We Help You Build Systems That Make You **A DIGITAL LEADER**

Find out more

ThoughtWorks®

Layered JARs

Optimise your Docker images with Layered JARs

By Johanna Lang and Albert Attard

jlang@thoughtworks.com albert.attard@thoughtworks.com

Agenda

Docker

Layers

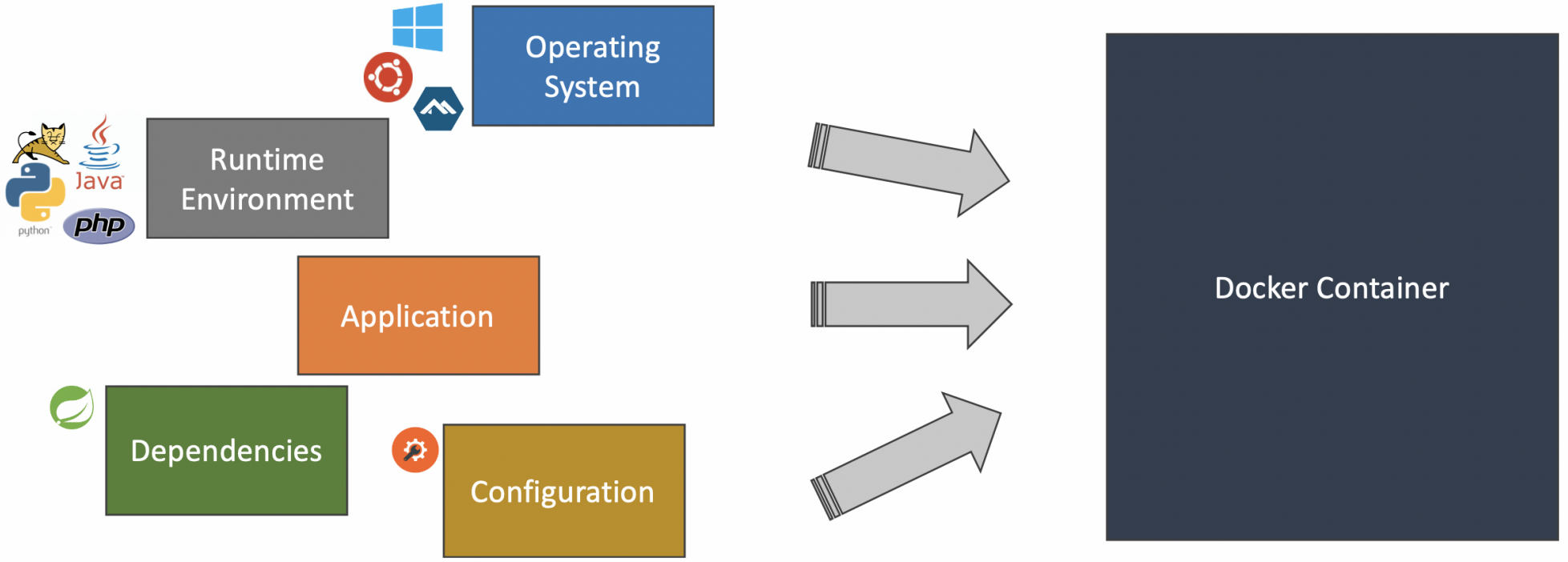
Spring Boot

Beyond Spring Boot

Docker

What is a Docker container?

- A standard deployment unit that encapsulates an application and all of its dependencies



How is a Docker container created?

- By running a *Docker image*

```
$ docker run \
  --rm \
  --name docker-container-demo \
  -p 8080:8080 \
  spkane/quantum-game:latest
```

➤ We don't have to worry about any specific runtime environment or any particular dependency version as everything is encapsulated in the Docker container

What is a Docker image?

- A **read-only** filesystem that contains
  - An operating system
  - The programs needed by the application (e.g. Java Runtime Environment)
  - The application executable, its dependencies, and configuration
- **Immutable** (cannot be modified once built)

➤ New image gets created every time a new version of our application is dockerized

Demo

Analyse a Docker image with *dive*

- Overview of *dive*, a tool for exploring a Docker image and its contents
- Analyse a Docker image with *dive*

How is a Docker image created?

- By building a *Dockerfile*:

```
$ docker build . -t boot-fat-jar:local
```

  The above command creates a Docker image and tags it as `boot-fat-jar:local`

- We can run this Docker image, creating a Docker container when doing so, using the given tag `boot-fat-jar:local`

```
$ docker run \
   --rm \
   --name boot-fat-jar-demo \
   -p 8080:8080 \
   boot-fat-jar:local
```

What is a *Dockerfile*?

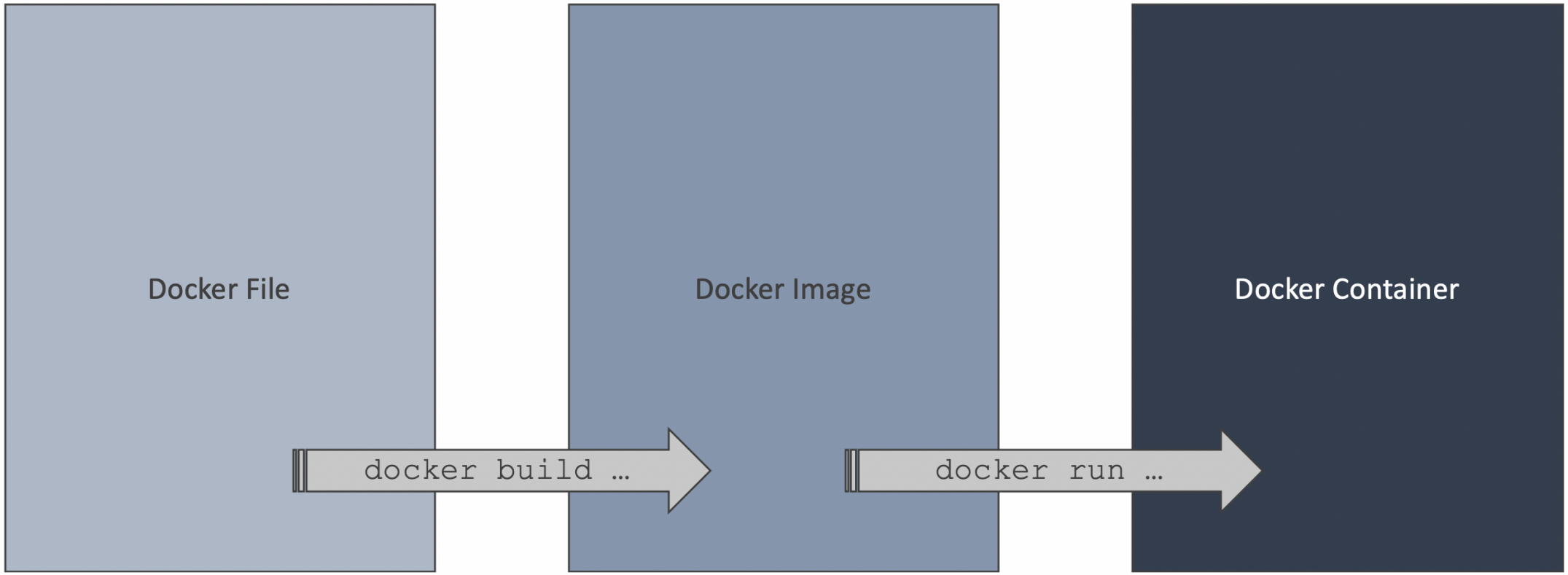
- A text file, usually named `Dockerfile`, that contains a set of instructions, used to create the Docker image
- The *Dockerfile* is the source file used to create the Docker image

