A container image builder
for Java applications

Build containers faster with Jib

Google Cloud

github.com/GoogleContainerTools/jib

Our Team

Cloud Tools for Java

Appu Goundan
@loosebazooka

Qingyang “Q” Chen
@coollog

Containers

“Write once, run anywhere”

Building a Java container

github.com/GoogleContainerTools/jib

Google Cloud

Me

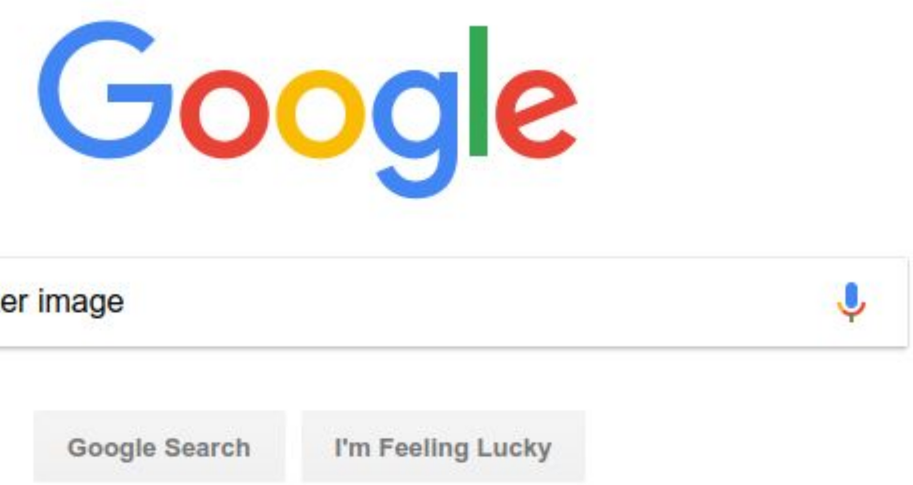
Java Developer

Building website for pet clinic

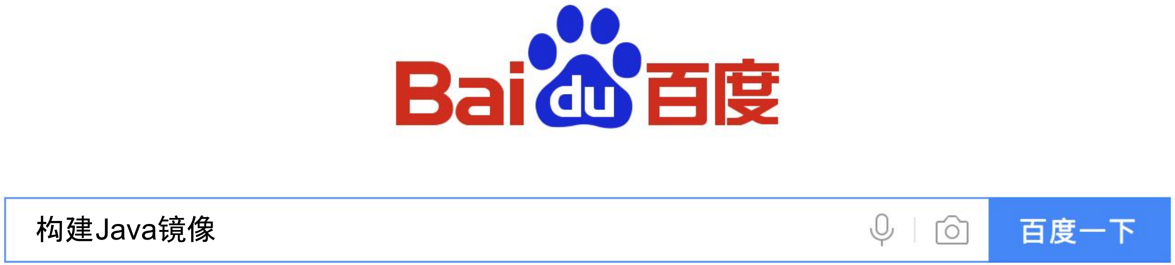
Wants to containerize the backend

Wants container on registry ilovejava.io/petclinic-app

Google Cloud

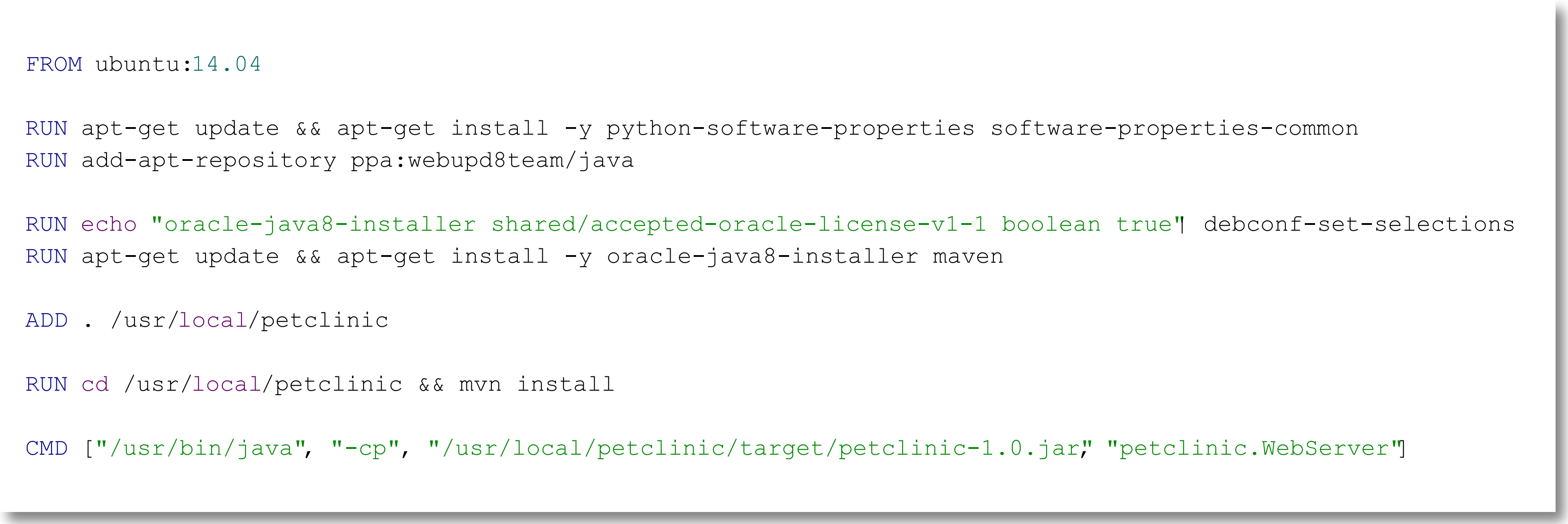


Build Java Docker image



构建Java镜像

So I read some tutorials



```
FROM ubuntu:14.04

RUN apt-get update && apt-get install -y python-software-properties software-properties-common
RUN add-apt-repository ppa:webupd8team/java

RUN echo "oracle-java8-installer shared/accepted-oracle-license-v1-1 boolean true"| debconf-set-selections
RUN apt-get update && apt-get install -y oracle-java8-installer maven

ADD . /usr/local/petclinic

RUN cd /usr/local/petclinic && mvn install

CMD ["/usr/bin/java", "-cp", "/usr/local/petclinic/target/petclinic-1.0.jar", "petclinic.WebServer"]
```

So I read some more tutorials

```
FROM openjdk:8
COPY target/petclinic-*.jar /app.jar
ENTRYPOINT java -jar /app.jar
```

```
FROM openjdk:8
COPY target/petclinic-*.jar /app.jar
ENTRYPOINT java -jar /app.jar
```

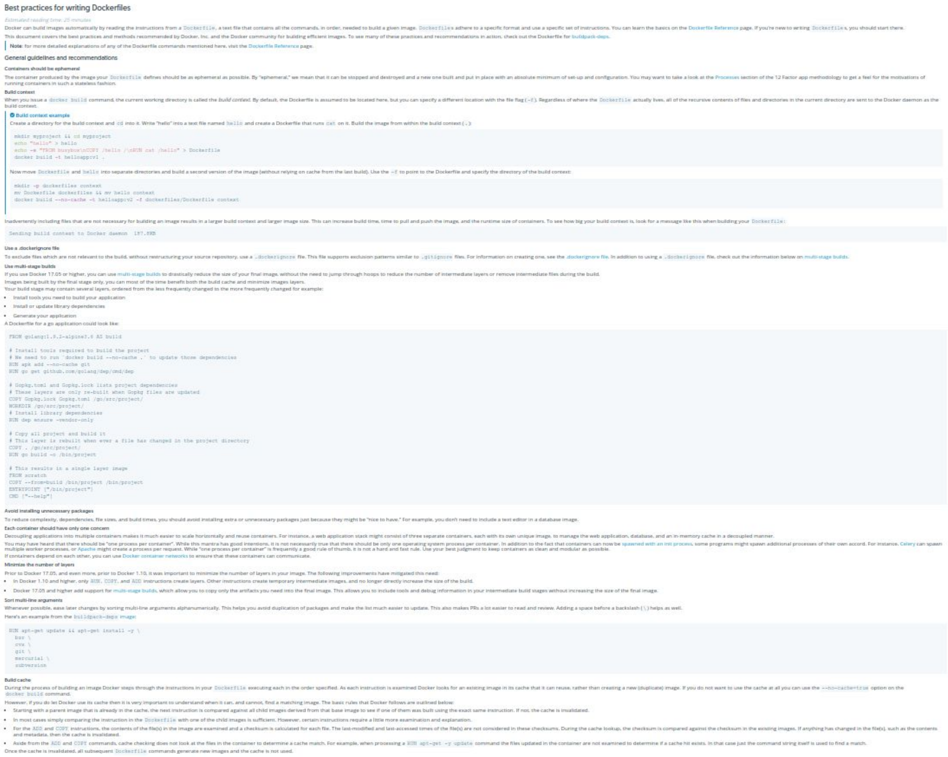
Problem: `openjdk:8` is 284MB

Some more searching

```
FROM openjdk:8-jre-alpine
COPY target/petclinic-*.jar /app.jar
ENTRYPOINT java -jar /app.jar
```

82 MB

Dockerfile best practices



Best practices for writing Dockerfiles

Estimated reading time: 25 minutes

Docker can build images automatically by reading the instructions from a `Dockerfile`, a text file that contains all the commands, in order, needed to build a given image. `Dockerfile`s adhere to a specific format and use a specific set of instructions. You can learn the basics on the Dockerfile Reference page. If you're new to writing `Dockerfile`s, you should start there.

