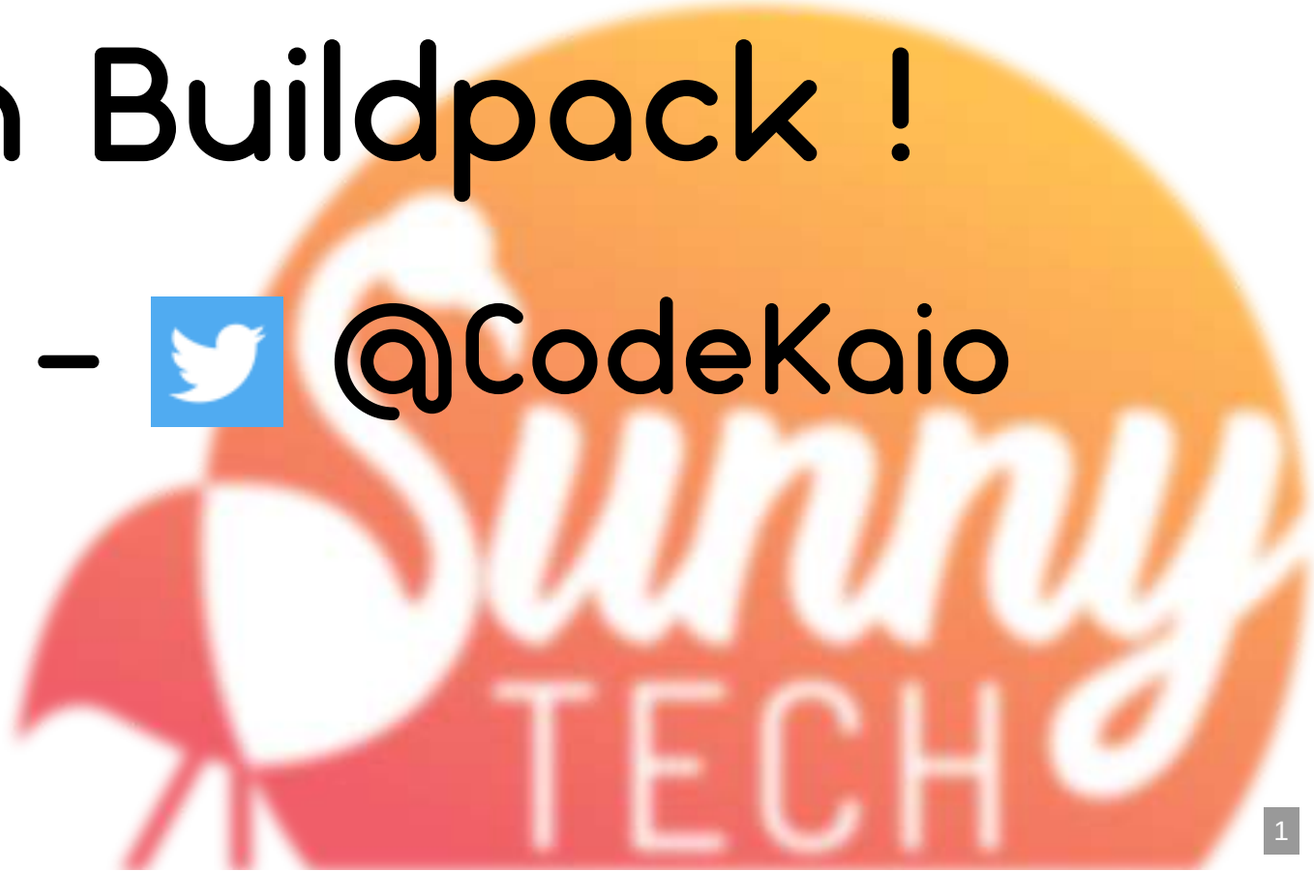
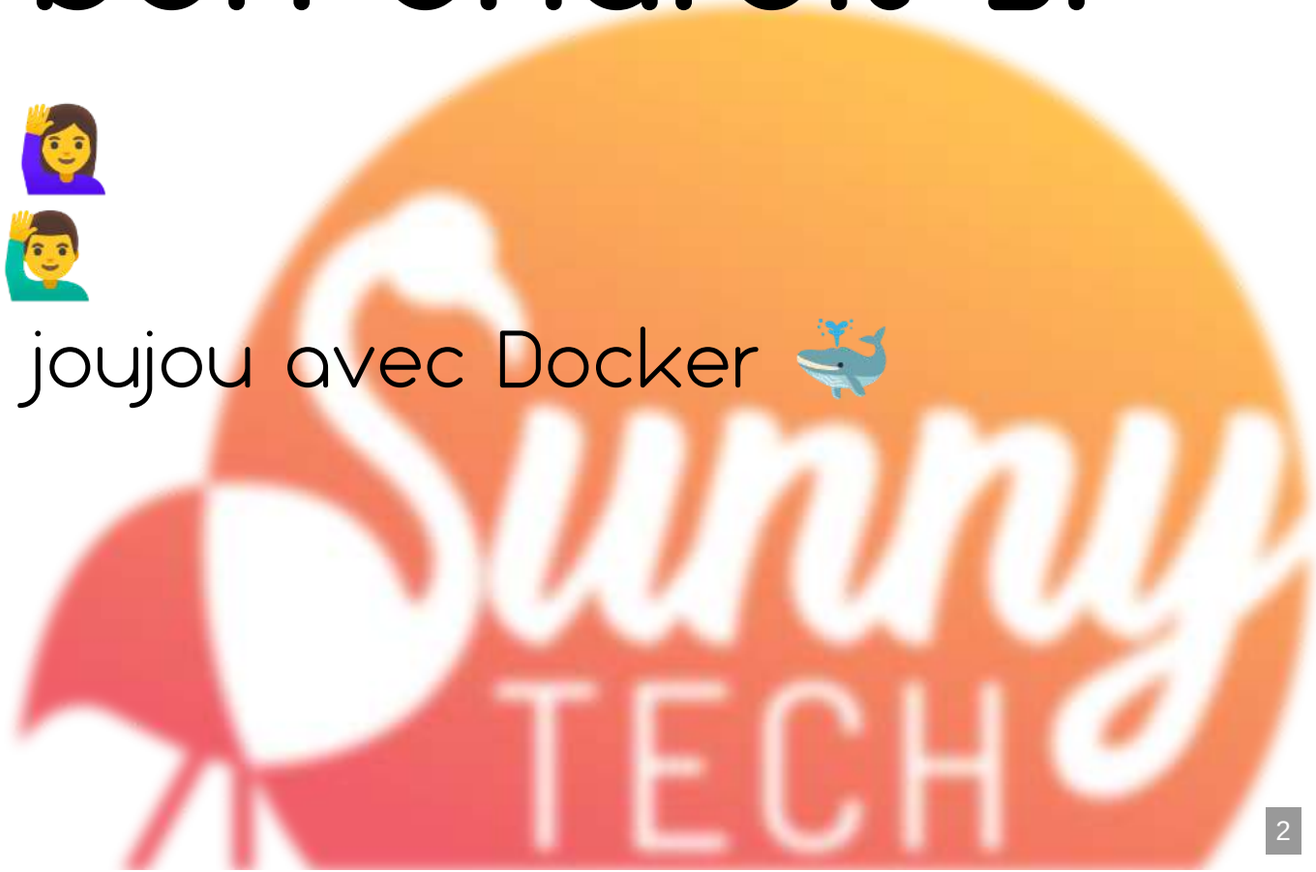
Laissez tomber vos Dockerfile, adoptez un Buildpack !

Julien Wittouck – @CodeKaio



Vous êtes au bon endroit si

- Vous êtes un·e Dev 🙋‍♀️
- Vous être un·e Ops 🤦‍♂️
- Vous faites souvent joujou avec Docker 🐳



Exécuter une application dans un container, c'est facile.

Un `Dockerfile` 🐋 et hop 🚀, en prod !

Non

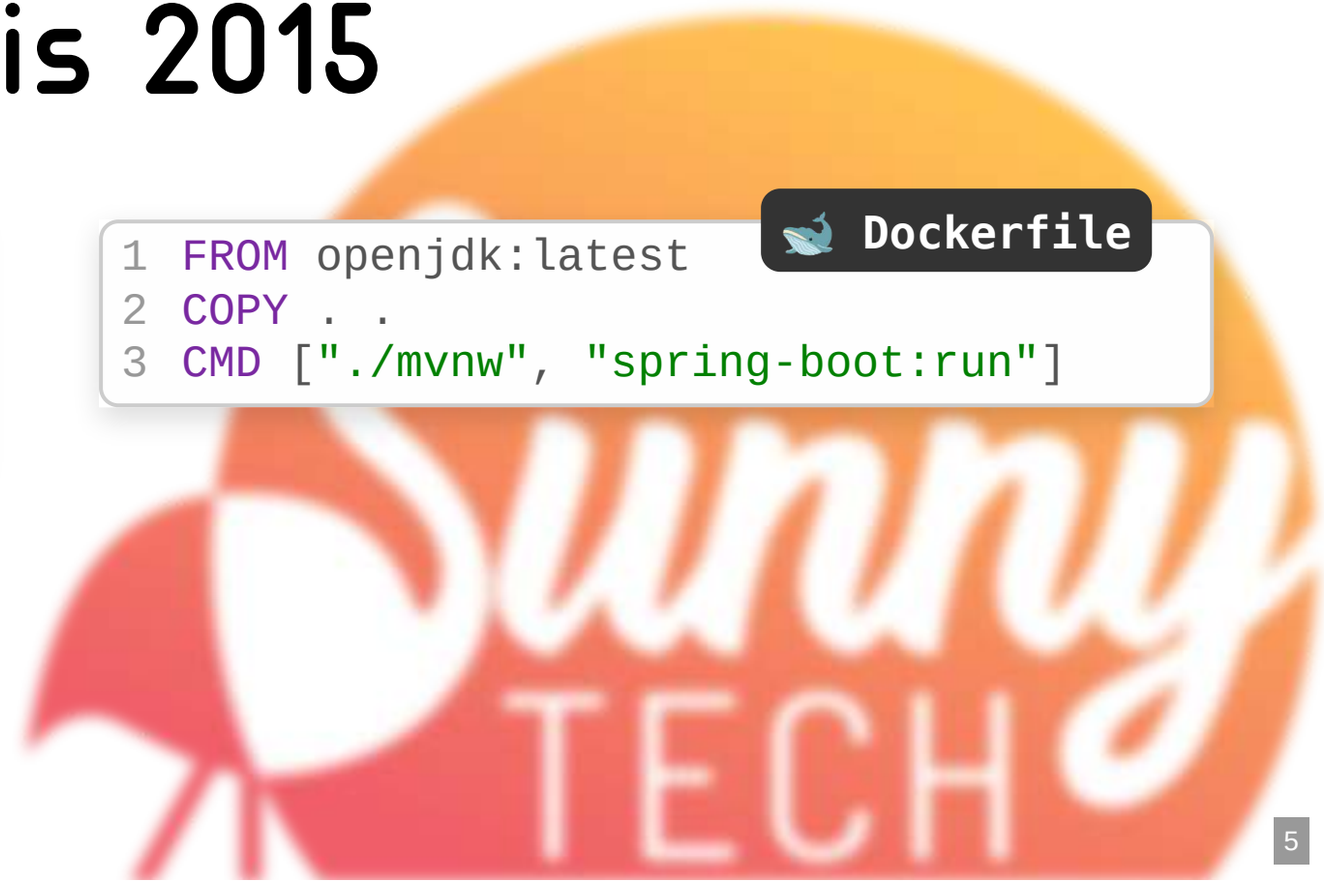
J'écris des (mauvais 🤮) Dockerfile depuis 2015