How does a *Dockerfile* look like?

- Following is a typical *Dockerfile* that hosts a Java 8 application:

```
FROM adoptopenjdk:8u262-b10-jre-hotspot
WORKDIR /opt/app
COPY ./build/libs/*.jar application.jar
ENTRYPOINT ["java", "-jar", "application.jar"]
```

Lifecycle


Docker File
Docker Image
Docker Container
docker build …
docker run …

Layers

What are layers?

- Remember our *Dockerfile*?

```
FROM adoptopenjdk:8u262-b10-jre-hotspot
WORKDIR /opt/app
COPY ./build/libs/*.jar application.jar
ENTRYPOINT ["java", "-jar", "application.jar"]
```

- The above *Dockerfile* has four layers

1. `FROM ...`
2. `WORKDIR ...`
3. `COPY ...`
4. `ENTRYPOINT ...`

Intermediate Images

Demo

Build docker image and analyse layers with *dive*

- Build a Docker image
- Discuss layers and see Docker takes advantage of caching
- Analyse the Docker image, using *dive*

JAR file

.jar (= java archive) is a package file format

FatJAR

- A very common way to package a JVM based application is a **FatJAR**
- A FatJAR contains
  - The application
  - Resources that the application needs
  - The application dependencies
- It is standalone and can be executed using the following command

```
$ java -jar application.jar
```

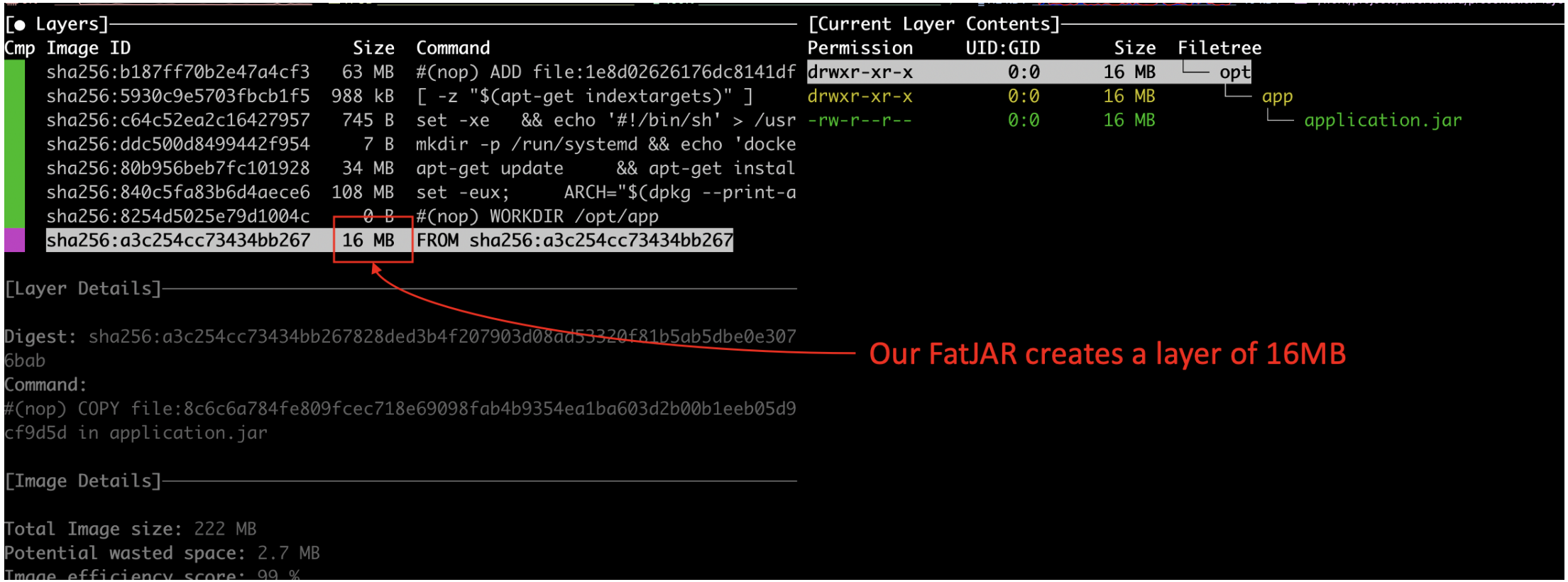
FatJAR in a Docker image

- The FatJAR is copied from our laptop into the Docker image using the `COPY` instruction

```
COPY ./build/libs/*.jar application.jar
```

- Every time the FatJAR is copied into a Docker image, a new layer is created
- Creating many large layers may consume large amounts of disk space