This document covers the best practices and methods recommended by Docker, Inc. and the Docker community for building efficient images. To see many of these practices and recommendations in action, check out the Dockerfile for buildpack-deps.

> **Note:** for more detailed explanations of any of the Dockerfile commands mentioned here, visit the Dockerfile Reference page.

General guidelines and recommendations

Containers should be ephemeral

The container produced by the image your `Dockerfile` defines should be as ephemeral as possible. By "ephemeral," we mean that it can be stopped and destroyed and a new one built and put in place with an absolute minimum of set-up and configuration. You may want to take a look at the Processes section of the 12 Factor app methodology to get a feel for the motivations of running containers in such a stateless fashion.

Build context

When you issue a `docker build` command, the current working directory is called the *build context*. By default, the Dockerfile is assumed to be located here, but you can specify a different location with the file flag (`-f`). Regardless of where the `Dockerfile` actually lives, all of the recursive contents of files and directories in the current directory are sent to the Docker daemon as the build context.

> Build context example
>
> Create a directory for the build context and `cd` into it. Write "hello" into a text file named `hello` and create a Dockerfile that runs `cat` on it. Build the image from within the build context (`.`):
>
> ```
> mkdir myproject && cd myproject
> echo "hello" > hello
> echo -e "FROM busybox\nCOPY /hello /\nRUN cat /hello" > Dockerfile
> docker build -t helloapp:v1 .
> ```
>
> Now move `Dockerfile` and `hello` into separate directories and build a second version of the image (without relying on cache from the last build). Use the `-f` to point to the Dockerfile and specify the directory of the build context:
>
> ```
> mkdir -p dockerfiles context
> mv Dockerfile dockerfiles && mv hello context
> docker build --no-cache -t helloapp:v2 -f dockerfiles/Dockerfile context
> ```

Inadvertently including files that are not necessary for building an image results in a larger build context and larger image size. This can increase build time, time to pull and push the image, and the runtime size of containers. To see how big your build context is, look for a message like this when building your `Dockerfile`:

```
Sending build context to Docker daemon  187.8MB
```

Use a .dockerignore file

To exclude files which are not relevant to the build, without restructuring your source repository, use a `.dockerignore` file. This file supports exclusion patterns similar to `.gitignore` files. For information on creating one, see the .dockerignore file. In addition to using a `.dockerignore` file, check out the information below on multi-stage builds.

Use multi-stage builds

If you use Docker 17.05 or higher, you can use multi-stage builds to drastically reduce the size of your final image, without the need to jump through hoops to reduce the number of intermediate layers or remove intermediate files during the build.

Images being built by the final stage only, you can most of the time benefit both the build cache and minimize images layers.

Your build stage may contain several layers, ordered from the less frequently changed to the more frequently changed for example:

- Install tools you need to build your application
- Install or update library dependencies
- Generate your application

A Dockerfile for a go application could look like:

```
FROM golang:1.9.2-alpine3.6 AS build

# Install tools required to build the project
# We need to run `docker build --no-cache .` to update those dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# Gopkg.toml and Gopkg.lock lists project dependencies
# These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# Install library dependencies
RUN dep ensure -vendor-only

# Copy all project and build it
# This layer is rebuilt when ever a file has changed in the project directory
COPY . /go/src/project/
RUN go build -o /bin/project

# This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

Avoid installing unnecessary packages

To reduce complexity, dependencies, file sizes, and build times, you should avoid installing extra or unnecessary packages just because they might be "nice to have." For example, you don't need to include a text editor in a database image.

Each container should have only one concern

Decoupling applications into multiple containers makes it much easier to scale horizontally and reuse containers. For instance, a web application stack might consist of three separate containers, each with its own unique image, to manage the web application, database, and an in-memory cache in a decoupled manner.

You may have heard that there should be "one process per container". While this mantra has good intentions, it is not necessarily true that there should be only one operating system process per container. In addition to the fact that containers can now be spawned with an init process, some programs might spawn additional processes of their own accord. For instance, Celery can spawn multiple worker processes, or Apache might create a process per request. While "one process per container" is frequently a good rule of thumb, it is not a hard and fast rule. Use your best judgment to keep containers as clean and modular as possible.

If containers depend on each other, you can use Docker container networks to ensure that these containers can communicate.

Minimize the number of layers

Prior to Docker 17.05, and even more, prior to Docker 1.10, it was important to minimize the number of layers in your image. The following improvements have mitigated this need:

- In Docker 1.10 and higher, only `RUN`, `COPY`, and `ADD` instructions create layers. Other instructions create temporary intermediate images, and no longer directly increase the size of the build.
- Docker 17.05 and higher add support for multi-stage builds, which allow you to copy only the artifacts you need into the final image. This allows you to include tools and debug information in your intermediate build stages without increasing the size of the final image.

Sort multi-line arguments

Whenever possible, ease later changes by sorting multi-line arguments alphanumerically. This helps you avoid duplication of packages and make the list much easier to update. This also makes PRs a lot easier to read and review. Adding a space before a backslash (`\`) helps as well.

Here's an example from the `buildpack-deps` image:

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion
```

Build cache

During the process of building an image Docker steps through the instructions in your `Dockerfile` executing each in the order specified. As each instruction is examined Docker looks for an existing image in its cache that it can reuse, rather than creating a new (duplicate) image. If you do not want to use the cache at all you can use the `--no-cache=true` option on the `docker build` command.

However, if you do let Docker use its cache then it is very important to understand when it can, and cannot, find a matching image. The basic rules that Docker follows are outlined below:

- Starting with a parent image that is already in the cache, the next instruction is compared against all child images derived from that base image to see if one of them was built using the exact same instruction. If not, the cache is invalidated.
- In most cases simply comparing the instruction in the `Dockerfile` with one of the child images is sufficient. However, certain instructions require a little more examination and explanation.
- For the `ADD` and `COPY` instructions, the contents of the file(s) in the image are examined and a checksum is calculated for each file. The last-modified and last-accessed times of the file(s) are not considered in these checksums. During the cache lookup, the checksum is compared against the checksum in the existing images. If anything has changed in the file(s), such as the contents and metadata, then the cache is invalidated.
- Aside from the `ADD` and `COPY` commands, cache checking does not look at the files in the container to determine a cache match. For example, when processing a `RUN apt-get -y update` command the files updated in the container are not examined to determine if a cache hit exists. In that case just the command string itself is used to find a match.

Once the cache is invalidated, all subsequent `Dockerfile` commands generate new images and the cache is not used.

docs.docker.com/develop/develop-images/dockerfile_best-practices

.dockerignore

```
**
!target/petclinic-*.jar
```

Some more tutorials later

```
$ mvn dependencies:copy-dependencies to target/dependencies/
```

```
FROM openjdk:8-jre-alpine
COPY target/dependencies /app/dependencies
COPY target/classes /app/classes
ENTRYPOINT java -cp /app/dependencies/*:/app/classes petclinic.WebServer
```