🐋 Dockerfile

```
1 FROM node:latest
2 COPY . .
3 RUN npm install
4 CMD ["npm", "start"]
```

🐋 Dockerfile

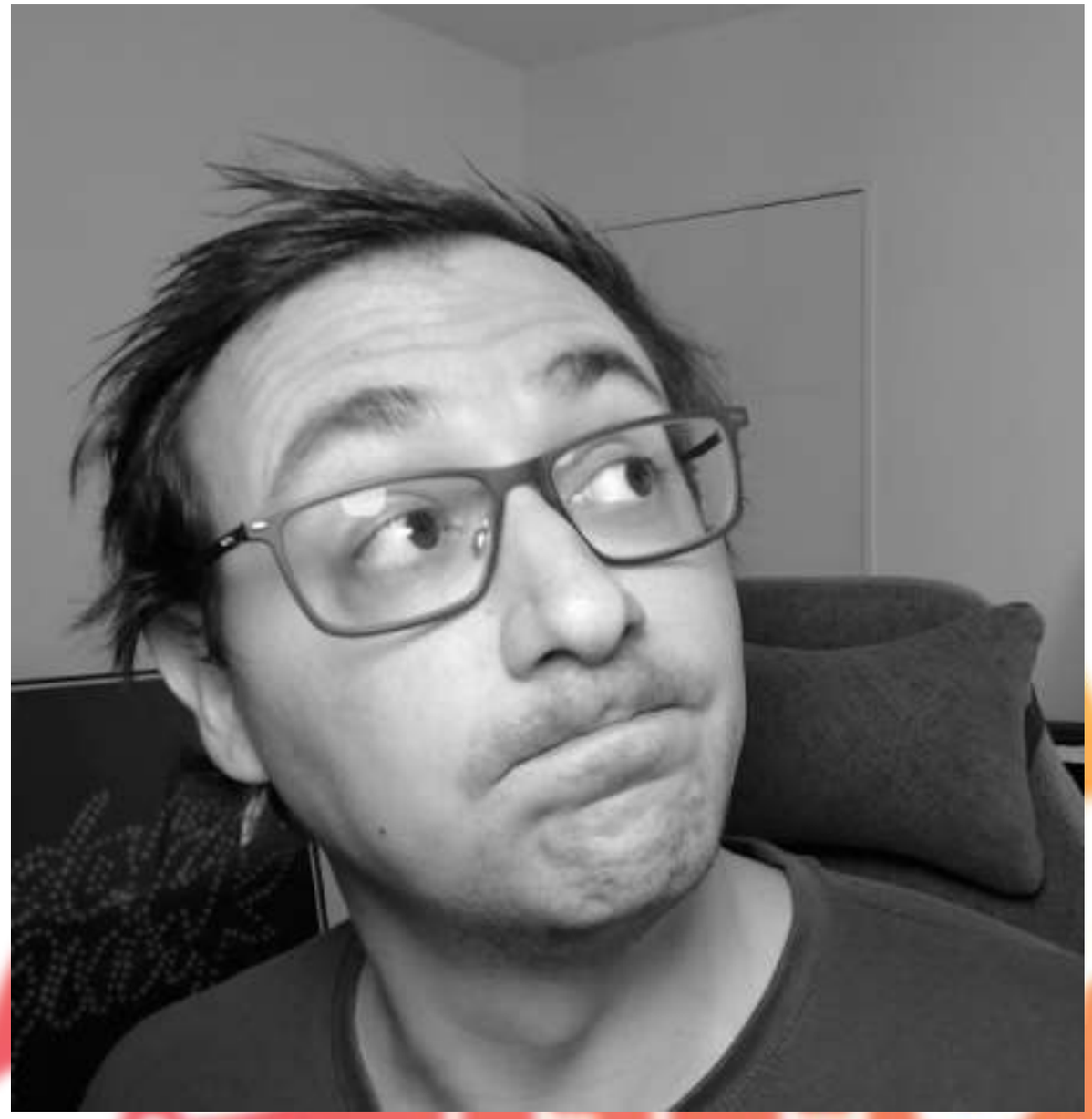
```
1 FROM openjdk:latest
2 COPY . .
3 CMD ["./mvnw", "spring-boot:run"]
```



Salut 👋

Moi c'est Julien

Freelance @CodeKaio
Associé @Ekité
Teacher @univ-lille



👶 Speaker (DevFest Lille - Sunny Tech)
🔎 Reviewer Cloud Nord

Allez, on embarque !

Dockerfiles

Comment Docker construit une image ?

- "Exécute" le `Dockerfile`
- Dans des containers séparés
- "SNAPSHOT" le filesystem à chaque étape pour créer les layers

Le *Hello World* Spring Boot, selon Docker 🤮

www.docker.com/blog

🐋 Dockerfile

```
FROM eclipse-temurin:17-jdk-focal

WORKDIR /app

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

COPY src ./src

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

Le *Hello World* Spring Boot, selon Docker 🤮

www.docker.com/blog

🐋 Dockerfile

```dockerfile
FROM eclipse-temurin:17-jdk-focal

WORKDIR /app

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

COPY src ./src

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

Le *Hello World* Spring Boot, selon Docker 🤮

www.docker.com/blog

🐋 Dockerfile

```dockerfile
FROM eclipse-temurin:17-jdk-focal

WORKDIR /app

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

COPY src ./src

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

Le *Hello World* Spring Boot, selon Docker 🤮

www.docker.com/blog

🐋 Dockerfile

```dockerfile
FROM eclipse-temurin:17-jdk-focal

WORKDIR /app

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

COPY src ./src

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

Le *Hello World* Spring Boot, selon Docker 🤮

www.docker.com/blog

🐋 Dockerfile

```dockerfile
FROM eclipse-temurin:17-jdk-focal

WORKDIR /app

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

COPY src ./src

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

Le *Hello World* Spring Boot, selon Spring Boot 🤔

spring.io/guides

🐋 Dockerfile

```dockerfile
# build stage
FROM eclipse-temurin:17-jdk-alpine as build
WORKDIR /workspace/app

COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src

RUN ./mvnw install -DskipTests
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

# run stage
FROM eclipse-temurin:17-jdk-alpine
```

Le *Hello World* Spring Boot, selon Spring Boot 🤔

spring.io/guides

🐋 Dockerfile

```dockerfile
# build stage
FROM eclipse-temurin:17-jdk-alpine as build
WORKDIR /workspace/app

COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src

RUN ./mvnw install -DskipTests
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

# run stage
FROM eclipse-temurin:17-jdk-alpine

```

Le *Hello World* Spring Boot, selon Spring Boot 🤔

spring.io/guides

🐳 Dockerfile

```dockerfile
# build stage
FROM eclipse-temurin:17-jdk-alpine as build
WORKDIR /workspace/app

COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src

RUN ./mvnw install -DskipTests
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

# run stage
FROM eclipse-temurin:17-jdk-alpine
```

Le *Hello World* Spring Boot, selon Spring Boot 🤔

spring.io/guides

🐳 Dockerfile

```
4
5  COPY mvnw .
6  COPY .mvn .mvn
7  COPY pom.xml .
8  COPY src src
9
10 RUN ./mvnw install -DskipTests
11 RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)
12
13 # run stage
14 FROM eclipse-temurin:17-jdk-alpine
15
16 RUN addgroup -S demo && adduser -S demo -G demo
17 USER demo
18
```

Le *Hello World* Spring Boot, selon Spring Boot 🤔

spring.io/guides

🐳 Dockerfile

```
RUN ./mvnw install -DskipTests
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

# run stage
FROM eclipse-temurin:17-jdk-alpine

RUN addgroup -S demo && adduser -S demo -G demo
USER demo

VOLUME /tmp
ARG DEPENDENCY=/workspace/app/target/dependency
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java","-cp","app:app/lib/*","hello.Application"]
```

Le *Hello World* Spring Boot, selon Spring Boot 🤔

spring.io/guides

🐳 Dockerfile