Size of FatJAR



Space requirements

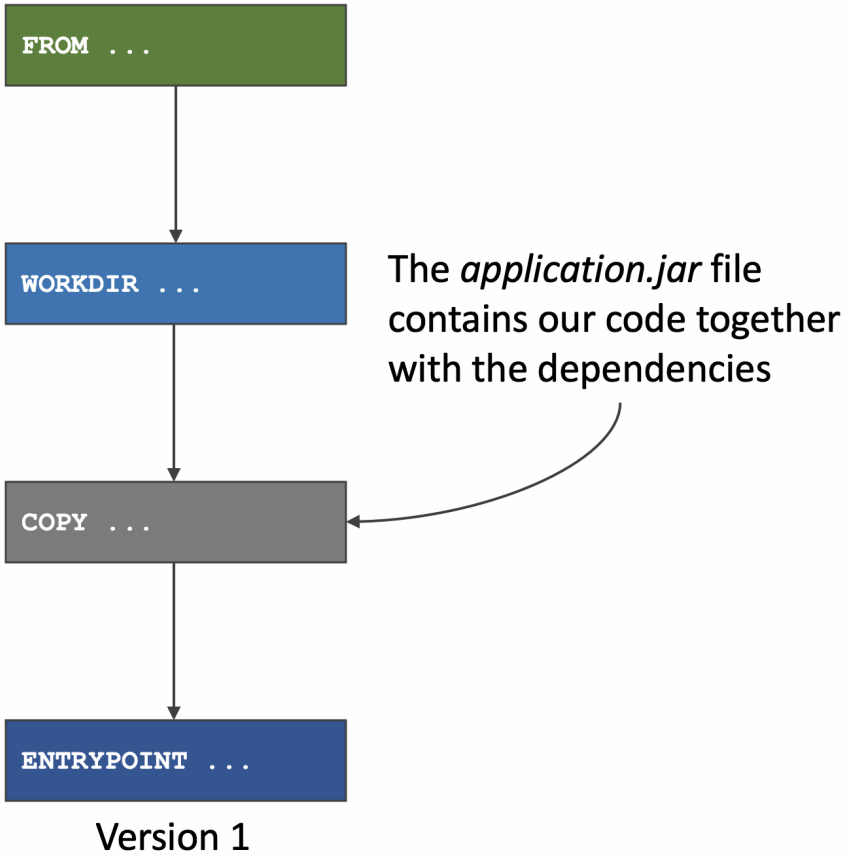
- Consider a team working 5 days a week and committing code 20 times per day
- Each commit is followed by a push, which triggers an automated build pipeline, which builds the application and **creates a new docker image**

Small change create large layer

- Our application (FatJAR) contains our code **and** its dependencies
- When new features are added, the dependencies are not necessarily updated

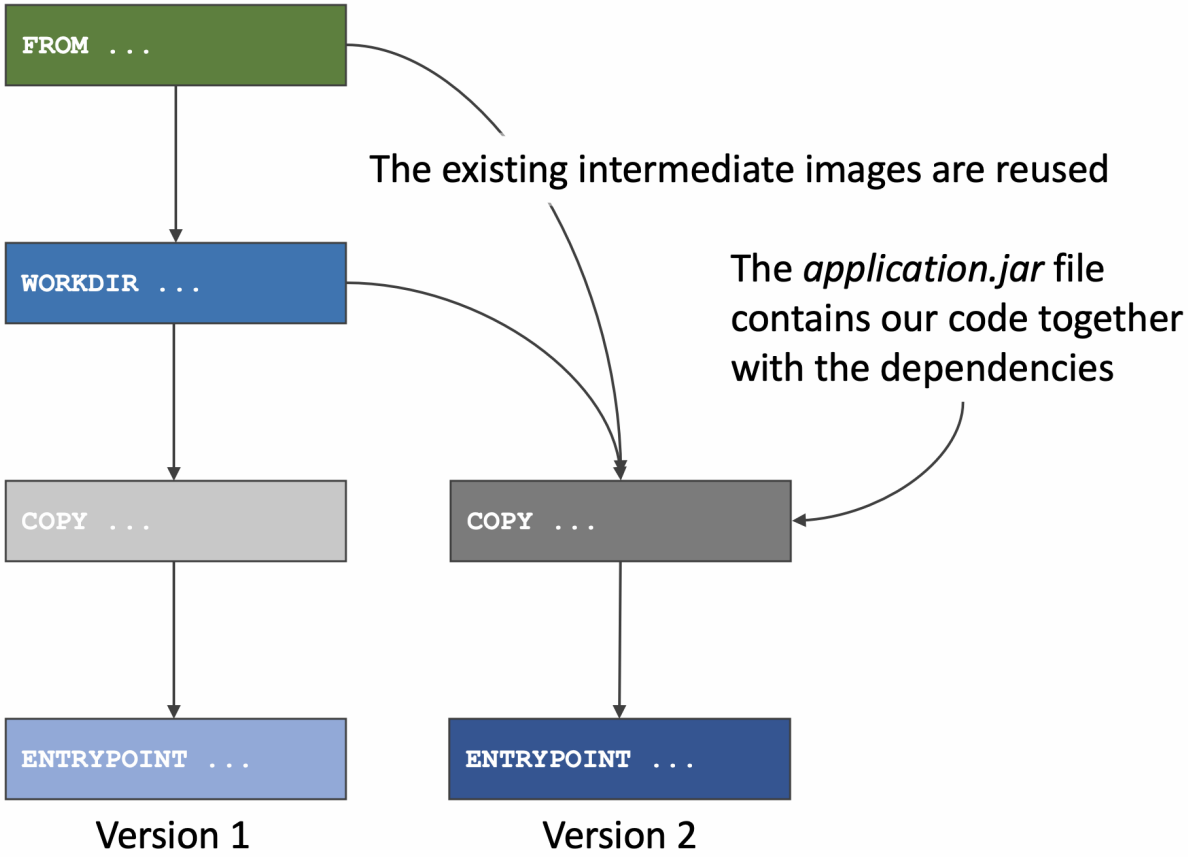
➤ **However, each small change in the code, creates a new docker layer of about 16MB in size**