Some more searching

```xml
...
<build>
  <plugins>
    <plugin>
      <groupId>com.spotify</groupId>
      <artifactId>dockerfile-maven-plugin</artifactId>
      <version>1.4.8</version>
      <configuration>
        <repository>ilovejava.io/petclinic-app</repository>
        <tag>${project.version}</tag>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

What did we do?

1. Write first Dockerfile
2. Reduce image size
3. Don’t run installs
4. Use better base image
5. Write .dockerignore
6. Improve incremental speed
7. Switch to use a Maven plugin

Download and install Docker

What did we do?

1. Write first Dockerfile
2. Reduce image size
3. Don’t run installs
4. Use better base image
5. Write .dockerignore
6. Improve incremental speed
7. Switch to use a Maven plugin

Download and install Docker

What did we do?

1. Write first Dockerfile
2. Reduce image size
3. Don’t run installs
4. Use better base image
5. Write .dockerignore
6. Improve incremental speed
7. Switch to use a Maven plugin

Order of layers to optimize for cache hits

Download and install Docker

What did we do?

1. Write first Dockerfile
2. Reduce image size
3. Don’t run installs
4. Use better base image
5. Write .dockerignore
6. Improve incremental speed
7. Switch to use a Maven plugin

Order of layers to optimize for cache hits

Use of multi-stage builds

Understanding Docker cache mechanism and quirks

Download and install Docker

What did we do?

1. Write first Dockerfile
2. Reduce image size
3. Don’t run installs
4. Use better base image
5. Write .dockerignore
6. Improve incremental speed
7. Switch to use a Maven plugin

Order of layers to optimize for cache hits

Use of multi-stage builds

Understanding Docker cache mechanism and quirks

Download and install Docker

What did we do?

1. Write first Dockerfile
2. Reduce image size
3. Don’t run installs
4. Use better base image
5. Write .dockerignore
6. Improve incremental speed
7. Switch to use a Maven plugin

Order of layers to optimize for cache hits

Use of multi-stage builds

Have elevated privileges to run Docker daemon

Understanding Docker cache mechanism and quirks

Download and install Docker

What did we do?

1. Write first Dockerfile
2. Reduce image size
3. Don’t run installs
4. Use better base image
5. Write .dockerignore
6. Improve incremental speed
7. Switch to use a Maven plugin

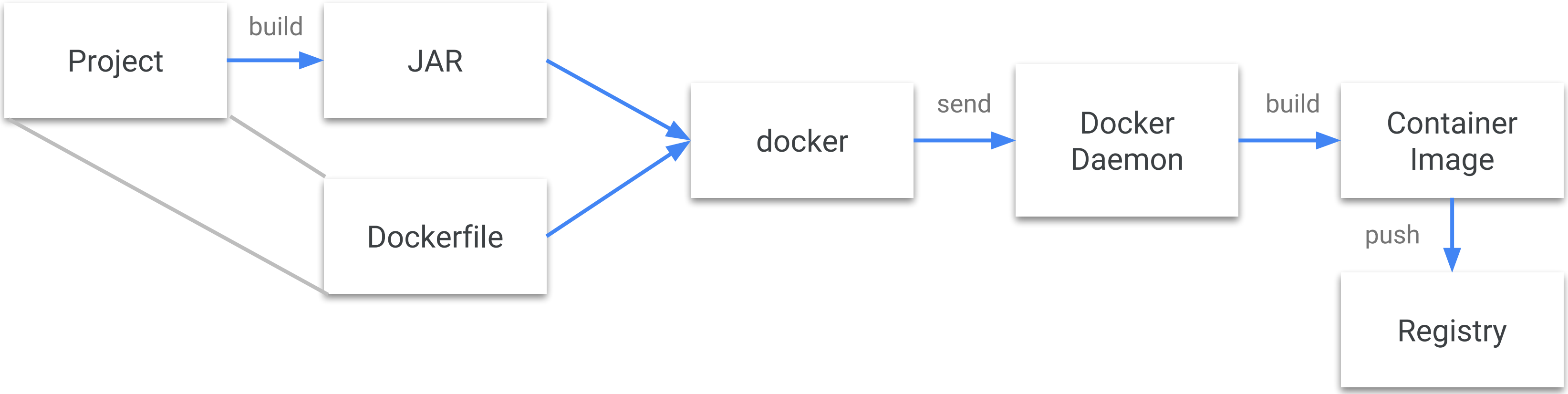
Order of layers to optimize for cache hits

Use of multi-stage builds

Have elevated privileges to run Docker daemon

saturnism.me/talk/docker-tips-and-tricks

Containerizing with Docker



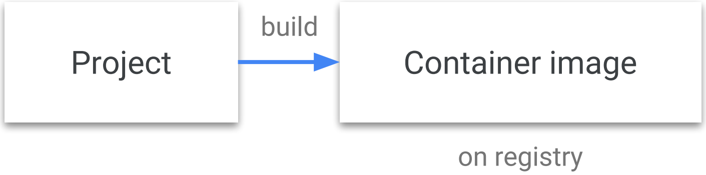
I’m a Java developer, I don’t want to have to care about Dockerfiles

Some Java Developer
Somewhere

github.com/GoogleContainerTools/jib

Google Cloud

Containerizing, simplified


Project

build

Container image

on registry

Jib

Containerize your Java application.

Steps:

github.com/GoogleContainerTools/jib

Google Cloud

Jib

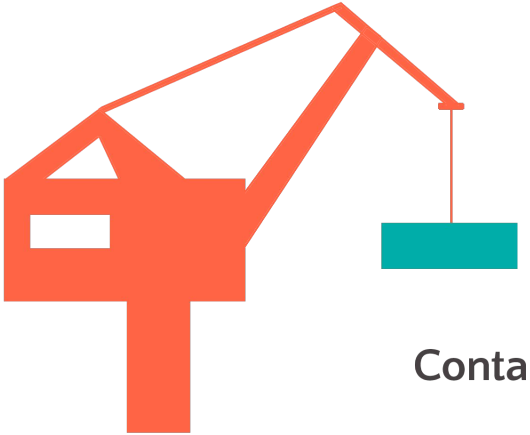
Containerize your Java application.

Steps:

1. Apply the plugin.

github.com/GoogleContainerTools/jib

Google Cloud



Jib

Containerize your Java application.

Steps:

1. Apply the plugin.
2. `mvn jib:build`

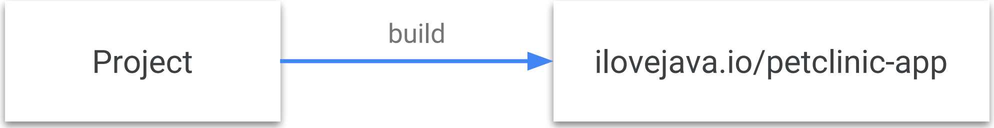
   (or `gradle jib`)

github.com/GoogleContainerTools/jib

Google Cloud

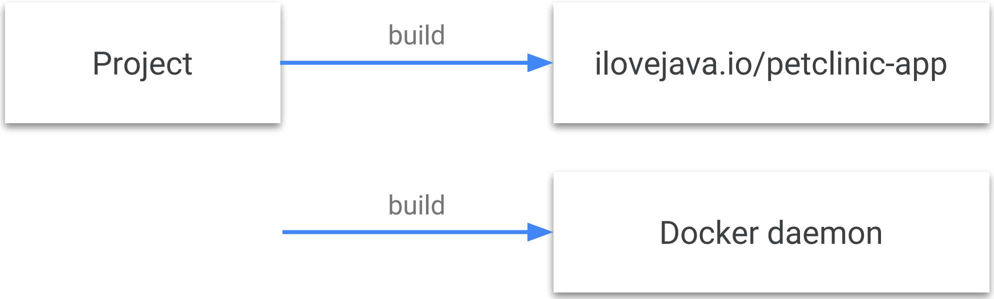
Demo