```
10 RUN ./mvnw install -DskipTests
11 RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)
12
13 # run stage
14 FROM eclipse-temurin:17-jdk-alpine
15
16 RUN addgroup -S demo && adduser -S demo -G demo
17 USER demo
18
19 VOLUME /tmp
20 ARG DEPENDENCY=/workspace/app/target/dependency
21 COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
22 COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
23 COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
24 ENTRYPOINT ["java","-cp","app:app/ lib/*","hello.Application"]
```

Le *Hello World* Spring Boot, selon Spring Boot 🤔

spring.io/guides

🐳 Dockerfile

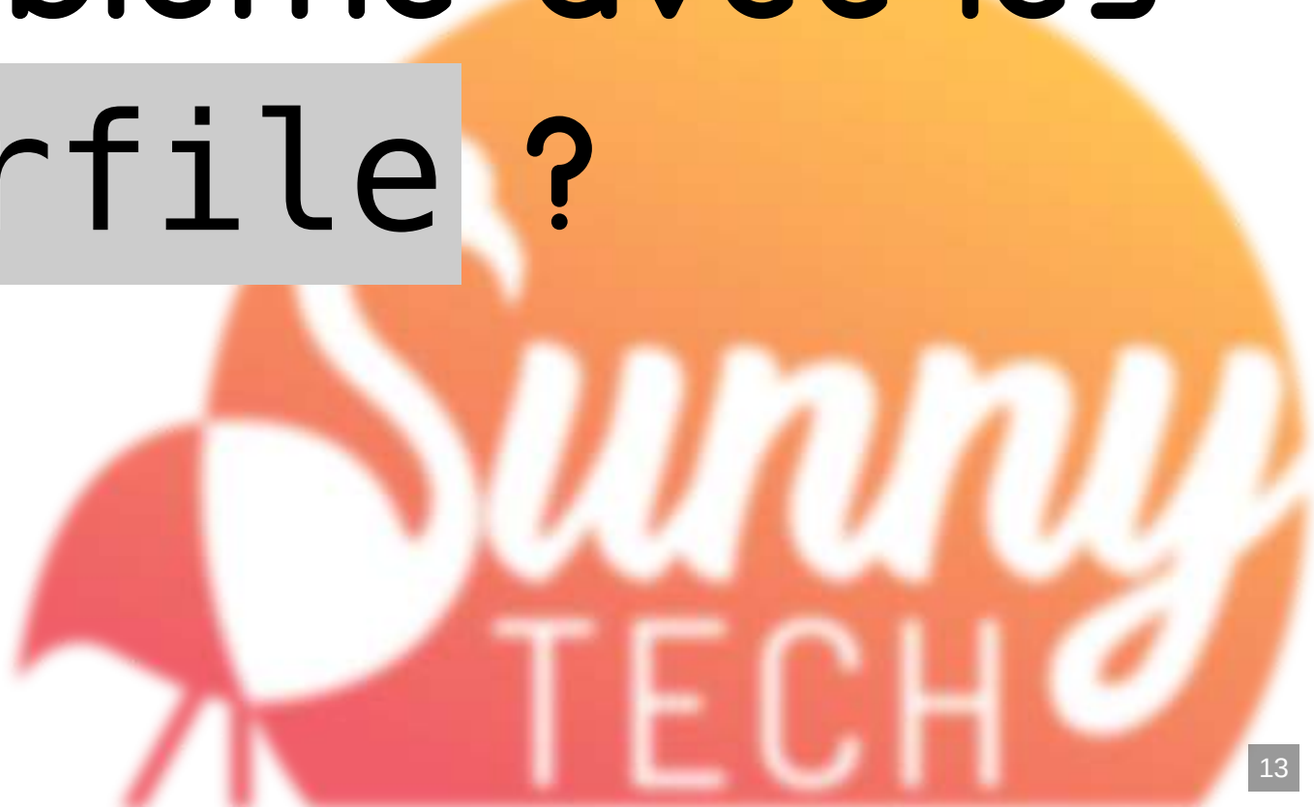
```
10 RUN ./mvnw install -DskipTests
11 RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)
12
13 # run stage
14 FROM eclipse-temurin:17-jdk-alpine
15
16 RUN addgroup -S demo && adduser -S demo -G demo
17 USER demo
18
19 VOLUME /tmp
20 ARG DEPENDENCY=/workspace/app/target/dependency
21 COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
22 COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
23 COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
24 ENTRYPOINT ["java","-cp","app:app/ lib/*","hello.Application"]
```

L'instant philosophie

Est-ce que ce `Dockerfile` était parfait ?

Est-ce qu'on attend d'un·e développeur·euse l'écriture d'un tel `Dockerfile` ?

Quel est le problème avec les `Dockerfile` ?



Ce que pense Darren Shepherd

(former @Rancher_Labs, k3s creator)

Darren Shepherd
@ibuildthecloud · **Follow**

So issues I have

1. Caching. I've been told this is hard. I believe there is stupider and more effective ways to (not) do it
2. Layers are soooo 2018
3. Multi stage is cool and all, but you really only need two stages (build env, run env)
4. COPY is kinda pointless.

> **Darren Shepherd** @ibuildthecloud
> I don't know if I can shake the urge to try to build a better Dockerfile.

9:25 AM · Jun 27, 2023

43 Reply Copy link

Read 6 replies

Darren Shepherd · Jun 27, 2023
@ibuildthecloud · **Follow**
Replying to @ibuildthecloud

Layers just suck. They've never really been used effectively and cause tons of cache issues. I'm not fundamentally opposed to layers, but maybe just 3. I can do 3.

Darren Shepherd
@ibuildthecloud · **Follow**

You really can't build a properly good docker image without multi stage because you don't want your compiler in the final image. I think a better UX can be built assuming you basically always want to stages: build and run.

9:34 AM · Jun 27, 2023

8 Reply Copy link

Read 6 replies

Écrire un 🐋 `Dockerfile`

- Image de base, Runtime / Version (`FROM`)
- Optimiser les layers (`RUN`)
- Script de démarrage (`ENTRYPOINT`)
- Configuration (`ENV`)
- Sécurité
  - User non-root (`USER`)
  - Versions des packages / runtime à jour
  - Pas de code source dans l'image finale
  - Pouvoir patcher les images en cas de nouvelle faille
  - Pas de secrets

Comment font les développeurs qui doivent écrire un 🐋 `Dockerfile` ?

- 🤮 Au pire : Stackoverflow, ChatGPT, ou un tutoriel random
- 🤢 Bof : le Dockerfile de Spring Boot, ou NodeJs
- 🤓 Mieux : ils sont experts Docker, ou un Ops écrit le `Dockerfile` avec eux

Et c'est pire en entreprise

Chaque projet a son propre `Dockerfile`, forcément différent des autres

Bienvenue en enfer 😈

Quand les normes évoluent, ouvrir des change-request sur tous les projets de l'entreprise pour mettre à jour les Dockerfile

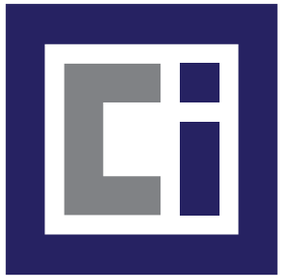
Les images Docker

Images 🐋 Docker ?

Images ~~Docker~~ ?

Images ~~Docker~~ OCI ?

Images ~~Docker~~ OCI ?



OPEN CONTAINER INITIATIVE

Images ~~Docker~~ OCI ?

OPEN CONTAINER INITIATIVE

On parle d'image OCI depuis 2015 (Open Container Initiative)

Normalisé : github.com/opencontainers/image-spec

La vision qu'on a souvent :

Les layers :

distrib + runtime/middleware + code

JSON + tar.gz = 💙

✈ manifest.json

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
  "layers": [
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 30430275,
      "digest": "sha256:d1669123f28121211977ed38e663dca1a397c0c001e5386598b96c8
    },
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 17038759,
      "digest": "sha256:2ec73b48ae406646223453ca41d5d6b7cb739853fb7a44f15d35a31
    },
    {
```

On appelle ça un *Manifest* d'image

JSON + tar.gz = 💙

✈ manifest.json

```
  "layers": [
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 30430275,
      "digest": "sha256:d1669123f28121211977ed38e663dca1a397c0c001e5386598b96c8
    },
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 17038759,
      "digest": "sha256:2ec73b48ae406646223453ca41d5d6b7cb739853fb7a44f15d35a31
    },
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 192587566,
      "digest": "sha256:9dbb3ddf83e9096c0e2f32bfa93d16285d2e9586b1a9aa25e33d04c
```

On appelle ça un *Manifest* d'image

JSON + tar.gz = 💙

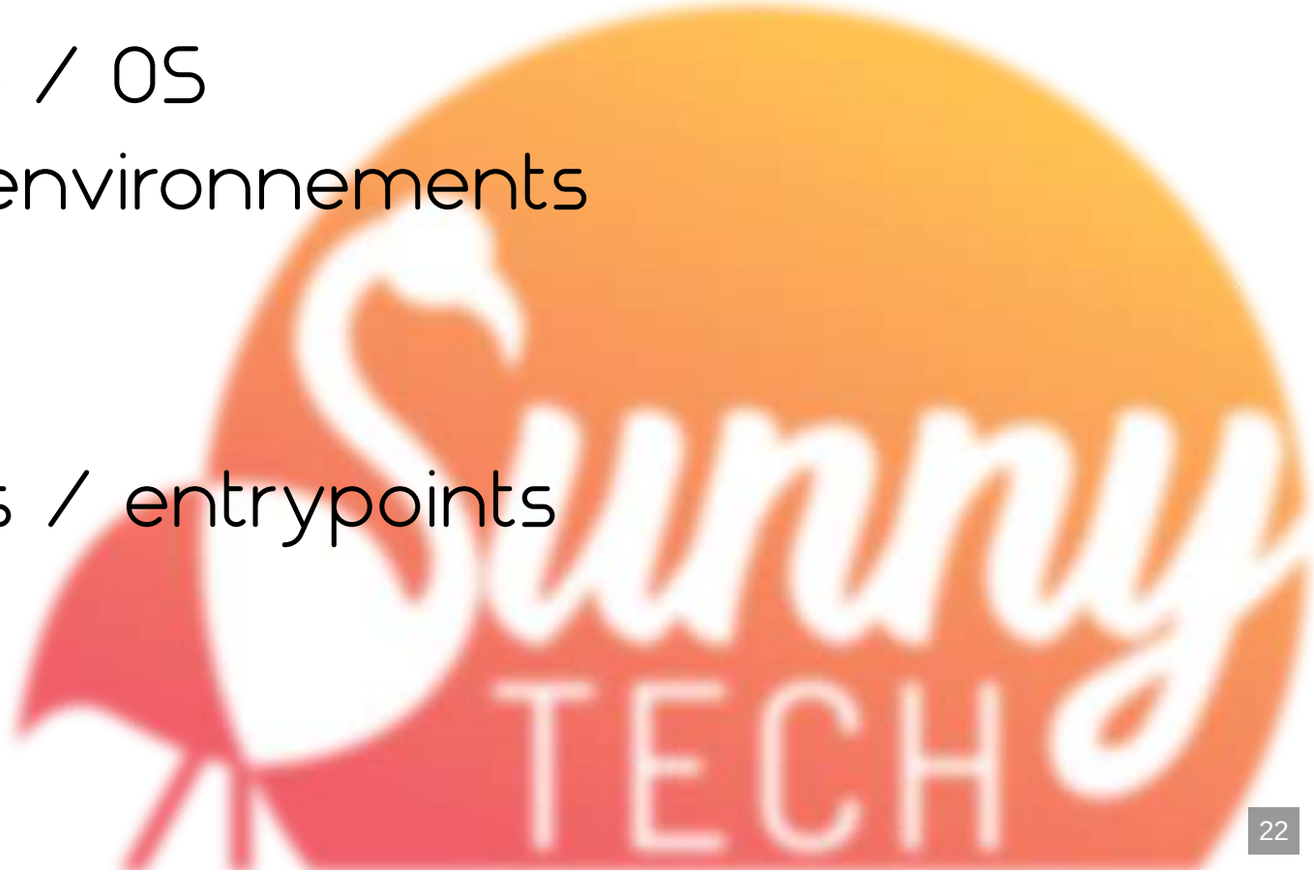
✈ manifest.json

```
17          "size": 192587566,
18          "digest": "sha256:9dbb3ddf83e9096c0e2f32bfa93d16285d2e9586b1a9aa25e33d04c
19        },
20        {
21          "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
22          "size": 175,
23          "digest": "sha256:08d2567dd626029019cc940915699f23b075d05189d99319604fbda
24        }
25      ],
26      "config": {
27        "mediaType": "application/vnd.docker.container.image.v1+json",
28        "size": 6305,
29        "digest": "sha256:50440189b0f4cd6264c2f03f92acf4772680d864e3a0d422ef4c46373
30      }
31  }
```

On appelle ça un *Manifest* d'image

La configuration aussi fait partie de l'image OCI !