The FatJAR application - Version 1



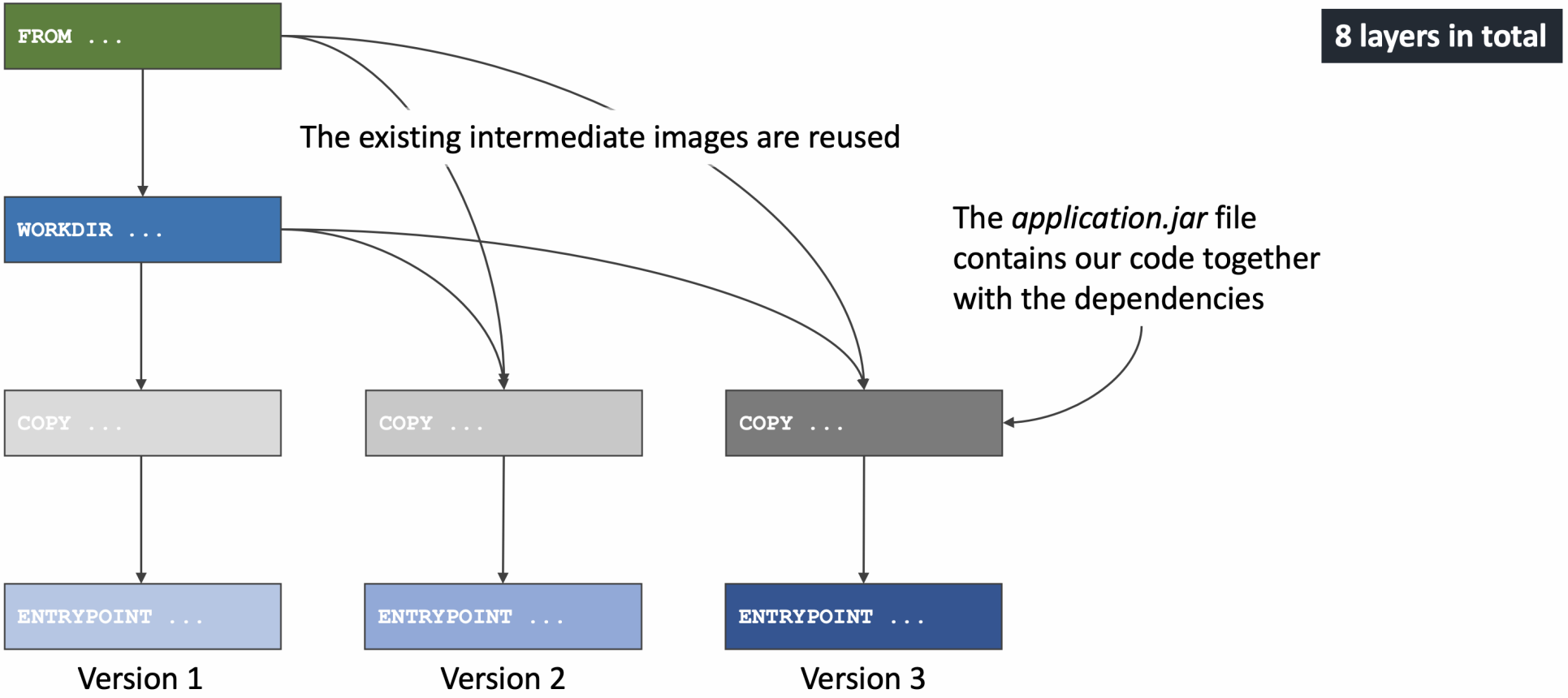
4 layers in total

The FatJAR application - Version 2

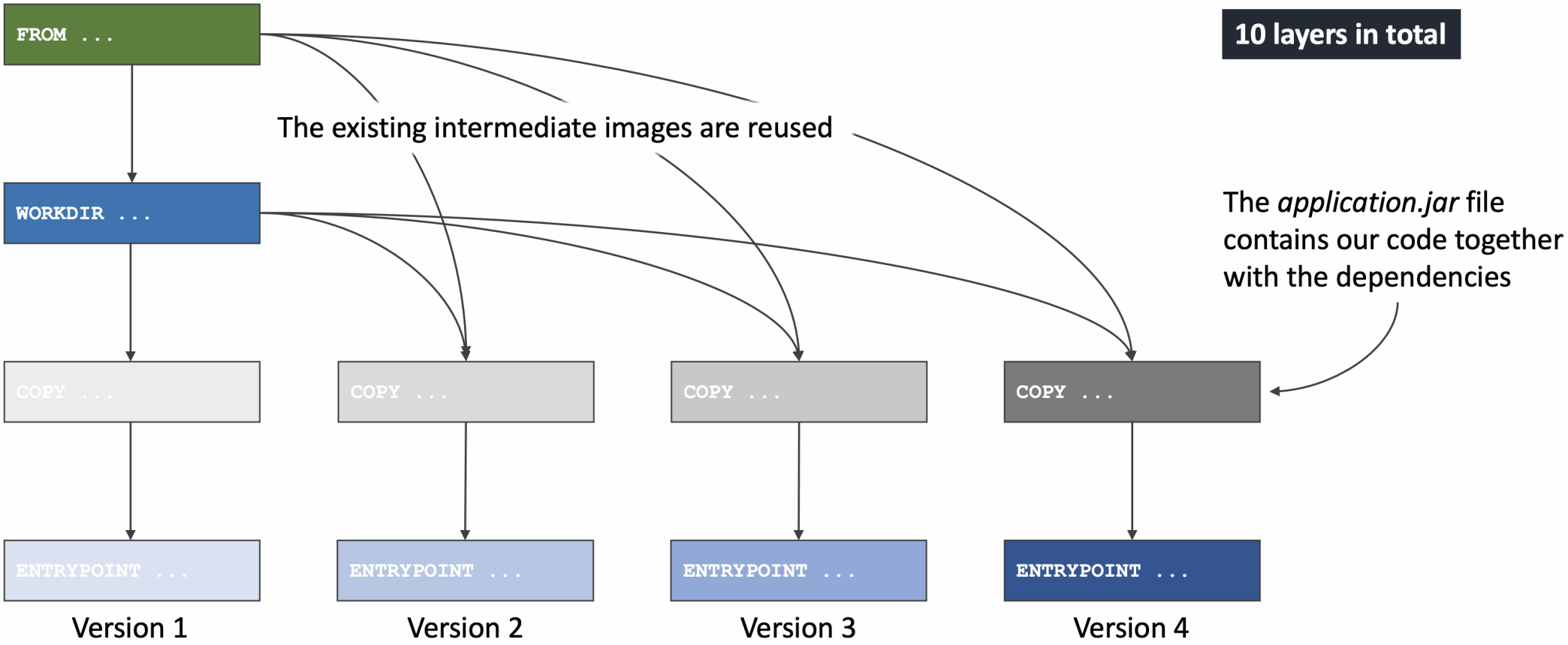


6 layers in total

The FatJAR application - Version 3



The FatJAR application - Version 4

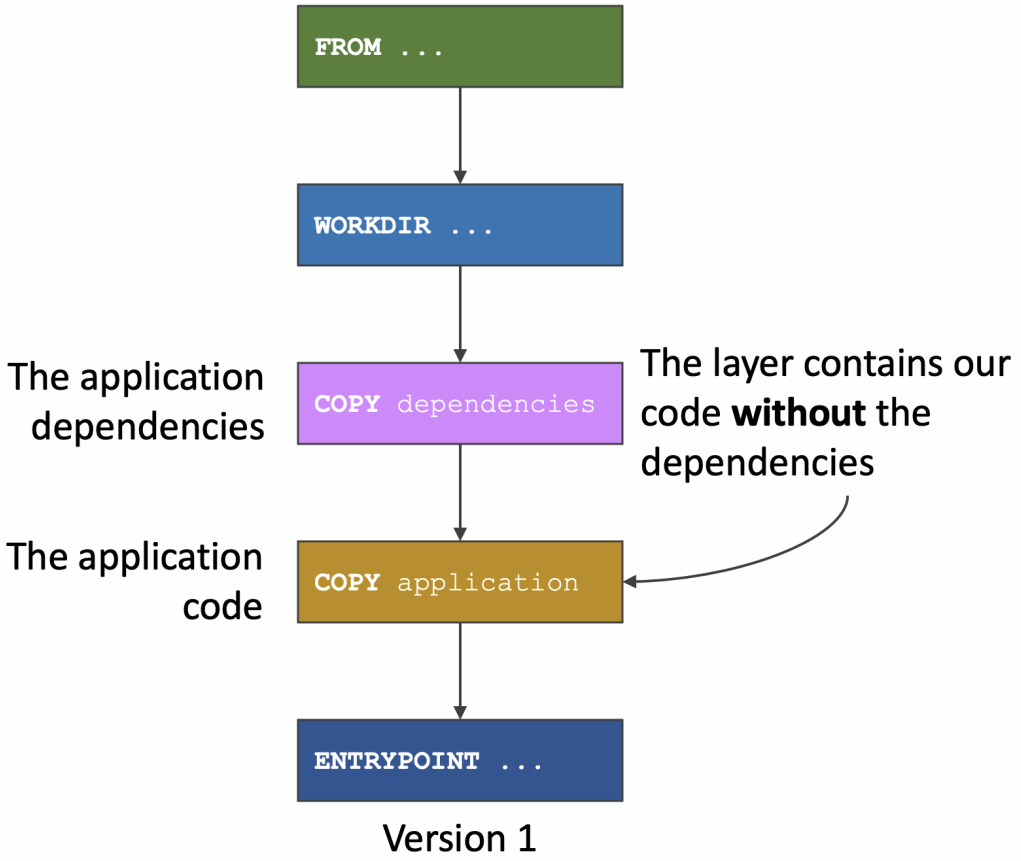