```
$ git clone https://github.com/spring-projects/spring-petclinic && cd spring-petclinic
$ ./mvnw compile jib:build -Dimage=coollog/petclinic
```


Project

build

ilovejava.io/petclinic-app



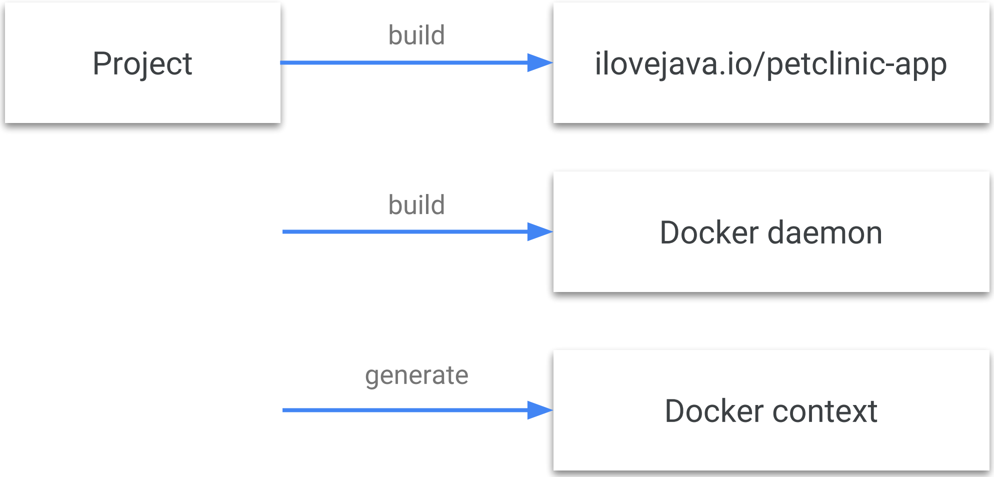
Project

build

ilovejava.io/petclinic-app

build

Docker daemon



Project

build

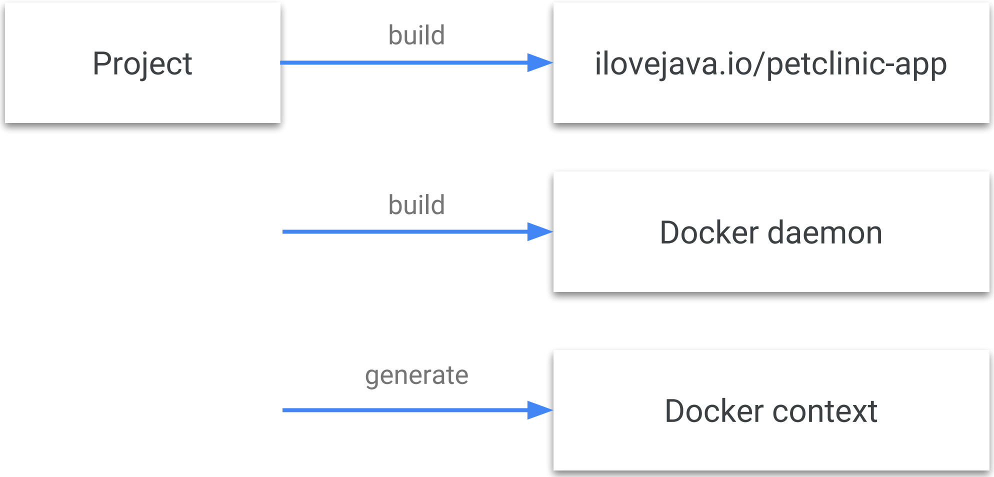
ilovejava.io/petclinic-app

build

Docker daemon

generate

Docker context



Extended Configuration

JVM flags credentials labels environment variables extra files ...

Demo

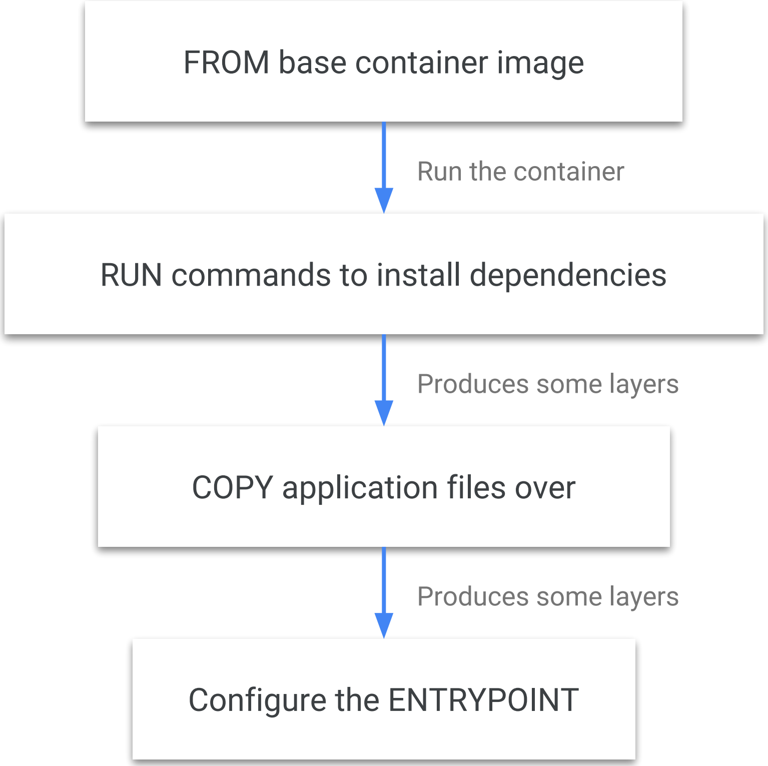
```
$ git clone https://github.com/coollog/micronaut-jib && cd micronaut-jib
$ ./gradlew jibDockerBuild
$ docker run -p 8080:8080 micronaut-jib:0.1
```

A “compiler” for containers

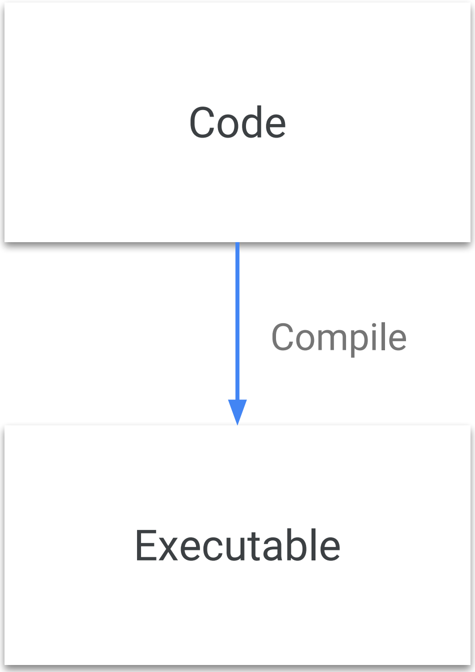
github.com/GoogleContainerTools/jib

Google Cloud

Dockerfile “script”


FROM base container image
Run the container
RUN commands to install dependencies
Produces some layers
COPY application files over
Produces some layers
Configure the ENTRYPOINT

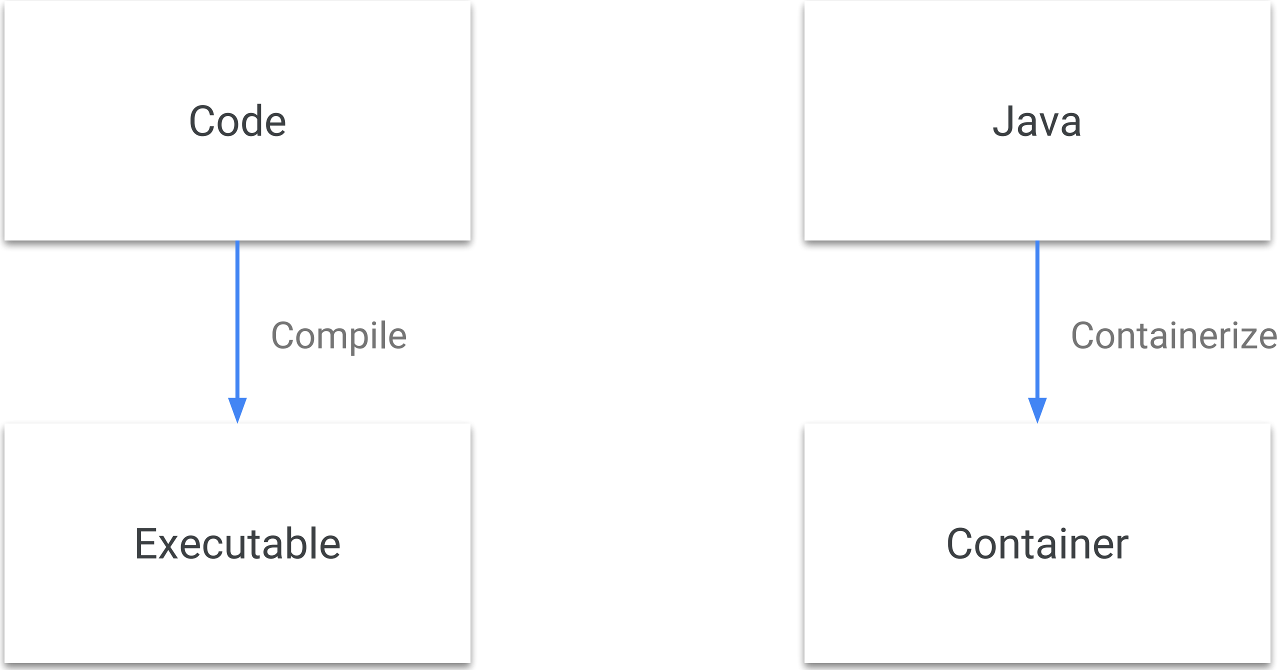
Compiler + Containerizer



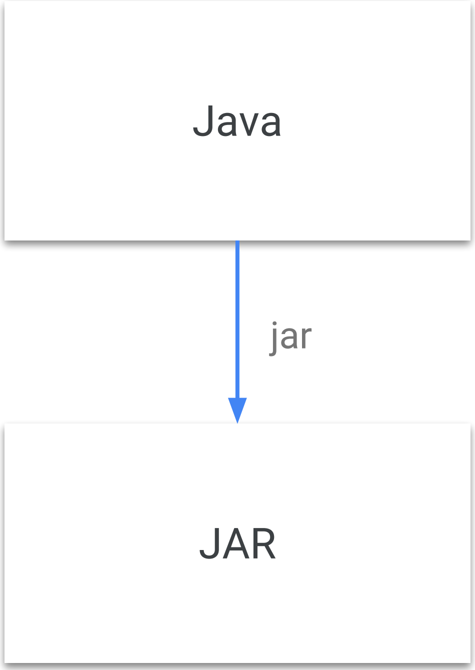
Code

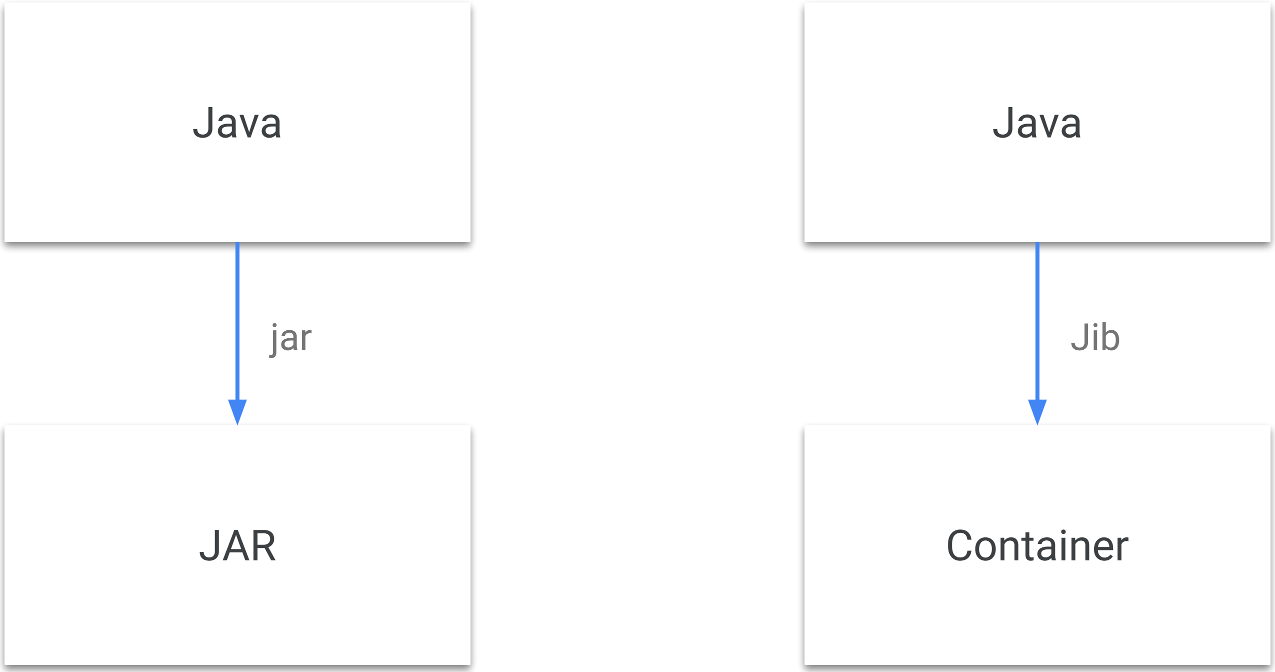
Compile

Executable


Code
Compile
Executable
Java
Containerize
Container

Containers are the executables of the cloud.


Java
jar
JAR



Java

jar

JAR

Java

Jib

Container

How Jib Works

Google Cloud

What benefits do we get from Jib

Pure Java

Speed

Reproducibility

Google Cloud

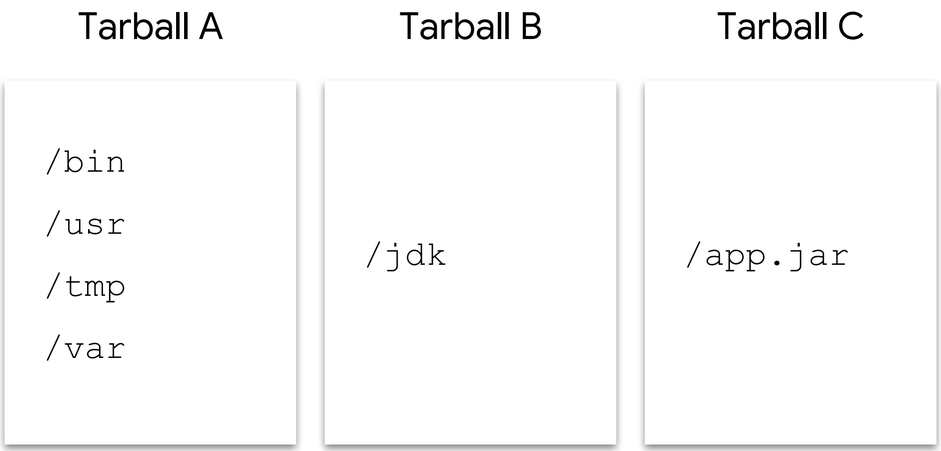
github.com/GoogleContainerTools/jib

Pure Java

A container image is a directory of files

Docker Image Format

Tarballs that compose into a single filesystem



Tarball A

/bin

/usr

/tmp

/var

Tarball B

/jdk

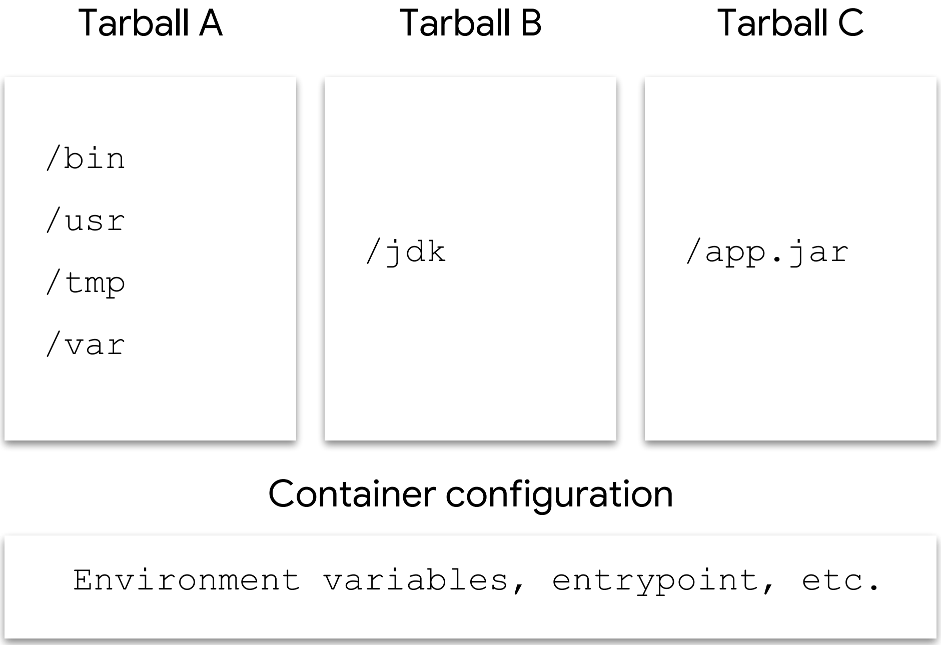
Tarball C

/app.jar

Docker Image Format

Tarballs that compose into a single filesystem

And a container configuration



Tarball A

/bin

/usr

/tmp

/var

Tarball B

/jdk

Tarball C

/app.jar

Container configuration

Environment variables, entrypoint, etc.