- architecture / OS
- variables d'environnements
- users
- labels
- commandes / entrypoints
- ports



eclipse-temurin:17.0.7_7-jdk

📦 config.json

```
{
  "architecture": "amd64",
  "config": {
    "Hostname": "",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [
      "PATH=/opt/java/openjdk/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr
      "JAVA_HOME=/opt/java/openjdk",
```

eclipse-temurin:17.0.7_7-jdk

📦 config.json

```
{
  "architecture": "amd64",
  "config": {
    "Hostname": "",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [
      "PATH=/opt/java/openjdk/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr
      "JAVA_HOME=/opt/java/openjdk",
```

eclipse-temurin:17.0.7_7-jdk

📦 config.json

```
10      "Tty": false,
11      "OpenStdin": false,
12      "StdinOnce": false,
13      "Env": [
14        "PATH=/opt/java/openjdk/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr
15        "JAVA_HOME=/opt/java/openjdk",
16        "LANG=en_US.UTF-8",
17        "LANGUAGE=en_US:en",
18        "LC_ALL=en_US.UTF-8",
19        "JAVA_VERSION=jdk-17.0.7+7"
20      ],
21      "Cmd": [
22        "jshell"
23      ],
```

eclipse-temurin:17.0.7_7-jdk

📦 config.json

```
{
  "architecture": "amd64",
  "config": {
    "Hostname": "",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [
      "PATH=/opt/java/openjdk/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr
      "JAVA_HOME=/opt/java/openjdk",
```

eclipse-temurin:17.0.7_7-jdk

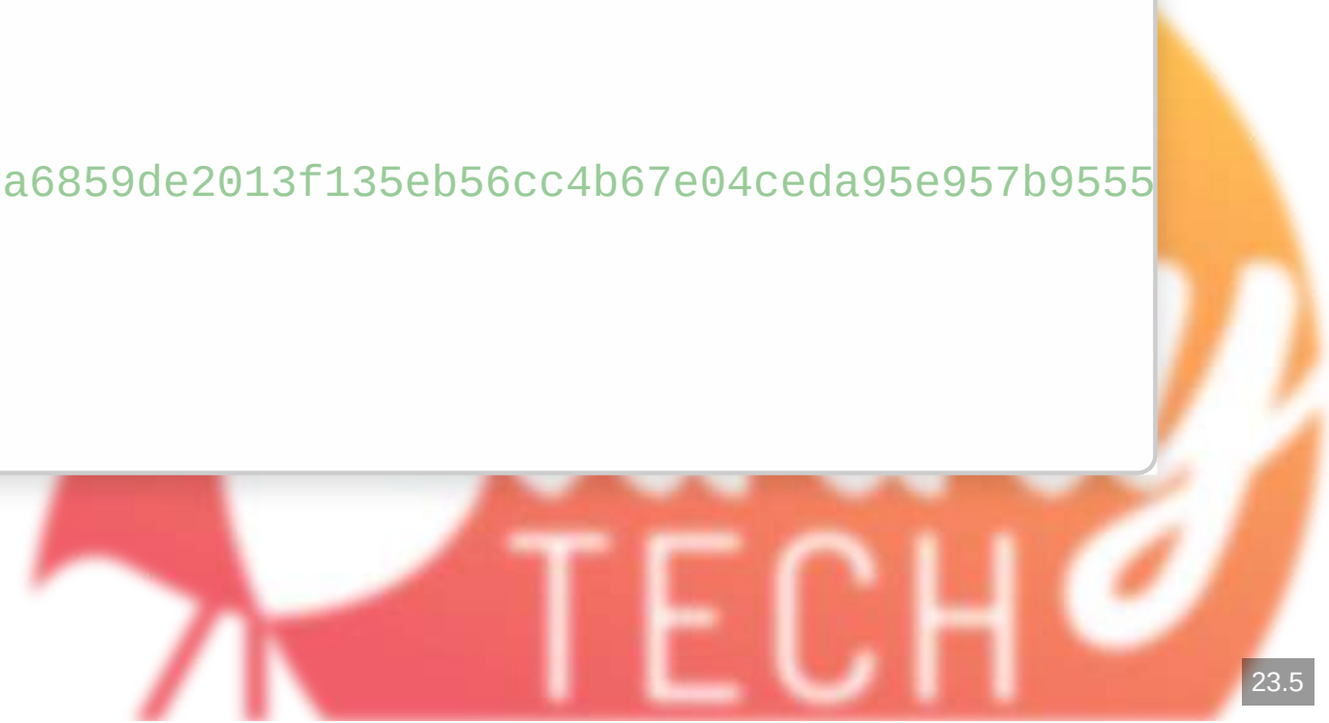
📦 config.json


```
    },
    "Image": "sha256:1dcdc9129900e8da6859de2013f135eb56cc4b67e04ceda95e957b9355
    "Volumes": null,
    "WorkingDir": "",
    "Entrypoint": null,
    "OnBuild": null,
    "Labels": {
      "org.opencontainers.image.ref.name": "ubuntu",
      "org.opencontainers.image.version": "22.04"
    }
  },
  "created": "2023-06-02T01:44:46.577735785Z",
  "docker_version": "20.10.23",
  "os": "linux",
  "rootfs": {
    "type": "layers",
```

eclipse-temurin:17.0.7_7-jdk

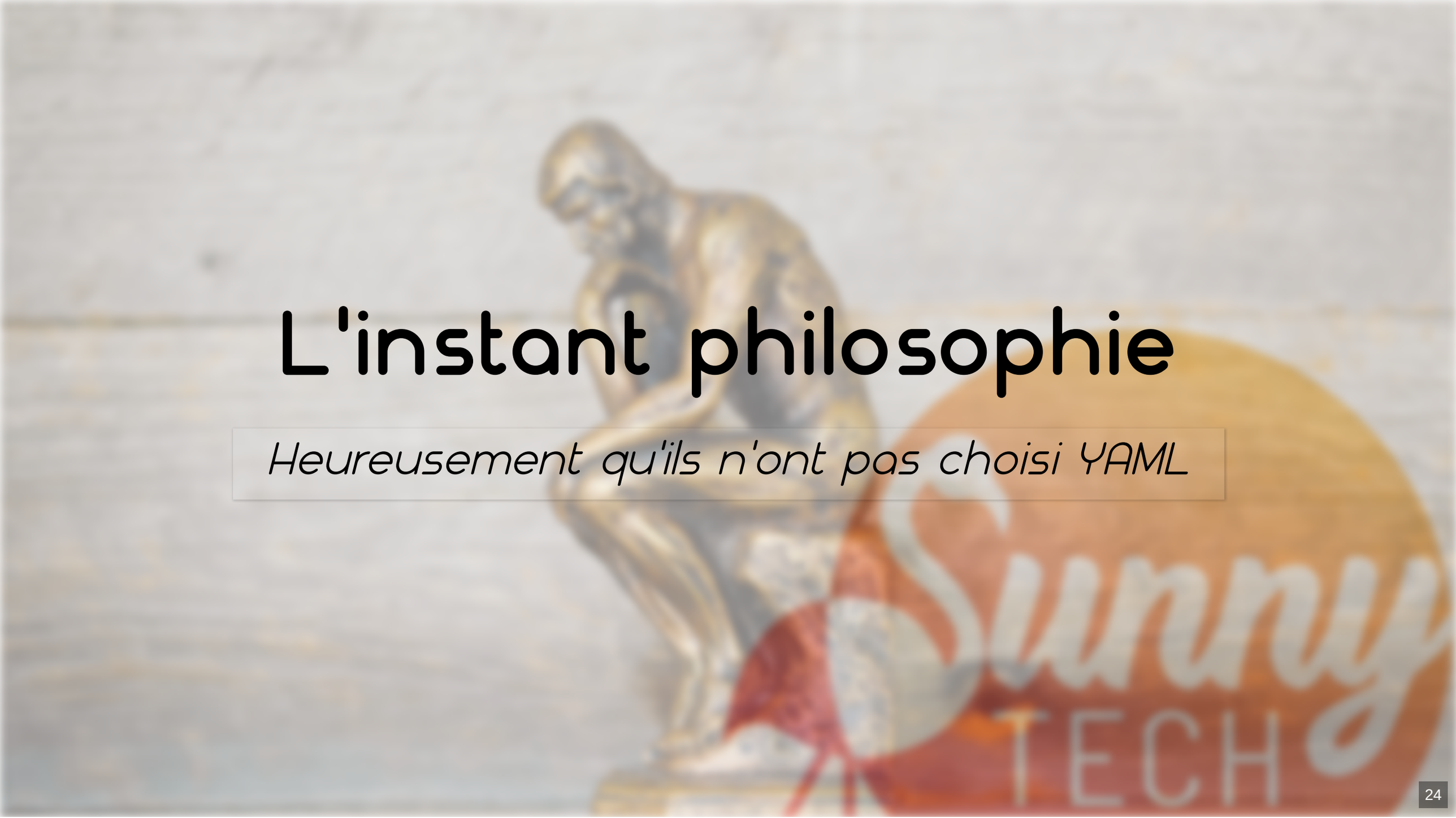
📦 config.json

```
15          "JAVA_HOME=/opt/java/openjdk",
16          "LANG=en_US.UTF-8",
17          "LANGUAGE=en_US:en",
18          "LC_ALL=en_US.UTF-8",
19          "JAVA_VERSION=jdk-17.0.7+7"
20        ],
21        "Cmd": [
22          "jshell"
23        ],
24        "Image": "sha256:1dcdc9129900e8da6859de2013f135eb56cc4b67e04ceda95e957b9555
25        "Volumes": null,
26        "WorkingDir": "",
27        "Entrypoint": null,
28        "OnBuild": null,
29        "Labels": {
```



L'instant philosophie

Heureusement qu'ils n'ont pas choisi YAML



docker image inspect 🔍

> bash

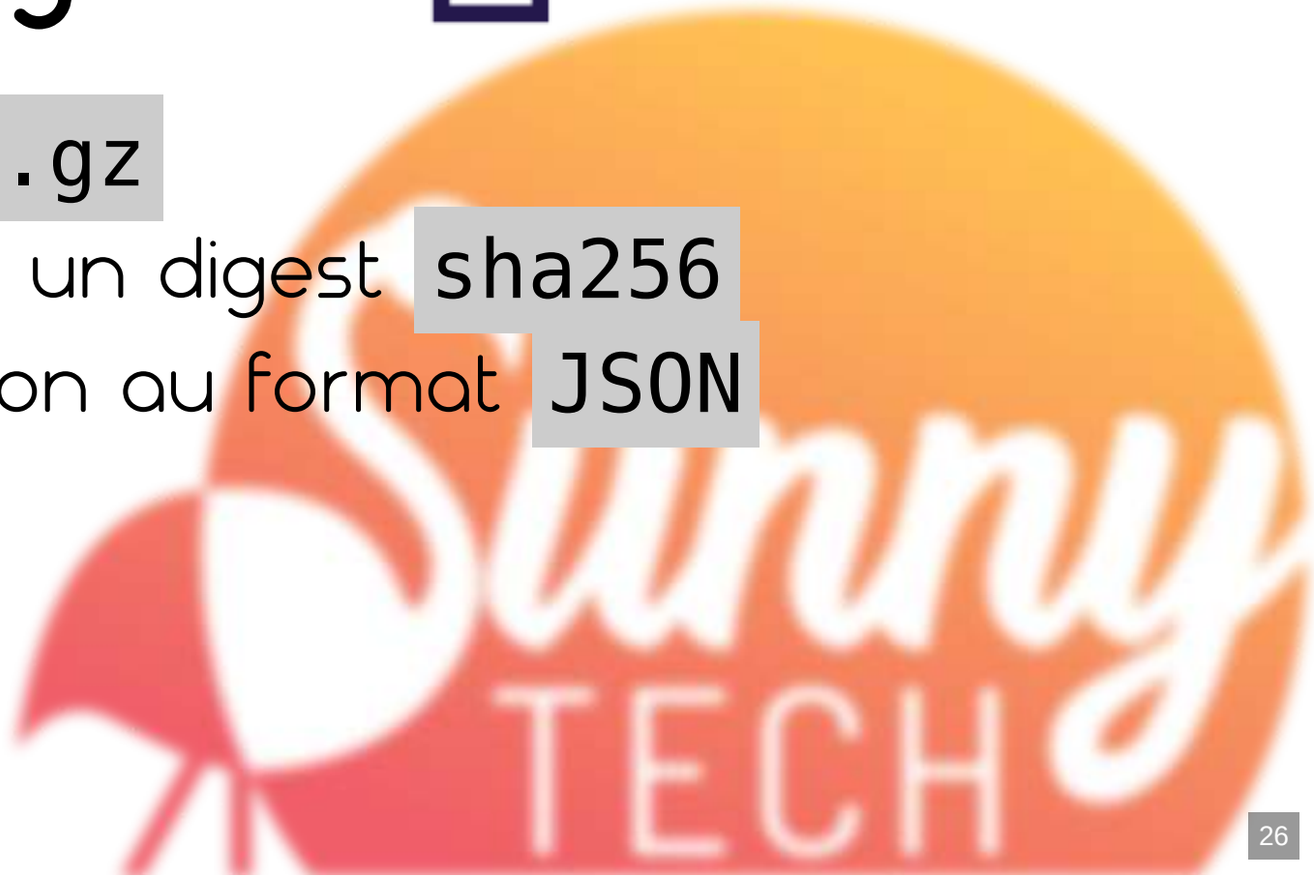
```
docker image inspect eclipse-temurin:17.0.7_7-jdk | bat -l json
```

> bash

```
# on parcours une image avec dive !
dive petclinic:dockerfile
```

Une image OCI

- Des fichiers `tar.gz`
- Chaque fichier a un digest `sha256`
- De la configuration au format `JSON`



Et si ?