An alternative approach

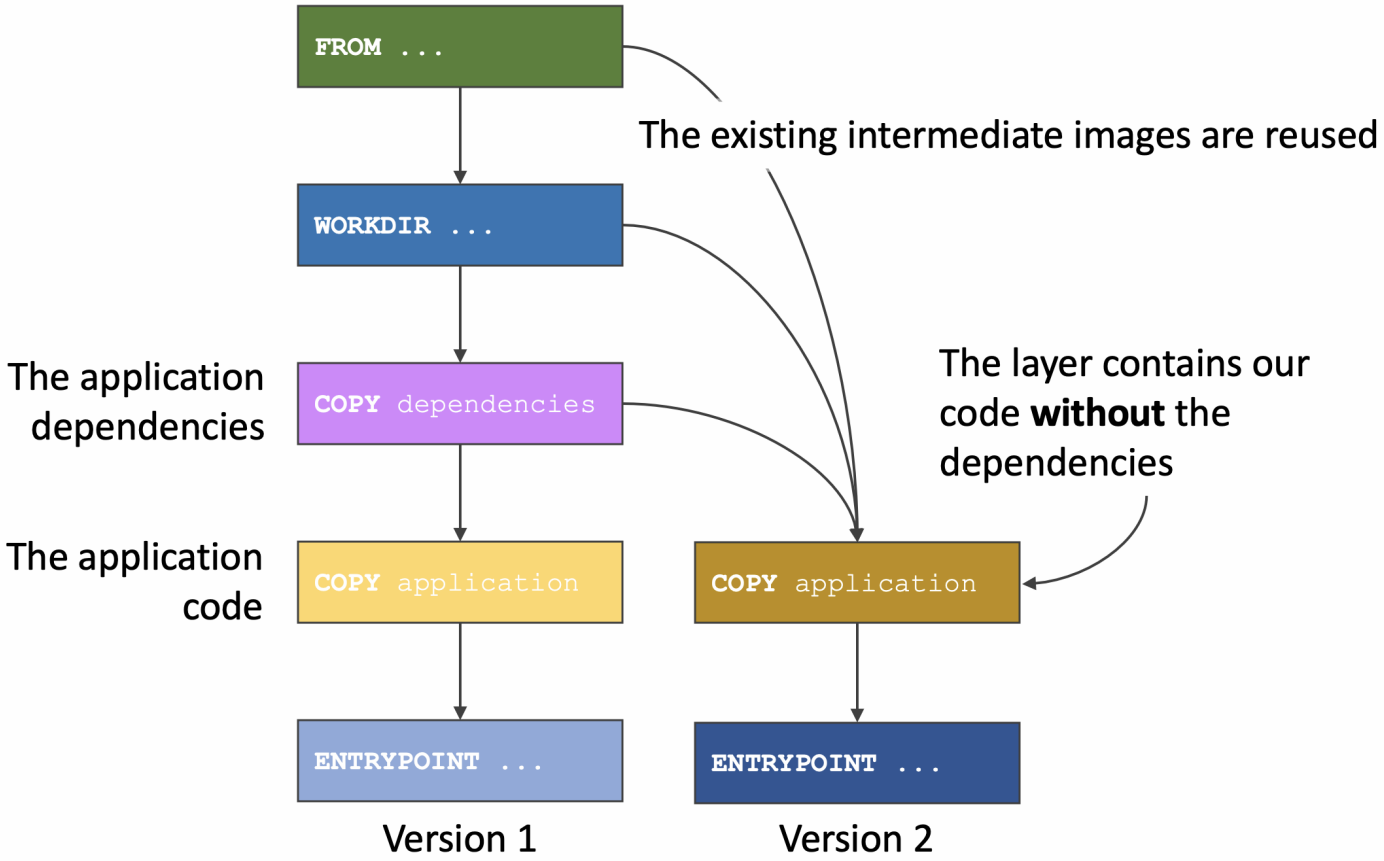
- Some parts of the FatJAR, such as the dependencies, change less frequently than others, yet take up a most of the space

➤ Solution: Separating the dependencies from the code by creating a new layer and taking advantage of Docker layer caching