```
{
  "architecture" : "amd64",
  "os": "linux",
  "config": {
    "Env": [],
    "Entrypoint": [
      "java",
      "-cp",
      "/app/libs/*:/app/resources/:/app/classes/" ,
      "com.test.HelloWorld"
    ]
  },
  "rootfs": {
    "type": "layers",
    "diff_ids": [
      "sha256:46e7865bff73b5a0c610bf9f20c91dfafa2518ace8703faaffff551a4773b947"  ,
      "sha256:6189abe095d53c1c9f2bfc8f50128ee876b9a5d10f9eda1564e5f5357d6ffe61"  ,
      "sha256:e8292403028e724f0c7686ede4cd89180faa85aeb63cd0e7d560e8a459d83afe"  ,
      "sha256:ff7666ffd3d45500f4af71f091a603413acb04d028ba03a6698f63819d246cb5"  ,
      "sha256:db22fdca5c6344265d841ec106e683fb39914f356fb1d8e69accb466a396dc62"  ,
      "sha256:9aa41c013edd2a6311dcdd4d26129b01b3ba0b08c8adb51759c63501a69d27f5"
    ]
  }
}
```

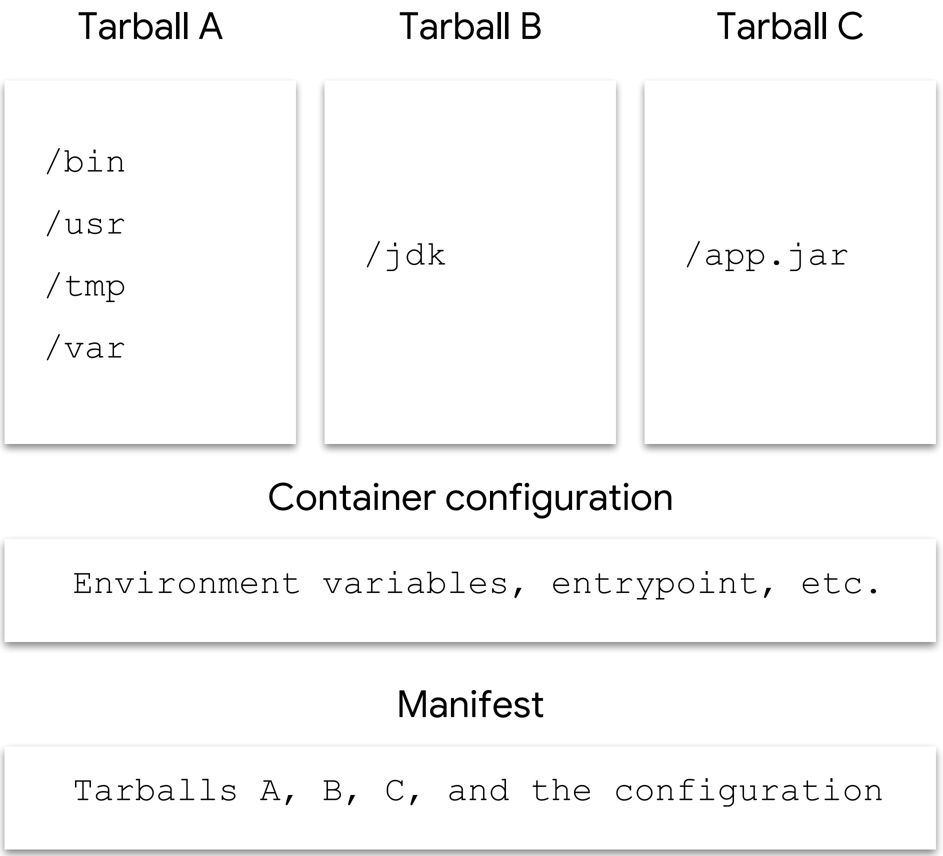
checksums ↓

Docker Image Format

Tarballs that compose into a single filesystem

And a container configuration

And a manifest



Tarball A