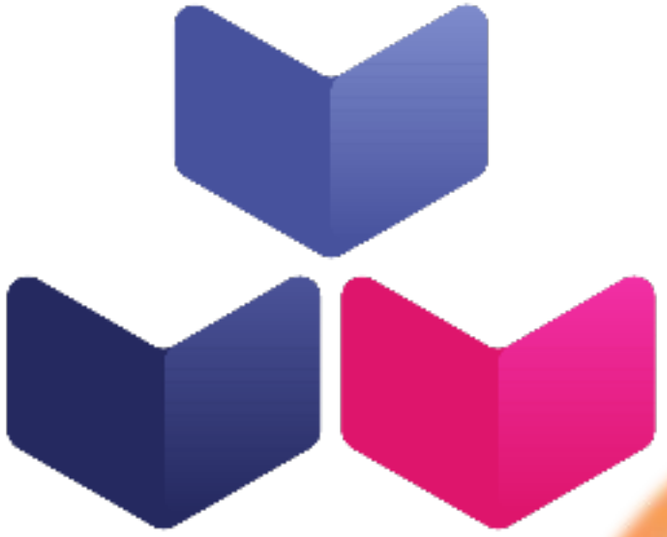
On pouvait générer tout ça ?

On pourrait créer des images OCI, sans avoir besoin de 🐋 Docker ou d'un `Dockerfile` !

L'instant philosophie

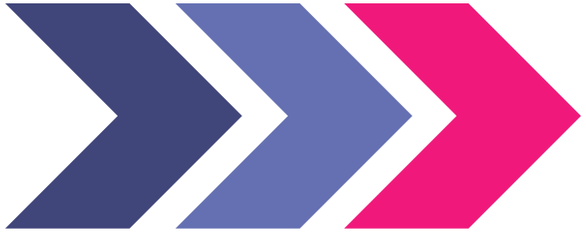
ça veut aussi dire, qu'on peut modifier une image, juste en allant modifier son manifest

Les buildpacks



Buildpacks.io

source



without Dockerfiles

OCI image

Buildpacks

- Concept par Heroku en **2011** pour leur propre besoin de PaaS multi-langage.
- CNB (Cloud Native Buildpacks) initié en **2018**, et a rejoint la CNCF (Cloud Native Computing Foundation) en 2018 en "Incubating".

Comment ça fonctionne ?

builder

construit une image

buildpack

contribue à une ou plusieurs layers dans la construction

Un buildpack est composé de 2 binaires :

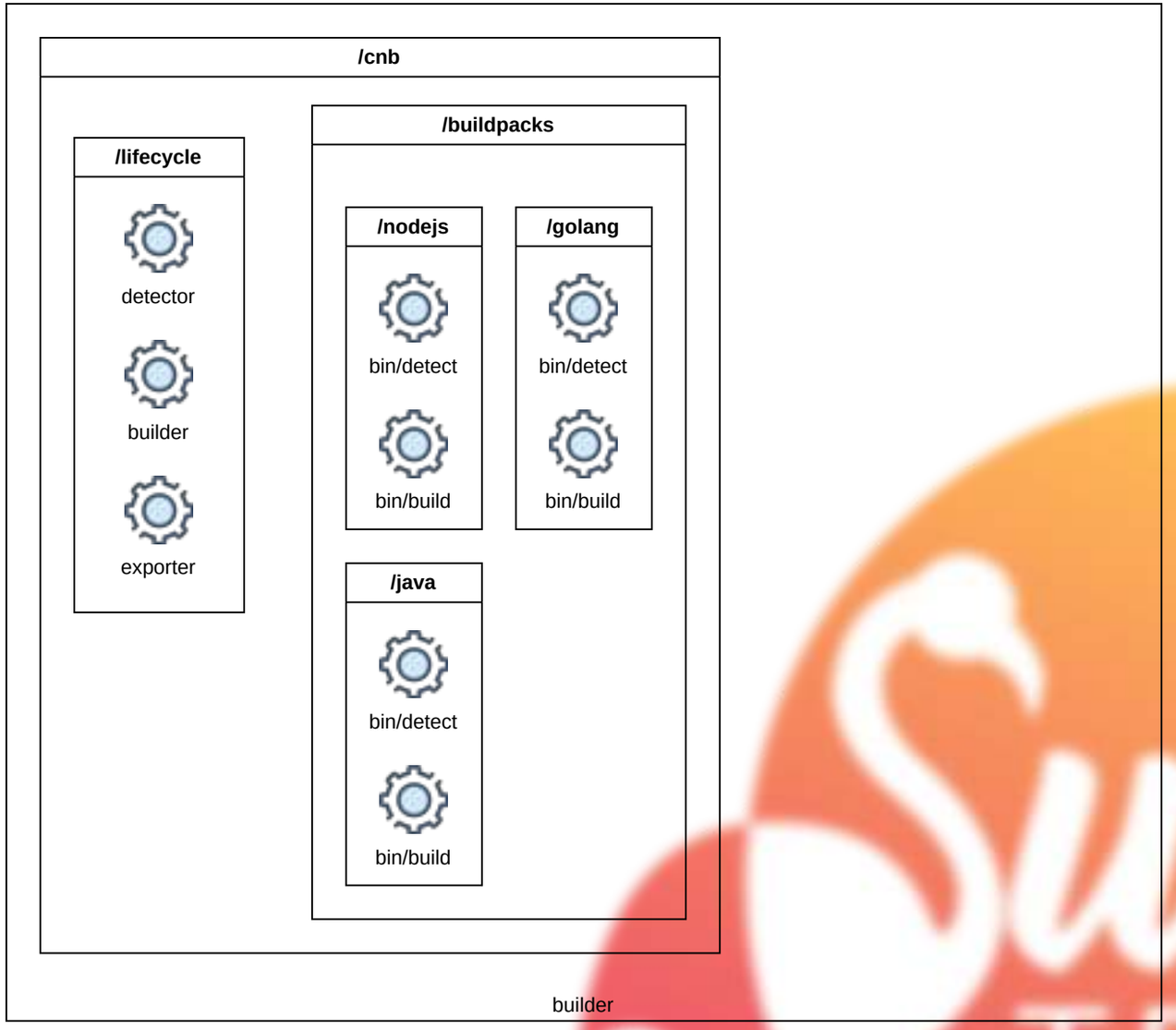
- `/bin/detect` : indique si le buildpack doit être activé
- `/bin/build` : contribue à la construction d'une ou plusieurs layers

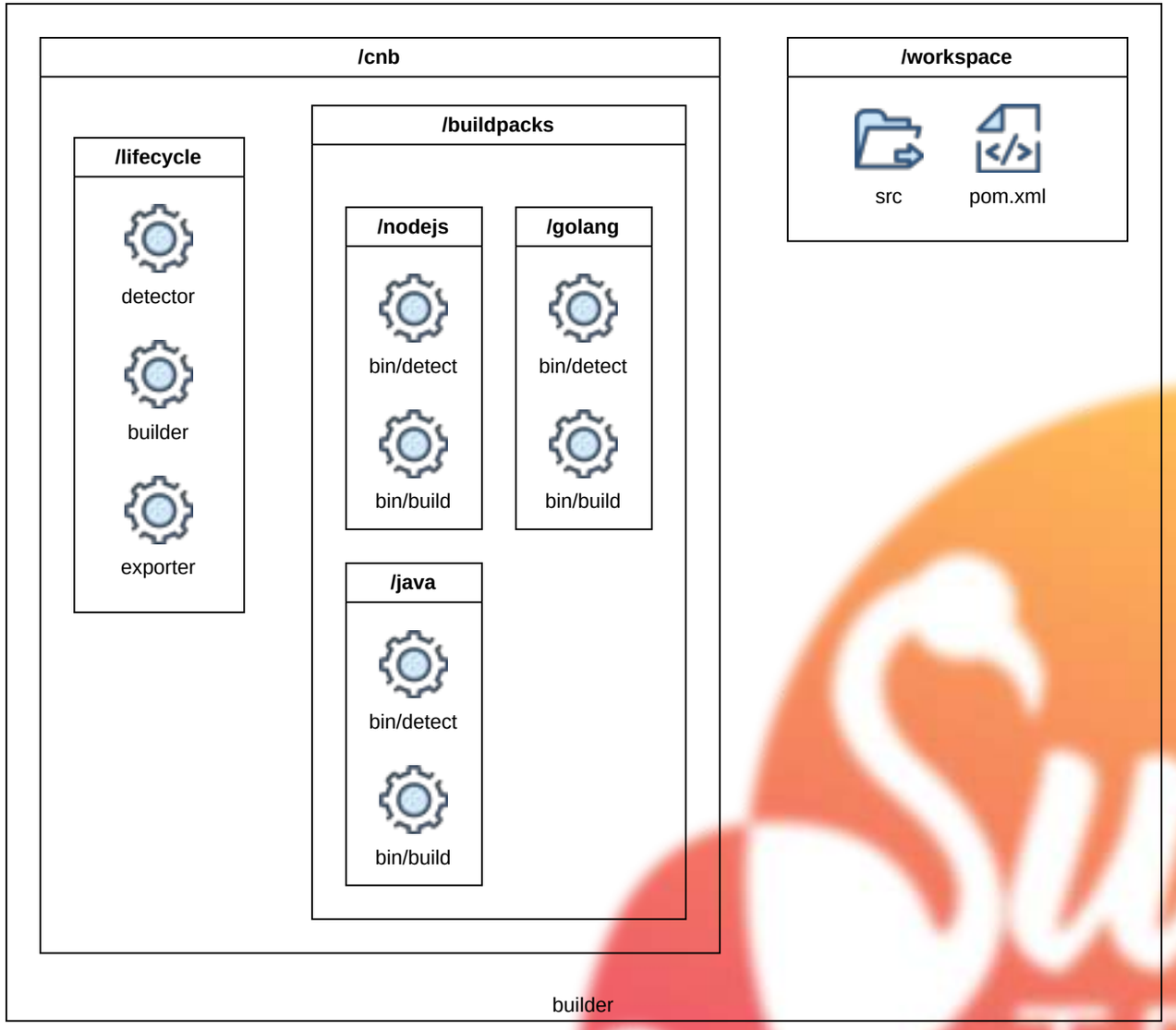
Le builder contient des scripts `lifecycle` :