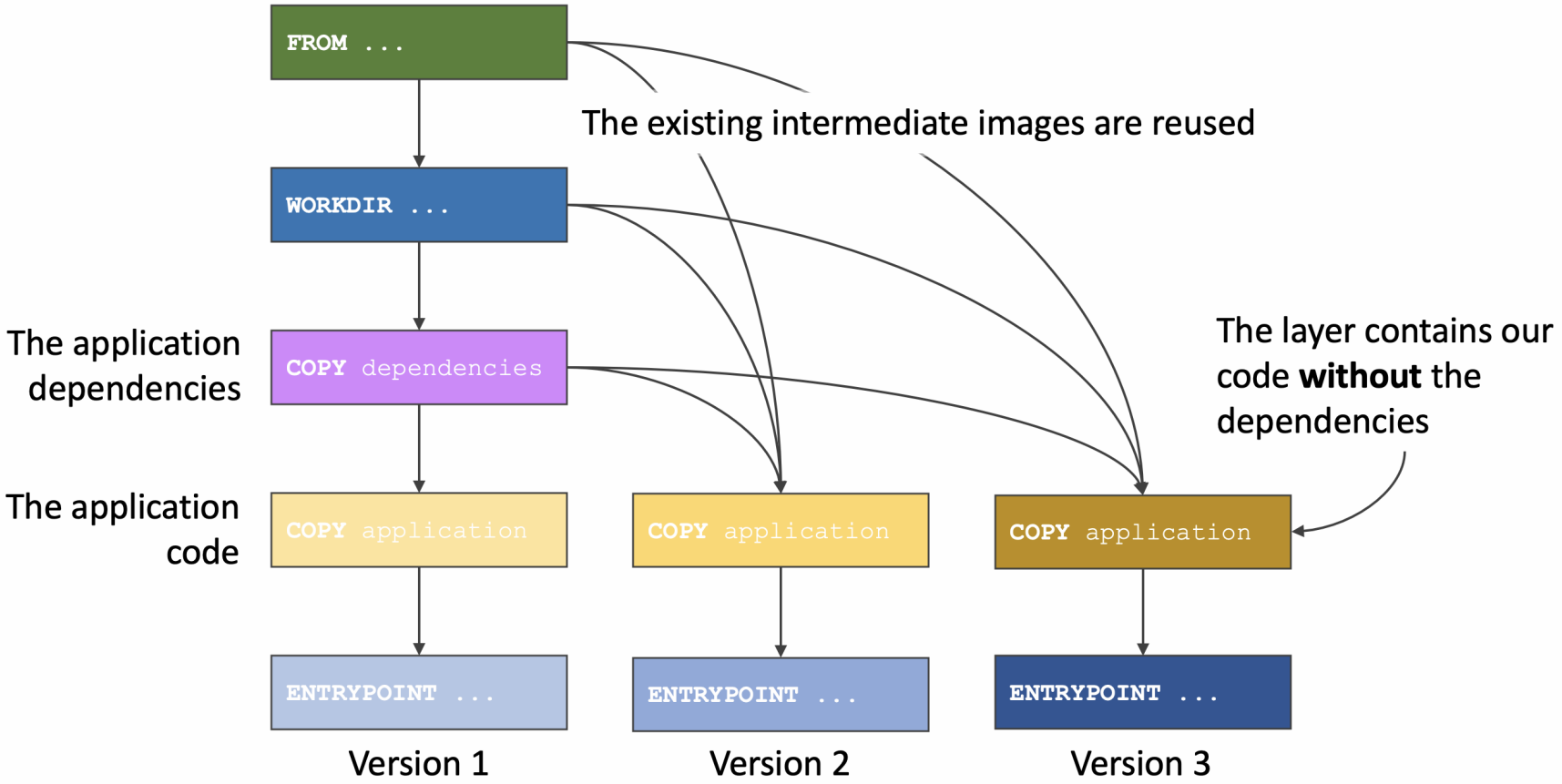
Splitting the FatJAR - Version 1



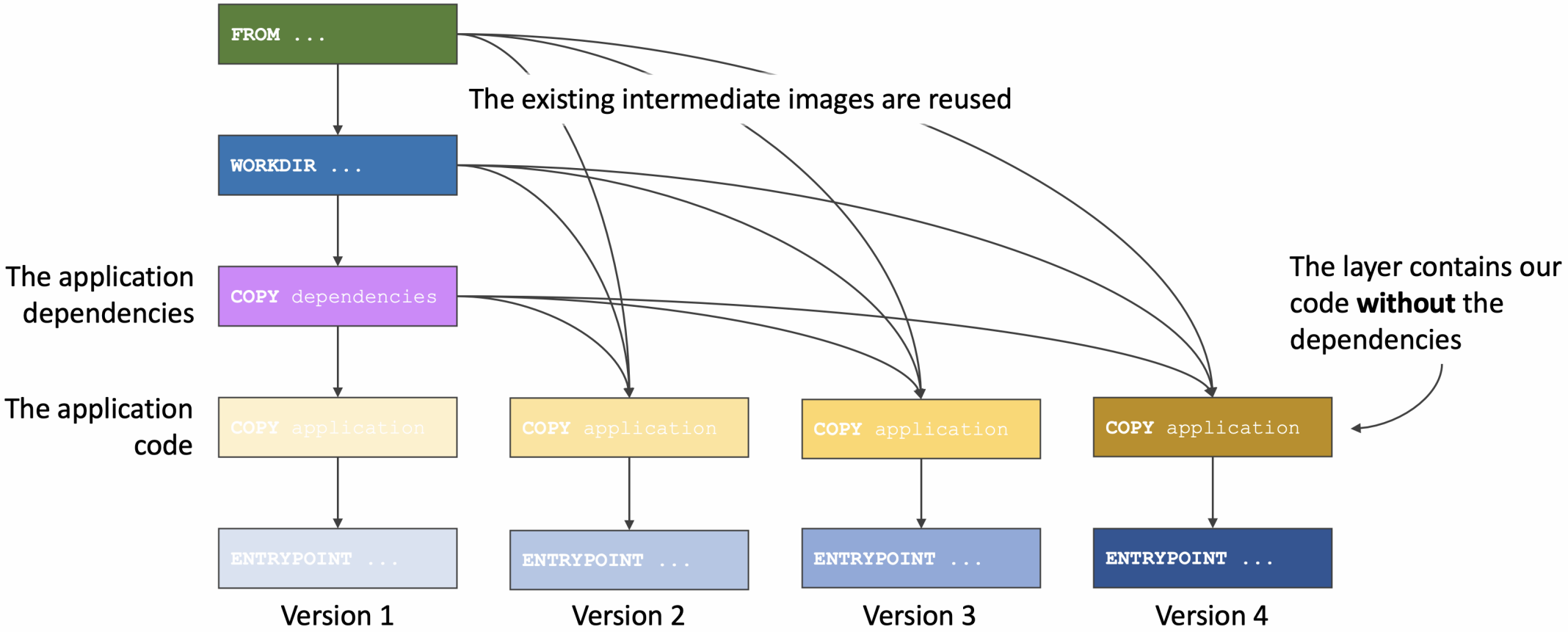
Splitting the FatJAR - Version 2



Splitting the FatJAR - Version 3



Splitting the FatJAR - Version 4



Spring Boot

Spring Boot

- Spring Boot is a very popular framework that promotes productivity

https://spring.io/projects/spring-boot

Layered JAR

- Spring Boot 2.3 comes with a new feature, *Layered JAR*
- This can be enabled by simply adding a configuration to the Gradle `build.gradle` file