```
/bin
/usr
/tmp
/var
```

Tarball B

```
/jdk
```

Tarball C

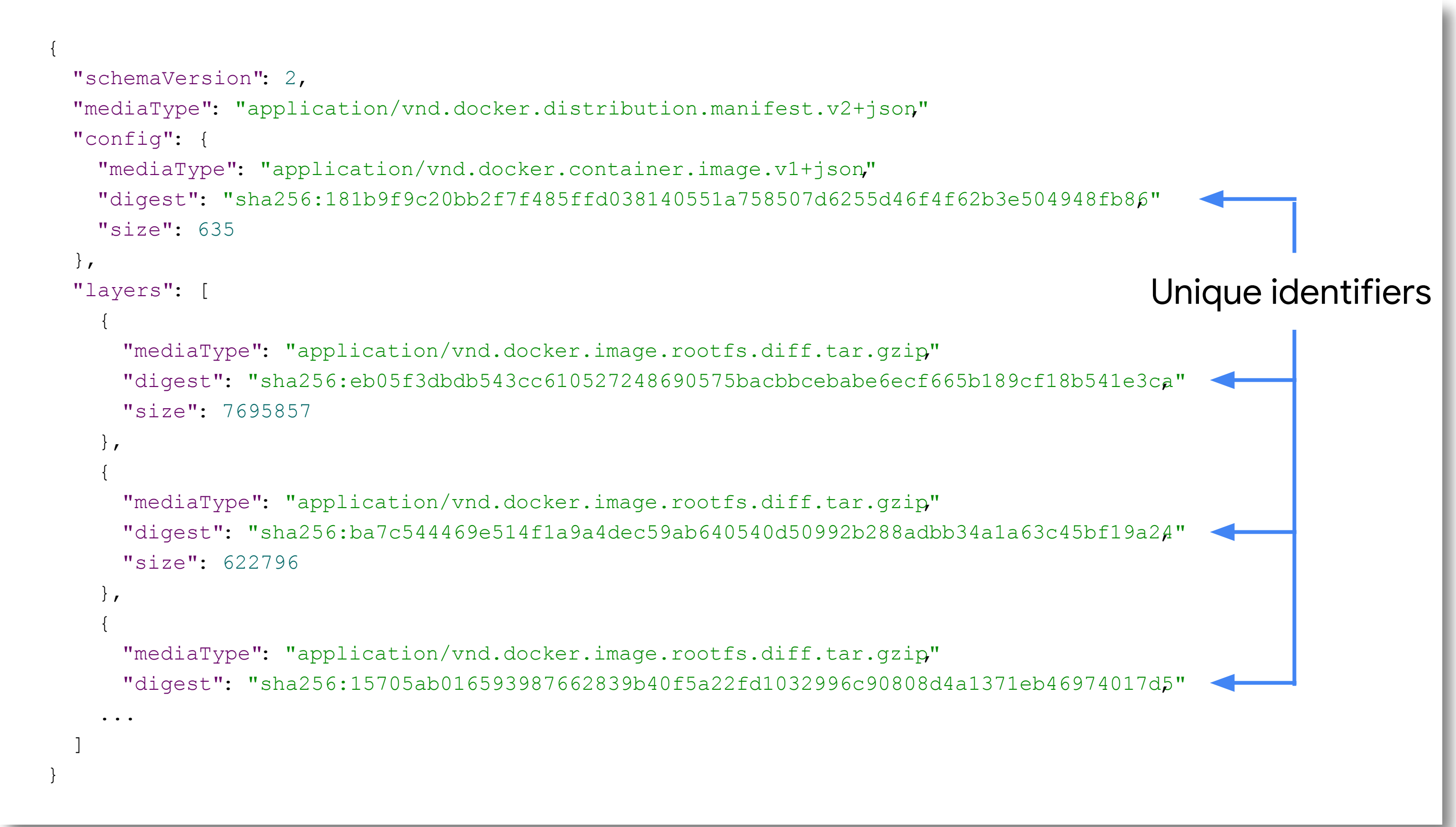
```
/app.jar
```

Container configuration

```
Environment variables, entrypoint, etc.
```

Manifest

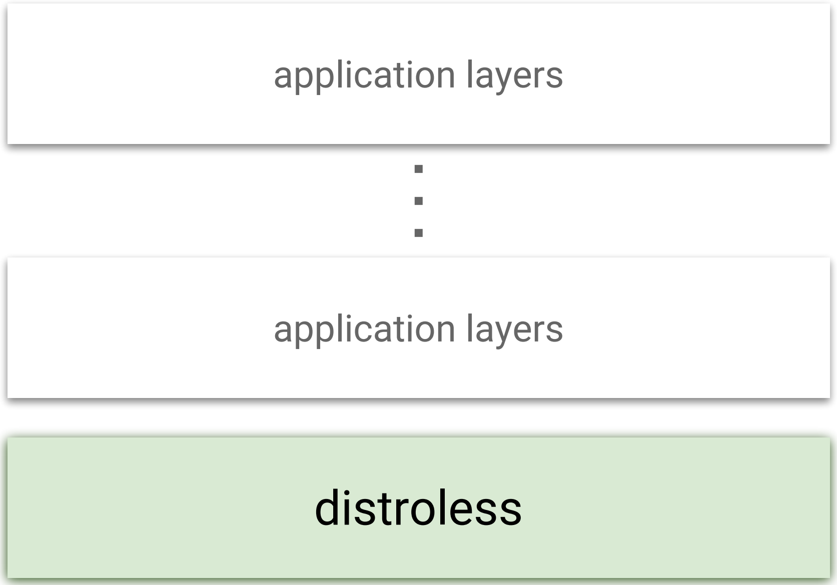
```
Tarballs A, B, C, and the configuration
```