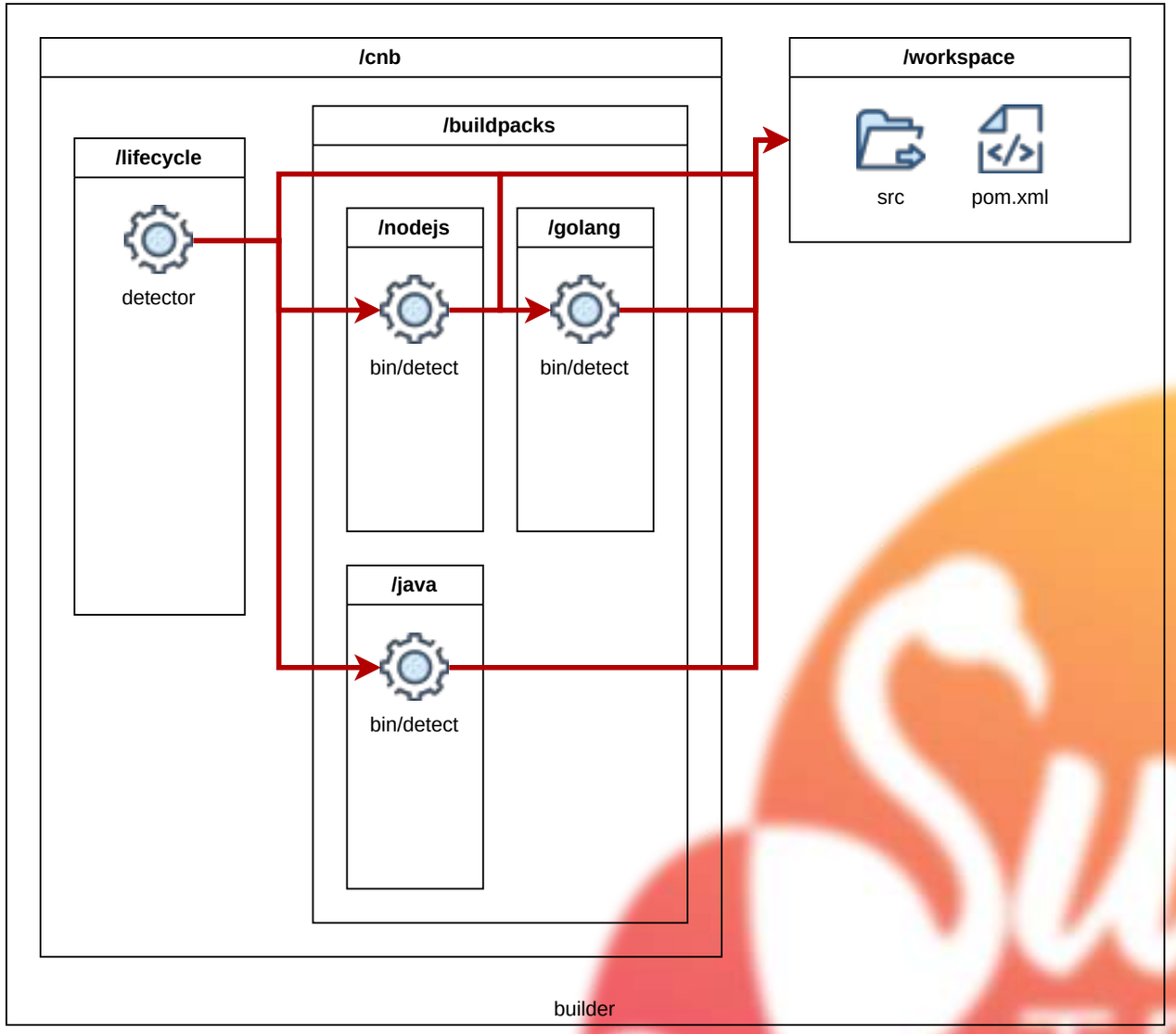
1. Code source dans un répertoire `/workspace`
2. Interroge chaque buildpack avec `/bin/detect`
3. Exécute tous les buildpacks qui doivent être exécutés
4. Chaque buildpack contribue une ou plusieurs layers dans `/layer`
5. Les layers dans `/layer` sont exportées pour créer une image OCI !

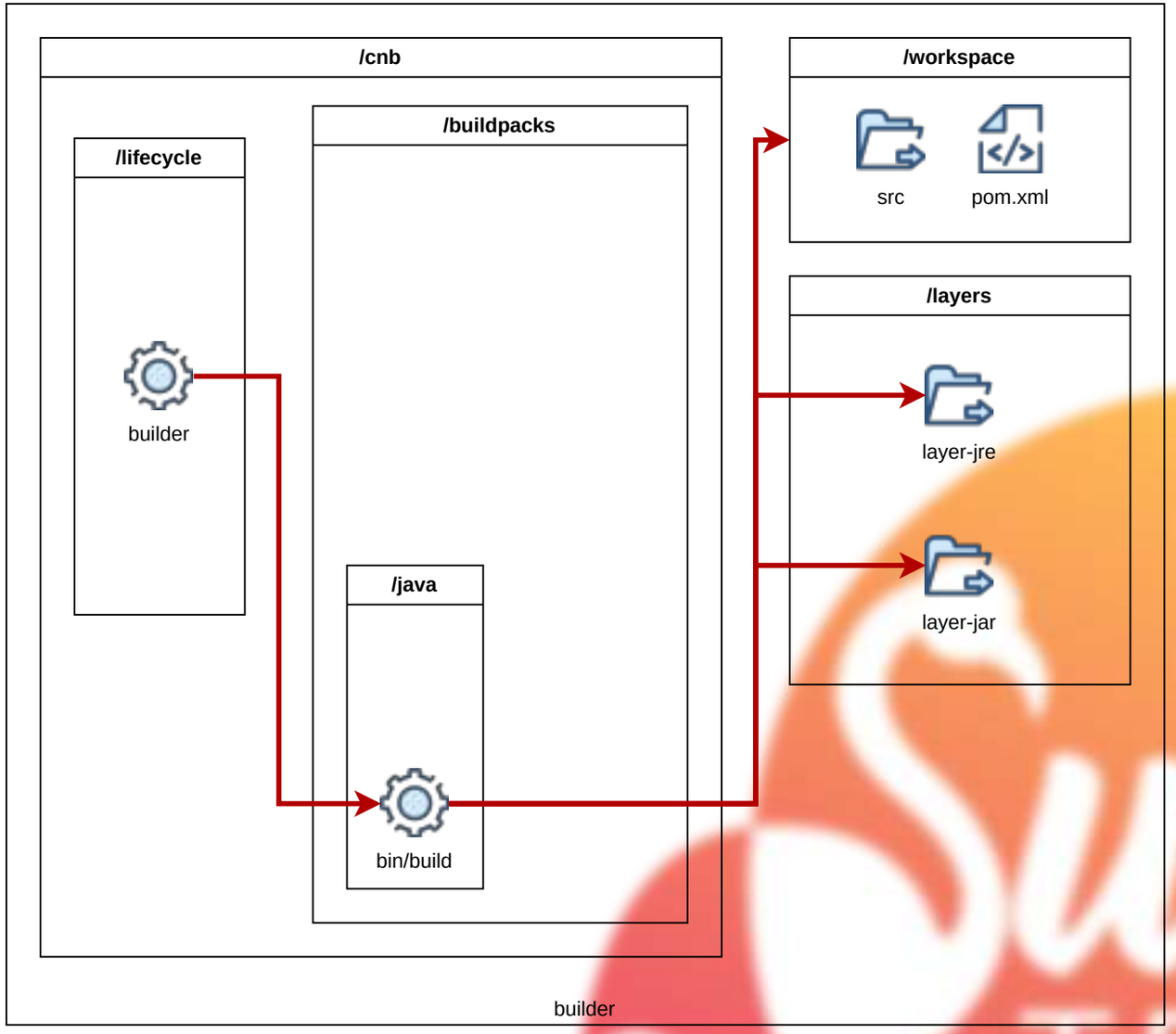

Sunny
TECH

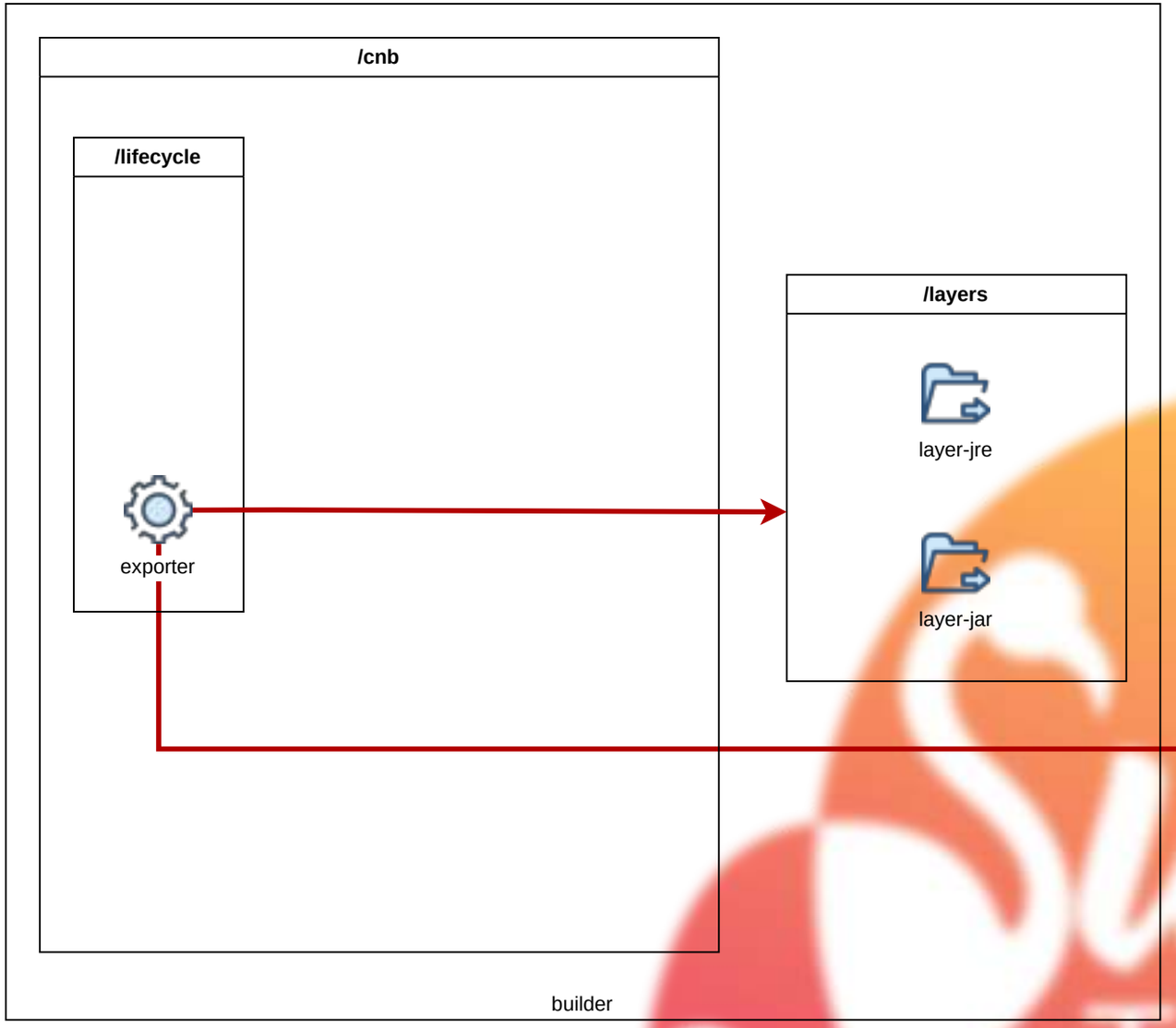

/cnb
/lifecycle
detector
builder
exporter
builder


/cnb
/lifecycle
detector
builder
exporter
/buildpacks
/nodejs
bin/detect
bin/build
/golang
bin/detect
bin/build
/java
bin/detect
bin/build
builder


/cnb
/lifecycle
detector
builder
exporter
/buildpacks
/nodejs
bin/detect
bin/build
/golang
bin/detect
bin/build
/java
bin/detect
bin/build
/workspace
src
pom.xml
builder


/cnb
/lifecycle
detector
/buildpacks
/nodejs
bin/detect
/golang
bin/detect
/java
bin/detect
/workspace
src
pom.xml
builder


/cnb
/lifecycle
builder
/buildpacks
/java
bin/build
/workspace
src
pom.xml
/layers
layer-jre
layer-jar
builder


/cnb
/lifecycle
exporter
/layers
layer-jre
layer-jar
registry
builder

Mais aussi

- Le builder peut charger du cache depuis un registry OCI (`.m2/`, `node_modules/`, ...)
- Les layers peuvent être reproductibles
- Le builder ne s'exécute pas en `root`
- Le builder est une image OCI !