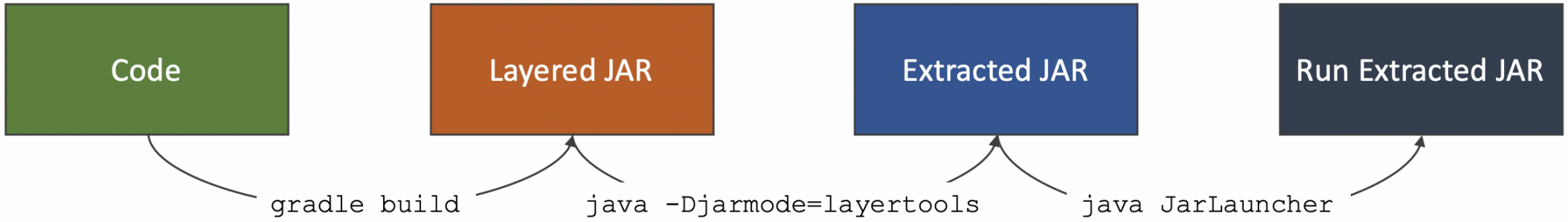
```
bootJar {
  layered()
}
```

  An equivalent plugin is available for Maven too

- Spring Boot 2.4 will have *Layered JAR* enabled by default

How does this work?

- Build the Layered JAR (using *Gradle* or *Maven*)
- Extract the Layered JAR (using *layertool*)
- Run the Extracted JAR (using *JarLauncher*)



Demo

Build layered JAR, extract it and run extracted JAR

- Build layered JAR
- Extract Layered JAR
- Run Extracted JAR

How does this work with Docker?

- We can take advantage of multistage Docker builds

```
FROM adoptopenjdk:8u262-b10-jre-hotspot as builder
WORKDIR /opt/app
COPY ./build/libs/*.jar application.jar
RUN java -Djarmode=layertools -jar application.jar extract

FROM adoptopenjdk:8u262-b10-jre-hotspot
WORKDIR /opt/app
COPY --from=builder /opt/app/dependencies ./
COPY --from=builder /opt/app/spring-boot-loader ./
COPY --from=builder /opt/app/snapshot-dependencies ./
COPY --from=builder /opt/app/application ./
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

Builder stage

- Copy the layered JAR created by Gradle, outside from Docker

```
FROM adoptopenjdk:8u262-b10-jre-hotspot as builder
WORKDIR /opt/app
COPY ./build/libs/*.jar application.jar
```

- Extract the layered JAR

```
RUN java -Djarmode=layertools -jar application.jar extract
```

This command will create four folders in the builder stage, which we will copy in the final stage

Final stage

- Starts with a Java image

```
FROM adoptopenjdk:8u262-b10-jre-hotspot
WORKDIR /opt/app
```

- Copy the extracted folders from the builder stage

```
COPY --from=builder /opt/app/dependencies ./
COPY --from=builder /opt/app/spring-boot-loader ./
COPY --from=builder /opt/app/snapshot-dependencies ./
COPY --from=builder /opt/app/application ./
```

- Run the application using `JarLauncher`

```
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

Demo

Create Docker image using layered JAR and analyse it with dive

- Create Docker image, using multistage and layered JAR
- Inspect with *dive*

Size of Layered JAR

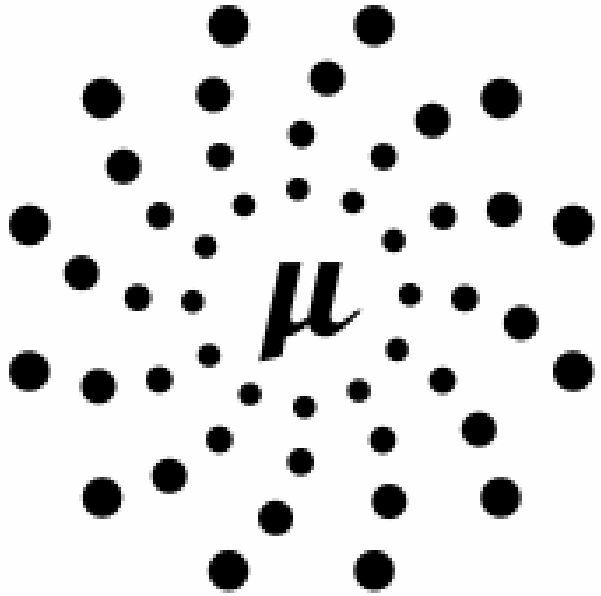
Comparison

- Comparing these two approaches we will find that a layered JAR is far more efficient than a FatJAR in terms of disk space

Beyond Spring Boot

Micronaut

- Micronaut is reflection free alternative framework to Spring Boot



MICRONAUT™

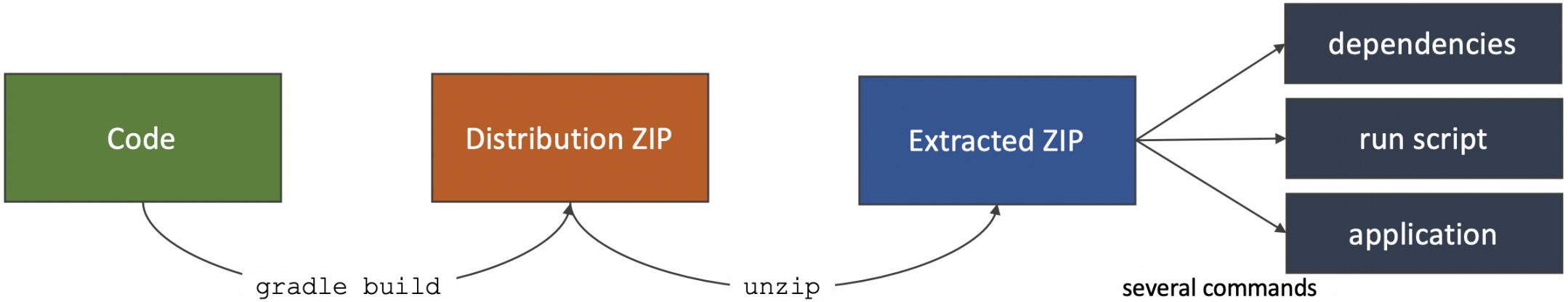
https://micronaut.io/

Lack of tooling

- Spring Boot provides the layered JAR functionality as a Gradle task
- This is not available for all other frameworks
- We can still take advantage of the docker multistage to split our dependencies from the application manually

How will this work?

- Extract the distribution ZIP file, generated by the `distribution` Gradle plugin which is applied by the `application` Gradle plugin
- Move our application thin JAR to its own directory
- Update the run script



Multistage to the rescue!