```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
  "config": {
    "mediaType": "application/vnd.docker.container.image.v1+json",
    "digest": "sha256:181b9f9c20bb2f7f485ffd038140551a758507d6255d46f4f62b3e504948fb86",
    "size": 635
  },
  "layers": [
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "digest": "sha256:eb05f3dbdb543cc610527248690575bacbbcebabe6ecf665b189cf18b541e3ca",
      "size": 7695857
    },
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "digest": "sha256:ba7c544469e514f1a9a4dec59ab640540d50992b288adbb34a1a63c45bf19a24",
      "size": 622796
    },
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "digest": "sha256:15705ab016593987662839b40f5a22fd1032996c90808d4a1371eb46974017d5",
    ...
  ]
}
```

Unique identifiers

Jib image



application layers

⋮

application layers

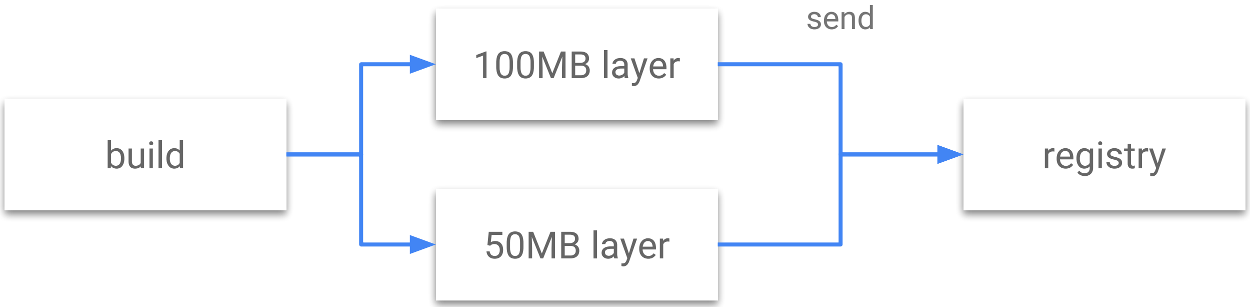
distroless

github.com/GoogleContainerTools/distroless

Speed

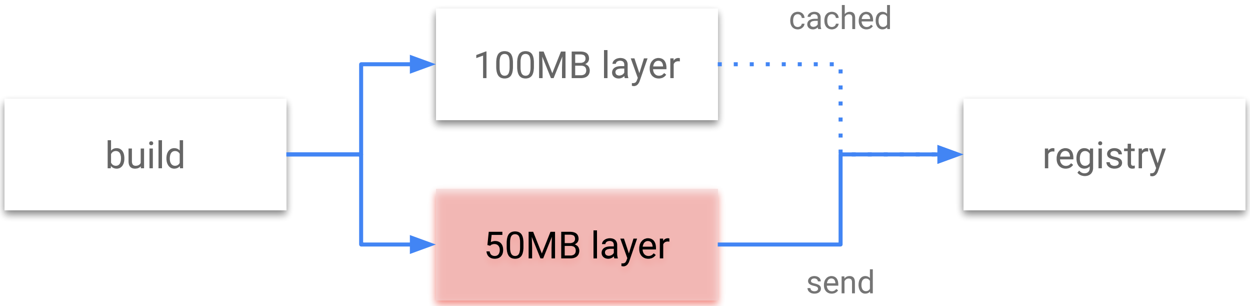
Docker registry

Set of layers, container configurations, and manifests



Docker registry

Set of layers, container configurations, and manifests

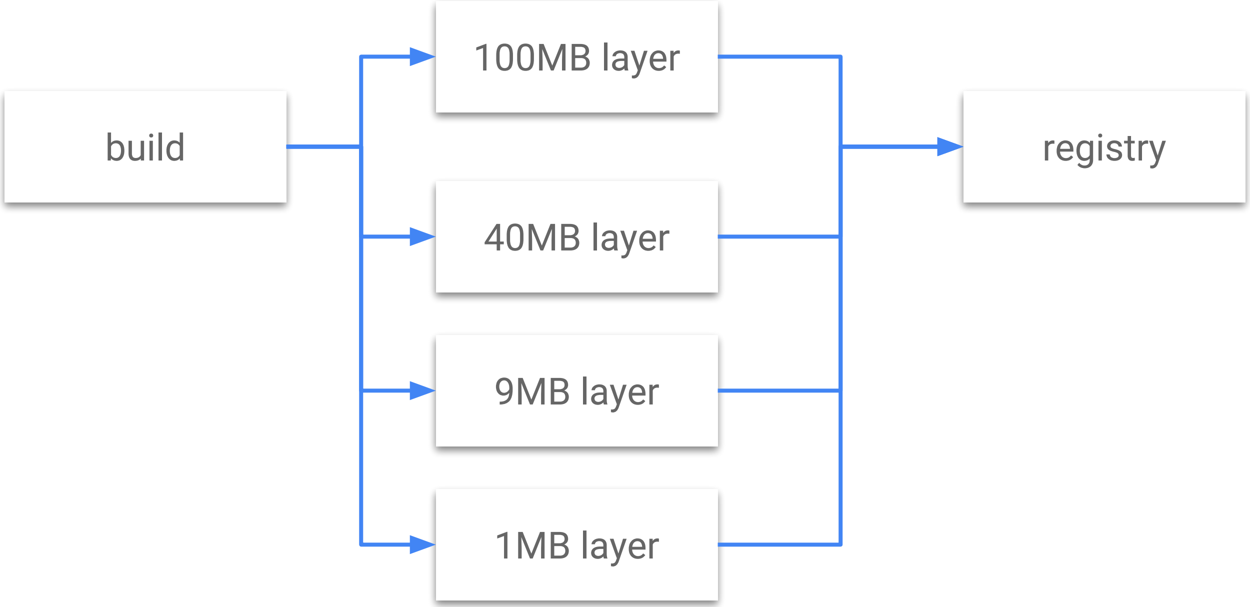


Google Cloud

github.com/GoogleContainerTools/jib

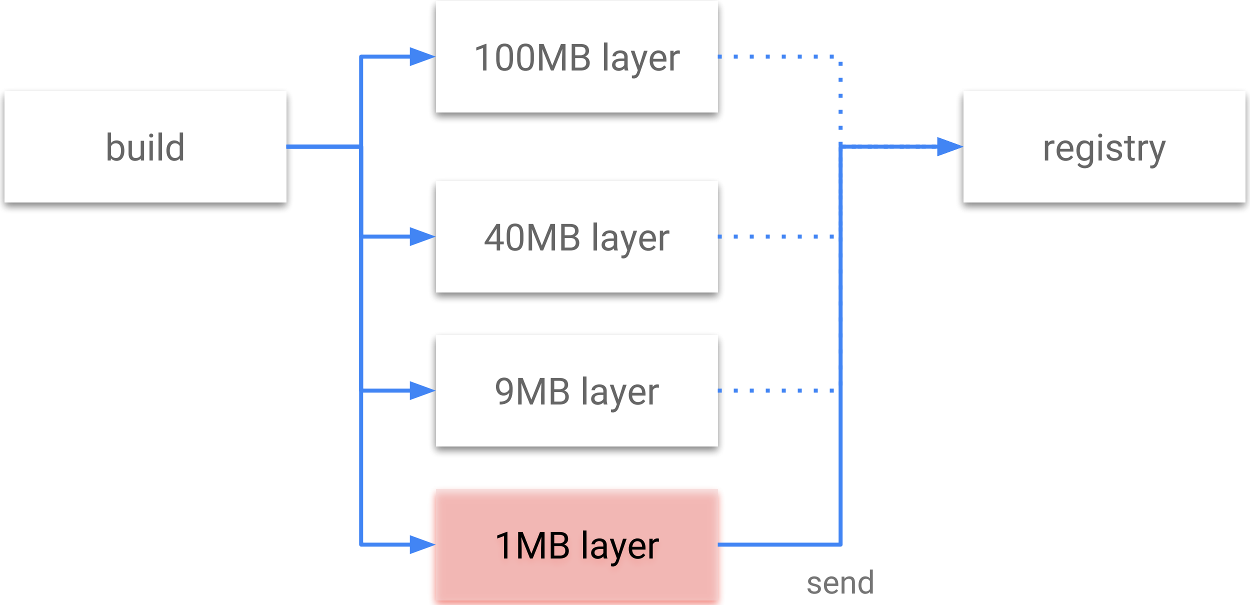
Docker registry

Set of layers, container configurations, and manifests



Docker registry

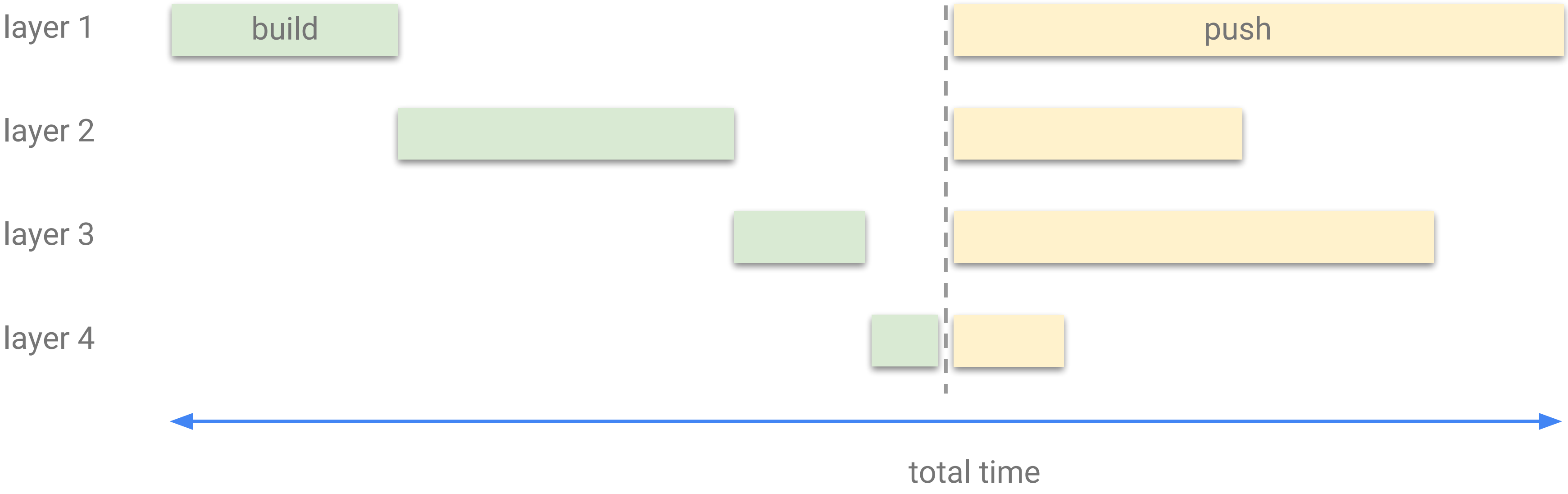
Set of layers, container configurations, and manifests



Jib does an optimized build like