Outillage

Le CLI `pack` permet d'exécuter des builders, pour construire des images.

> bash

```
# installation du CLI avec apt
sudo add-apt-repository ppa:cncf-buildpacks/pack-cli
sudo apt-get update
sudo apt-get install pack-cli
```

> bash

```
# premiers secours
pack --help
```

> bash

```
# construction d'une image OCI !
pack build ma-jolie-image --builder paketobuildpacks/builder:base
```

Les builders de la communauté

Paketo

(Cloud-Foundry / VMWare + Pivotal)

> bash

```
# paketo-base
docker container run --rm -t paketobuildpacks/builder:base ls /cnb/buildpacks
```

> bash

```
# paketo-tiny construit des image distroless
docker container run --rm -t paketobuildpacks/builder:tiny ls /cnb/buildpacks
```

Les builders de la communauté

Heroku

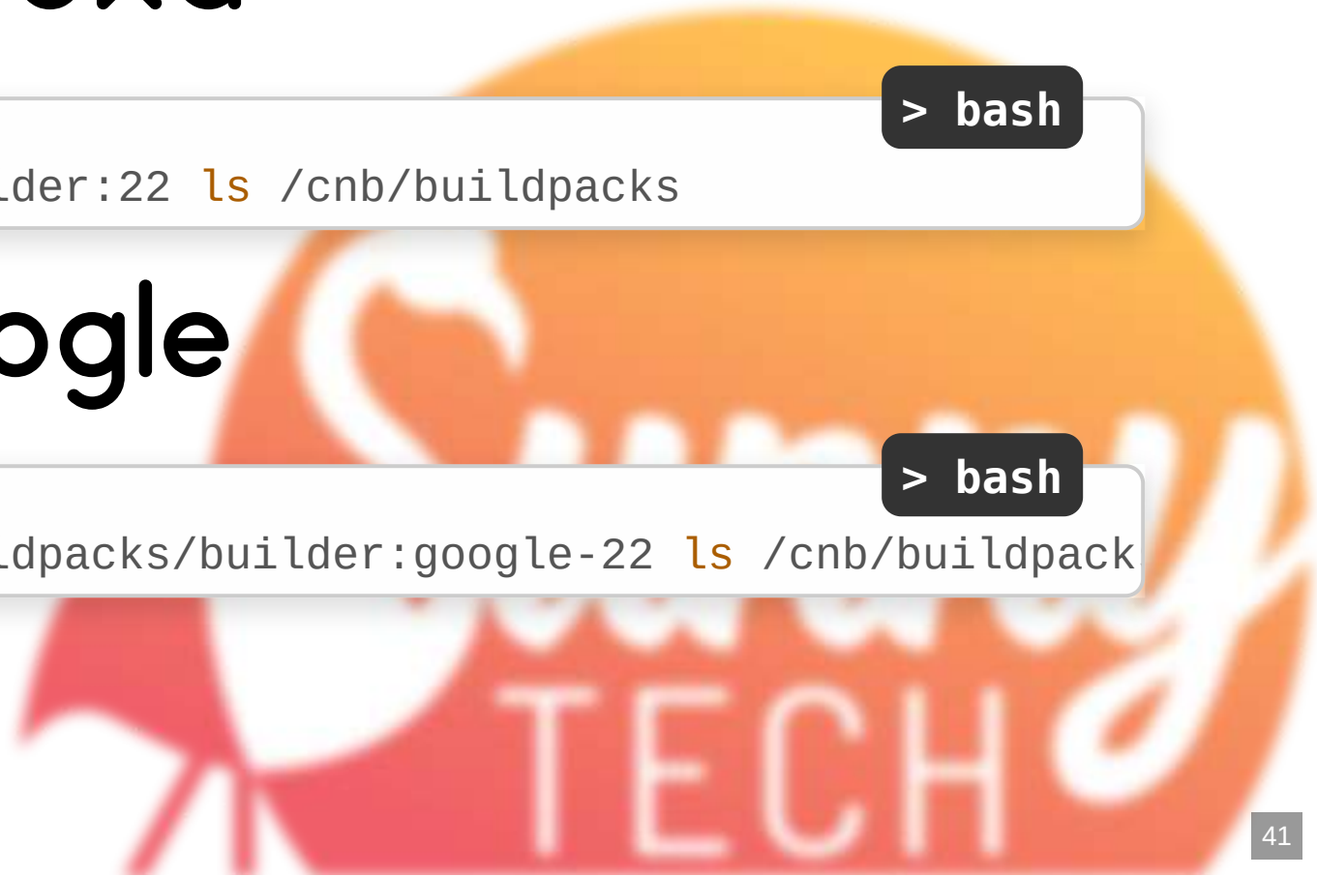
> bash

```
# heroku
docker container run --rm -t heroku/builder:22 ls /cnb/buildpacks
```

Google

> bash

```
# google
docker container run --rm -t gcr.io/buildpacks/builder:google-22 ls /cnb/buildpack
```



Construction d'image !

> bash

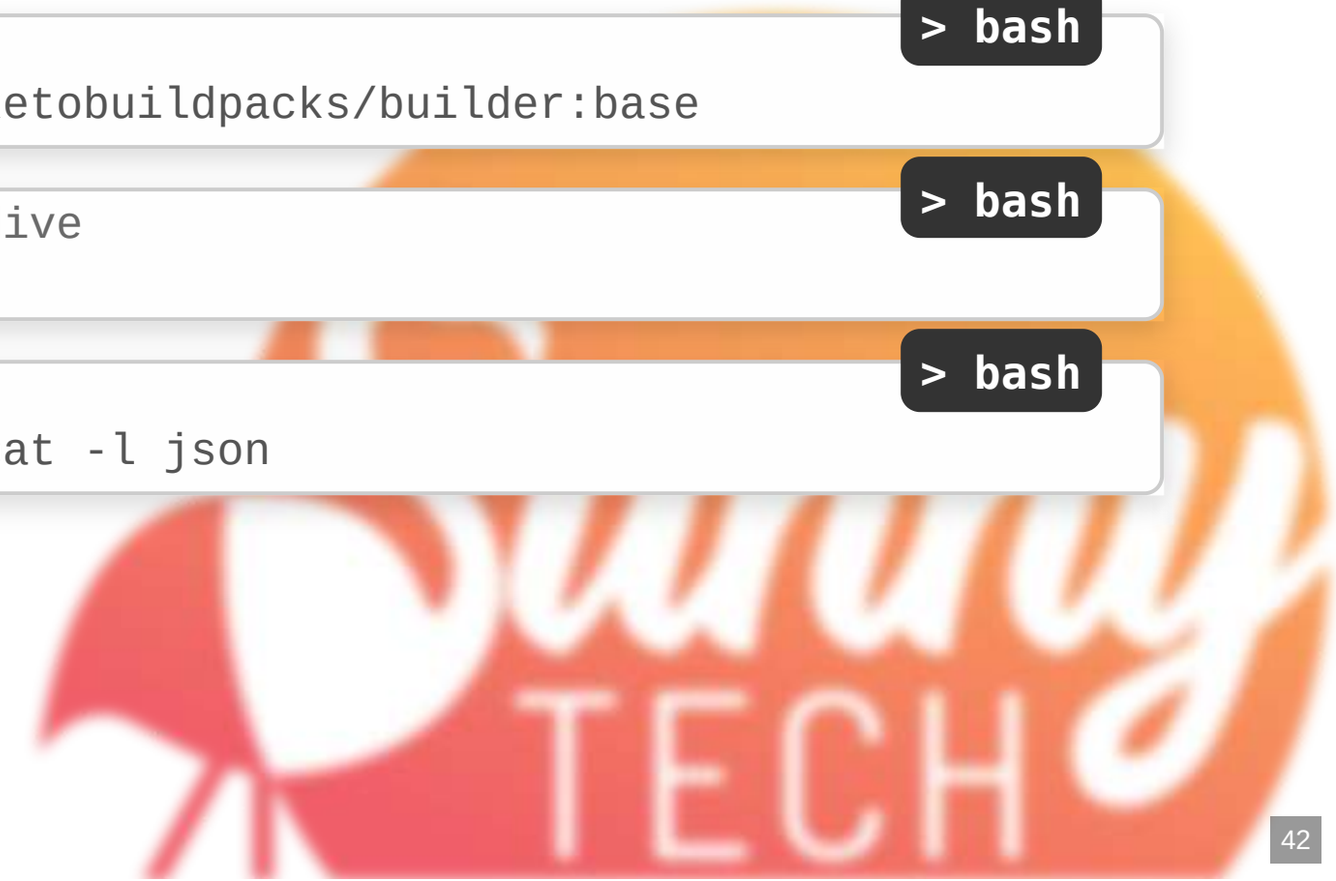
```
# construction d'une image
pack build petclinic:demo --builder paketobuildpacks/builder:base
```

> bash

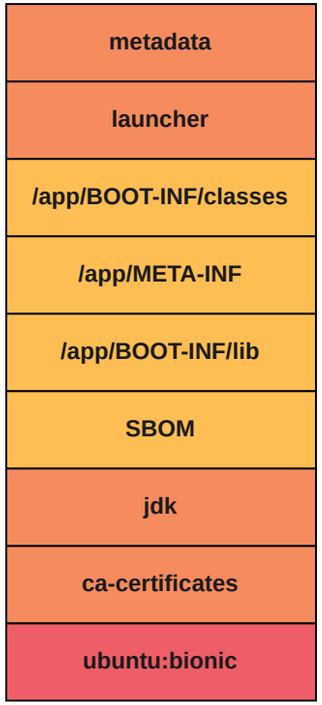
```
# parcours des layers construite avec dive
dive petclinic:demo
```

> bash

```
# inspection de l'image
docker image inspect petclinic:demo | bat -l json
```



L'image produite

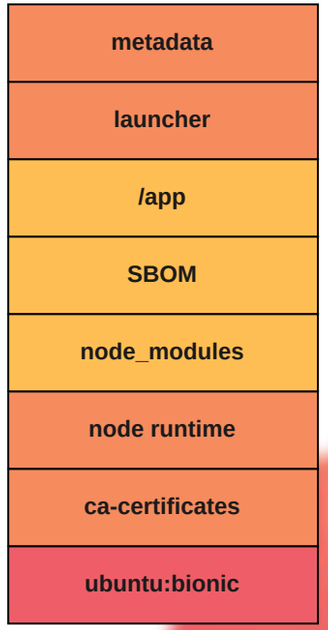


Respecte les bonnes pratiques de layering de Spring Boot

Est plus légère que celle proposée par Spring Boot

```
> bash
REPOSITORY   TAG          IMAGE ID   SIZE    CREATED
petclinic    dockerfile   3c06306a   409MB   7 minute
petclinic    demo         c3448bd3   315MB   43 years
```

Une image produite pour une appli NodeJS



Qui l'utilise en production ? Est-ce que c'est mature ?

