply • Share ›

Mikael • 17 days ago

Is it possible to introduce additional layers? For example a layer for organisation/company internal artifacts.

⌃ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Mikael • 17 days ago

Not yet, but it's something we'd like to add in the future. It's listed in https://github.com/spring-p... as something we'd like to tackle now that M1 has been released.

1 ⌃ | ⌄ • Reply • Share ›

Mikael ➦ Andy Wilkinson • 17 days ago

Thanks! I'll keep an eye on that issue.

⌃ | ⌄ • Reply • Share ›

Tomáš Poledný • 17 days ago

Can I combine layered jar with Jib? We use Jib and we are really happy. Jib makes layers painlessly.

⌃ | ⌄ • Reply • Share ›

Andy Wilkinson ➦ Tomáš Poledný • 17 days ago

We explored that possibility when implementing the feature, but it doesn't really align with how Jib works. If you're really happy with Jib, then I would continue using it.

⌃ | ⌄ • Reply • Share ›

Hendi Santika • 18 days ago

Nice article. Is there any github repo for this sample? Thanks

⌃ | ⌄ • Reply • Share ›

Stéphane Nicoll ➦ Hendi Santika • 17 days ago

This article creates an app from scratch. Go to https://start.spring.io, create an app with Spring Boot 2.3.0.M1 (or later) and follow the rest of the instructions (running `mvn spring-boot:build-image` is all that it takes as long as you have the docker daemon running locally).

spring

Get the Spring newsletter

Email Address

□ Yes, I would like to be contacted by The Spring Team and Pivotal for newsletters, promotions and events per the terms of Pivotal's Privacy Policy

SUBSCRIBE

Get ahead

Pivotal offers training and certification to turbo-charge your progress.

Learn more

Get support

Spring Runtime offers support and binaries for OpenJDK™, Spring, and Apache Tomcat® in one simple subscription.

Learn more

Upcoming events

SpringOne Platform

Sep 21–24, Seattle

Why Spring

Microservices

Reactive

Event Driven

Cloud

Web Applications

Serverless

Batch

Learn

Quickstart

Guides

Blog

Community

Events

Team

Store

Projects

Training

Support

Thank You

© 2020 Pivotal Software, Inc. All Rights Reserved. Terms of Use • Privacy • Trademark Guidelines Preferencias para cookies

Apache®, Apache Tomcat®, Apache Kafka®, Apache Cassandra™, and Apache Geode™ are trademarks or registered trademarks of the Apache Software Foundation in the United States and/or other countries. Java™, Java™ SE, Java™ EE, and OpenJDK™ are trademarks of Oracle and/or its affiliates. Kubernetes® is a registered trademark of the Linux Foundation in the United States and other countries. Linux® is the registered trademark of Linus Torvalds in the United States and other countries. Windows® and Microsoft® Azure are registered trademarks of Microsoft Corporation. "AWS" and "Amazon Web Services" are trademarks or registered trademarks of Amazon.com Inc. or its affiliates. All other trademarks and copyrights are property of their respective owners and are only mentioned for informative purposes. Other names may be trademarks of their respective owners.