```
FROM gcr.io/distroless/java

COPY target/dependencies /app/dependencies

COPY target/resources /app/resources

COPY target/classes /app/classes

ENTRYPOINT java -cp /app/dependencies/*:/app/resources:/app/classes my.app.Main
```

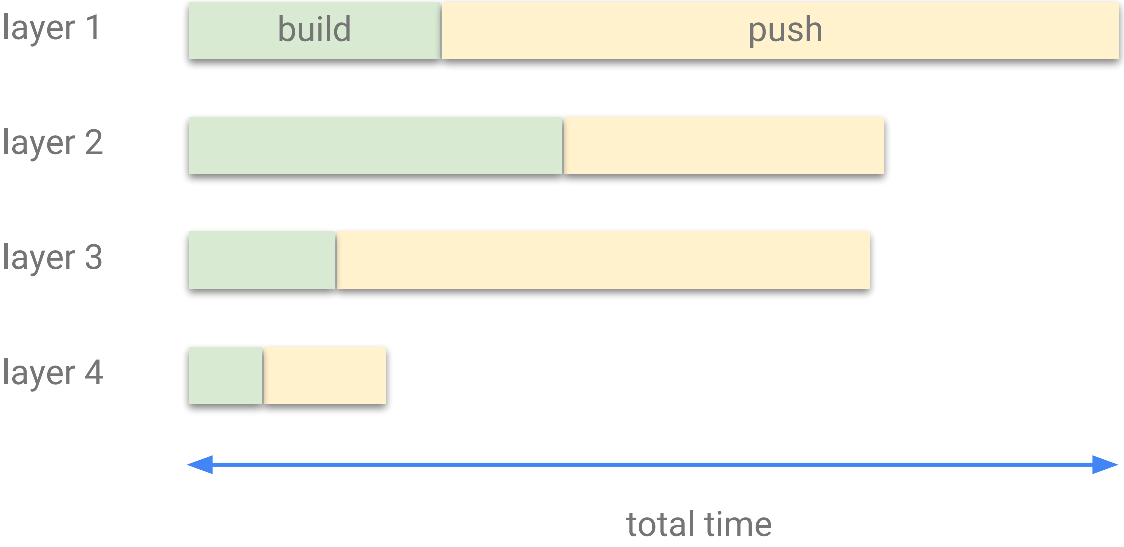
Containerizing with Docker



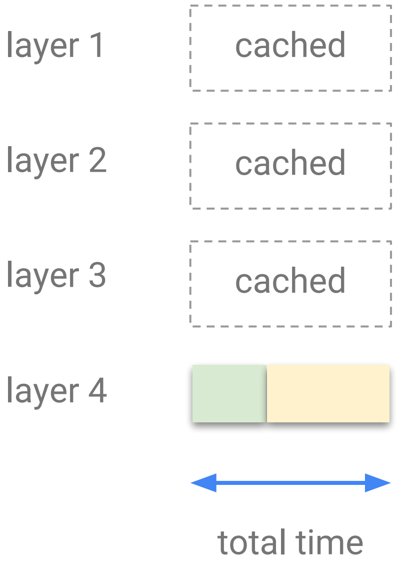
Google Cloud

github.com/GoogleContainerTools/jib

Containerizing with Jib



Containerizing with Jib (cached)



layer 1 cached

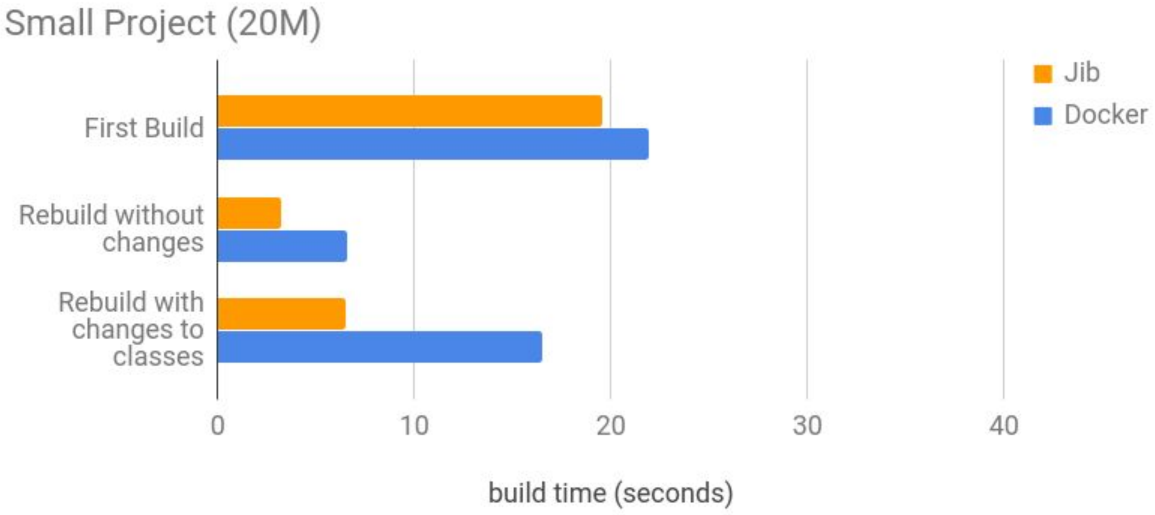
layer 2 cached

layer 3 cached

layer 4

total time

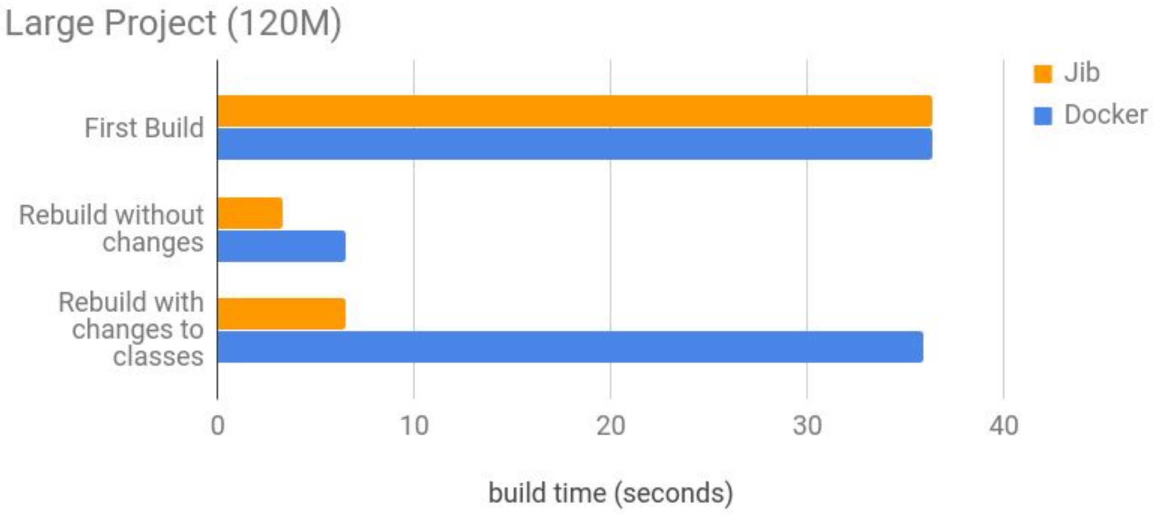
Jib vs Docker



Google Cloud

github.com/GoogleContainerTools/jib

Jib vs Docker



Google Cloud

github.com/GoogleContainerTools/jib

Reproducibility

Why reproducible ?

Version Control

Reduce variation between prod and dev

How?

Wipe metadata that vary between builds

~~Timestamps~~

~~Users~~

~~Groups~~

Possibilities for a container “compiler”

Possibilities for a container "compiler"

Smart inferences

Container optimizations

Even faster builds

Smaller images

Google Cloud

github.com/GoogleContainerTools/jib

Possibilities for a container “compiler”

Smart inferences

Container optimizations

Even faster builds

Smaller images

Tools for running the container

Possibilities for a container “compiler”

Smart inferences

Container optimizations

Even faster builds

Smaller images

Tools for running the container

Java Development on Kubernetes

Skaffold + Jib

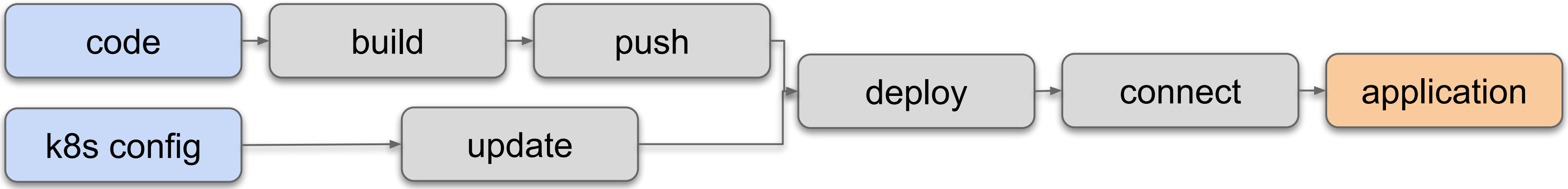
Continuous development for Kubernetes

Google Cloud

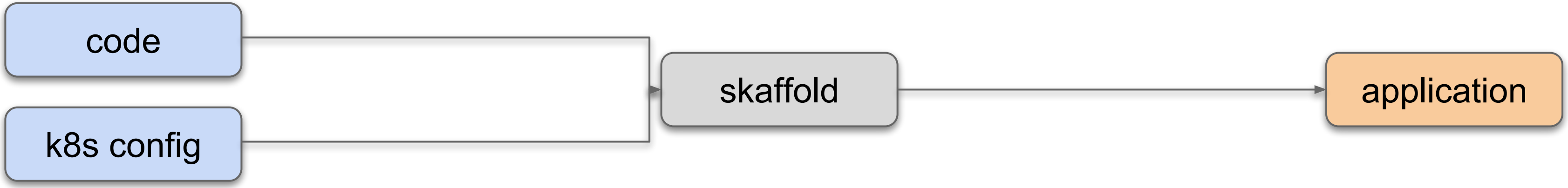
> Skaffold is a command line tool that facilitates continuous development for Kubernetes applications. You can iterate on your application source code locally then deploy to local or remote Kubernetes clusters. **Skaffold handles the workflow for building, pushing and deploying your application**. It can also be used in an automated context such as a CI/CD pipeline to leverage the same workflow and tooling when moving applications to production.

github.com/GoogleContainerTools/skaffold
official website

Development Process



Development Process



Demo

Jib Core

Java library for building containers

Google Cloud

```
Jib.from("busybox")
    .addLayer(Arrays.asList(Paths.get( "helloworld.sh")),
AbsoluteUnixPath.get( "/"))
    .setEntrypoint( "/bin/sh", "/helloworld.sh" )
    .containerize(
        Containerizer.to(DockerDaemonImage.named( "testjibcore")));
```

```java
Jib.from("busybox")
   .addLayer(Arrays.asList(Paths.get("helloworld.sh")), AbsoluteUnixPath.get("/"))
   .setEntrypoint("/bin/sh", "/helloworld.sh")
   .containerize(
       Containerizer.to(DockerDaemonImage.named( "testjibcore")));
```

Google Cloud

github.com/GoogleContainerTools/jib

```java
Jib.from("busybox")
    .addLayer(Arrays.asList(Paths.get( "helloworld.sh")),
AbsoluteUnixPath.get("/"))
    .setEntrypoint("/bin/sh", "/helloworld.sh")
    .containerize(
        Containerizer.to(DockerDaemonImage.named( "testjibcore")));
```

```
Jib.from("busybox")
    .addLayer(Arrays.asList(Paths.get( "helloworld.sh")),
AbsoluteUnixPath.get("/"))
    .setEntrypoint("/bin/sh", "/helloworld.sh")
    .containerize(
        Containerizer.to(DockerDaemonImage.named("testjibcore")));
```

Google Cloud

github.com/GoogleContainerTools/jib

Demo

```
$ git clone https://github.com/coollog/jib-core-demo && cd jib-core-demo/helloworld
$ ./mvnw exec:java
```

And more...

Support for WARs

Knative Jib BuildTemplate

sbt plugin

JHipster integration

...

The Future

More containerization tools for more languages

More Skaffold integration features

Be able to write code and have it run automatically in a distributed container cluster

...

github.com/GoogleContainerTools/jib

github.com/GoogleContainerTools/skaffold

saturnism.me/talk/docker-tips-and-tricks

github.com/GoogleContainerTools/distroless

Google Cloud