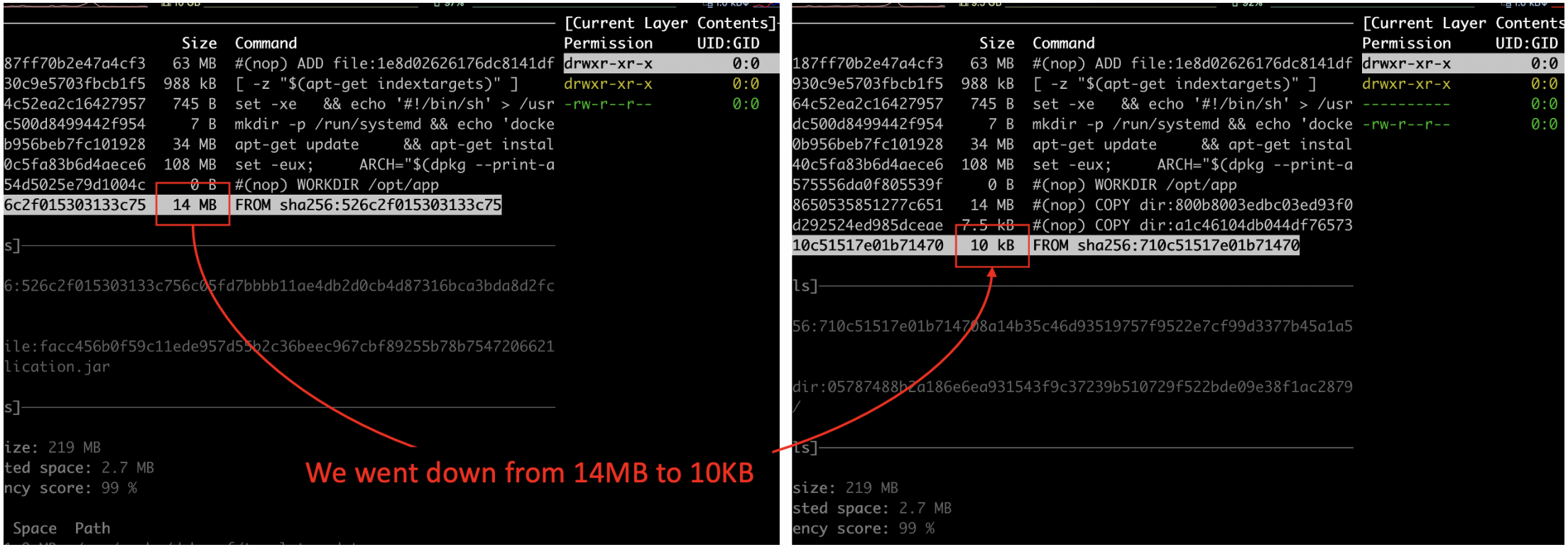
```
FROM alpine:3.12.0 as builder
WORKDIR /opt/app
COPY ./build/distributions/*.zip application.zip
RUN unzip application.zip && rm application.zip \
    && mv * dist && rm dist/bin/*.bat && mv dist/bin/* dist/bin/run.original \
    && sed 's|$APP_HOME/lib/application.jar|$APP_HOME/app/application.jar|g' dist
    && chmod +x dist/bin/run \
    && rm dist/bin/run.original \
    && mkdir dist/app \
    && mv dist/lib/application.jar dist/app/application.jar

FROM adoptopenjdk:8u262-b10-jre-hotspot
ENV APP_HOME /opt/app
WORKDIR ${APP_HOME}
COPY --from=builder /opt/app/dist/lib lib/
COPY --from=builder /opt/app/dist/bin bin/
COPY --from=builder /opt/app/dist/app app/
ENTRYPOINT ["./bin/run"]
```

Demo

Create Docker image using distribution ZIP and analyse it with *dive*

- Go through the `micronaut-layered-jar-1.0.zip` file
- Go through the multistage *Dockerfile*
- Create Docker image, using multistage
- Inspect with *dive*

Comparison


Size Command
87ff70b2e47a4cf3 63 MB #(nop) ADD file:1e8d02626176dc8141df
30c9e5703fbcb1f5 988 kB [-z "$(apt-get indextargets)"]
4c52ea2c16427957 745 B set -xe && echo '#!/bin/sh' > /usr
c500d8499442f954 7 B mkdir -p /run/systemd && echo 'docke
b956beb7fc101928 34 MB apt-get update && apt-get instal
0c5fa83b6d4aece6 108 MB set -eux; ARCH="$(dpkg --print-a
54d5025e79d1004c 0 B #(nop) WORKDIR /opt/app
6c2f015303133c75 14 MB FROM sha256:526c2f015303133c75
[Current Layer Contents]
Permission UID:GID
drwxr-xr-x 0:0
drwxr-xr-x 0:0
-rw-r--r-- 0:0
We went down from 14MB to 10KB
Size Command
187ff70b2e47a4cf3 63 MB #(nop) ADD file:1e8d02626176dc8141df
930c9e5703fbcb1f5 988 kB [-z "$(apt-get indextargets)"]
64c52ea2c16427957 745 B set -xe && echo '#!/bin/sh' > /usr
dc500d8499442f954 7 B mkdir -p /run/systemd && echo 'docke
0b956beb7fc101928 34 MB apt-get update && apt-get instal
40c5fa83b6d4aece6 108 MB set -eux; ARCH="$(dpkg --print-a
575556da0f805539f 0 B #(nop) WORKDIR /opt/app
8650535851277c651 14 MB #(nop) COPY dir:800b8003edbc03ed93f0
d292524ed985dceae 7.5 kB #(nop) COPY dir:a1c46104db044df76573
10c51517e01b71470 10 kB FROM sha256:710c51517e01b71470
[Current Layer Contents
Permission UID:GID
drwxr-xr-x 0:0
drwxr-xr-x 0:0
---------- 0:0
-rw-r--r-- 0:0

Clojure

- Clojure is a functional programming language that runs on the JVM

https://clojure.org/

No to UberJAR!!

- Clojure application are typically packaged as an UberJAR, equivalent to a FatJAR
- Badigeon is a build library based on `tools.deps`, that can be used to create a slim JAR, a JAR file without dependencies

Demo

Create Clojure layered JAR Docker image using Badigeon and analyse it with *dive*

- Go through the project layout
- Run the project locally
- Go through the multistage *Dockerfile*
- Create Docker image, using multistage
- Inspect with *dive*

Thank You

Feedback makes us better

Please send any feedback to: albert.attard@thoughtworks.com or jlang@thoughtworks.com


WE ARE HIRING
26 YEARS EXPERIENCE
43 OFFICES IN 14 COUNTRIES
THOUGHT LEADER IN AGILE SOFTWARE DEVELOPMENT AND CONTINUOUS DELIVERY
+7.000 THOUGHTWORKERS WORLDWIDE
+300 THOUGHTWORKERS IN GERMANY
de-recruiting@thoughtworks.com