Qui l'utilise en production ? Est-ce que c'est mature ?

Google App Engine

Qui l'utilise en production ? Est-ce que c'est mature ?

Google App Engine

Google Cloud Run

Qui l'utilise en production ? Est-ce que c'est mature ?

Google App Engine

Google Cloud Run

Google Cloud Functions

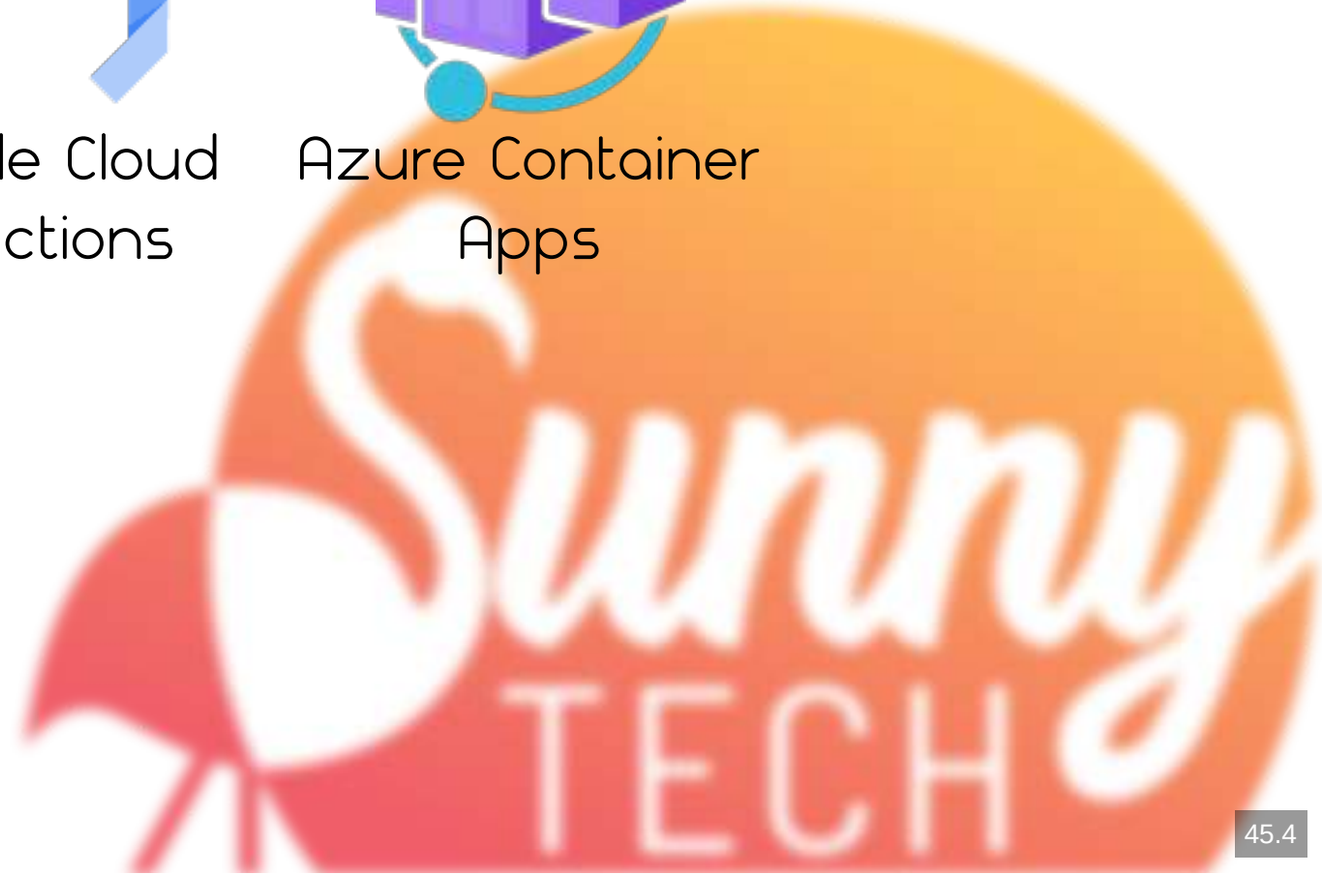
Qui l'utilise en production ? Est-ce que c'est mature ?

Google App Engine

Google Cloud Run

Google Cloud Functions

Azure Container Apps



Qui l'utilise en production ? Est-ce que c'est mature ?

Google App Engine

Google Cloud Run

Google Cloud Functions

Azure Container Apps

Gitlab Auto DevOps

Qui l'utilise en production ? Est-ce que c'est mature ?

Google App Engine

Google Cloud Run

Google Cloud Functions

Azure Container Apps

Gitlab Auto DevOps

KNative

Qui l'utilise en production ? Est-ce que c'est mature ?

Google App Engine

Google Cloud Run

Google Cloud Functions

Azure Container Apps

Gitlab Auto DevOps

KNative

Dokku

Qui l'utilise en production ? Est-ce que c'est mature ?

Google App Engine

Google Cloud Run

Google Cloud Functions

Azure Container Apps

Gitlab Auto DevOps

KNative

Dokku

Spring Boot (mvn spring-

Qui l'utilise en production ? Est-ce que c'est mature ?

Google App Engine

Google Cloud Run

Google Cloud Functions

Azure Container Apps

Gitlab Auto DevOps

KNative

Dokku

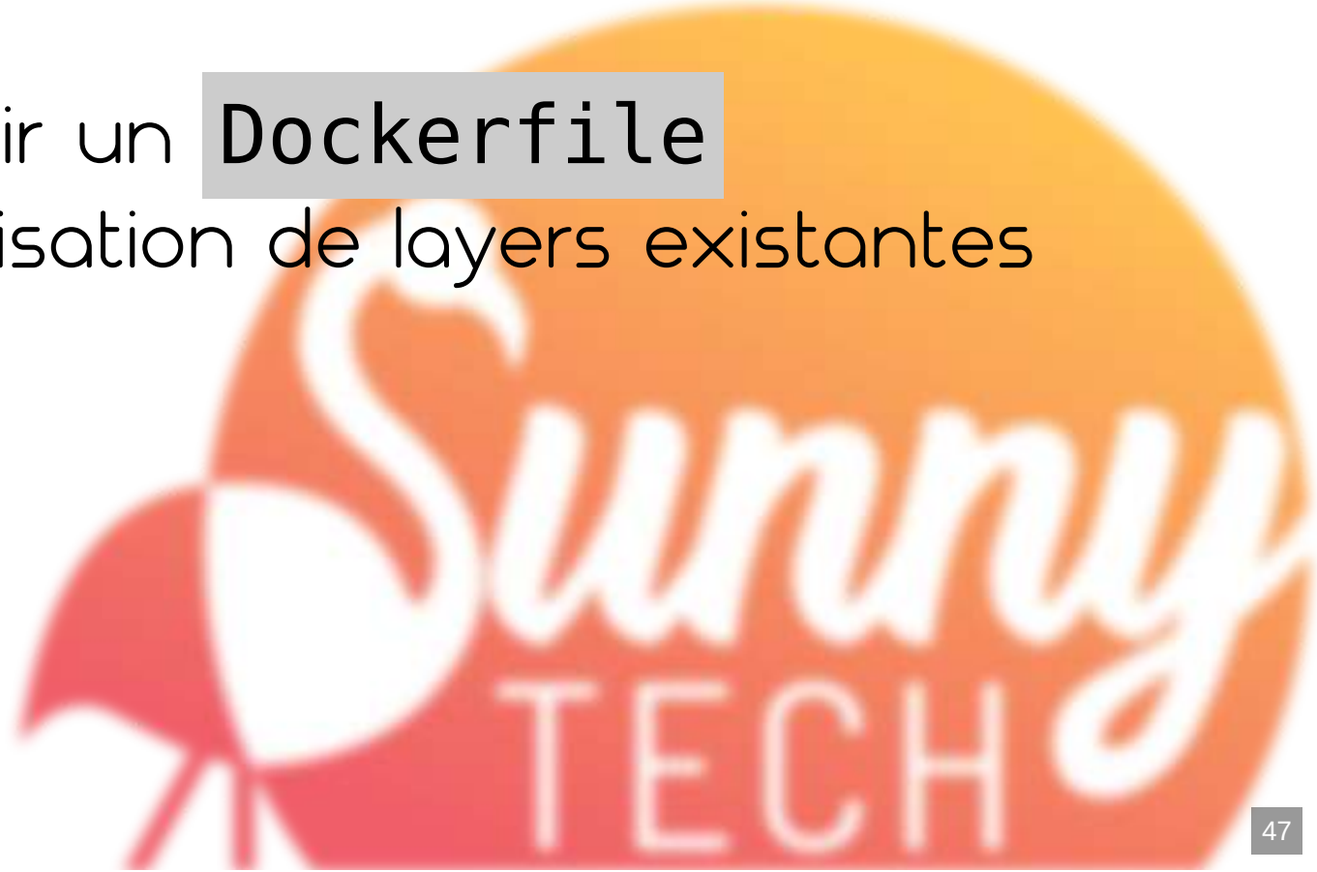
Spring Boot (mvn spring-

Riff (Netlify)

Quels avantages ?

En vrac

- Plus besoin de maintenir un `Dockerfile`
- Gestion de cache, réutilisation de layers existantes
- Modularité



SBoM

Software Bill of Materials

Chaque buildpack contribue à la construction d'un SBoM dans une layer dédiée

C'est vos RSSI et vos RSO qui vont être contents 😀

Reproductibilité des builds ? 🤔

À partir du même code source, produit strictement le même binaire / la même image, avec le même digest `sha256` !

Nécessite de mettre à `"0"` les dates des fichiers.

{} JSON

```
"Created": "1980-01-01T00:00:01Z"
```

> bash

```
REPOSITORY   TAG                 IMAGE ID       CREATED          SIZE
petclinic    demo                c3448bd3501d   43 years ago     315MB
```

Ça évite de produire de "nouvelles" layers si du code n'a pas changé (exemple, les libs)

Reproductibilité des builds ? 🤔

> bash

```
# pull d'une image construite précédemment
docker image pull rg.fr-par.scw.cloud/sunny-tech-buildpacks/petclinic:paketo-base
```

> bash

```
# comparaison des layers
docker image inspect petclinic:demo | jq '.[].RootFS'
docker image inspect rg.fr-par.scw.cloud/sunny-tech-buildpacks/petclinic:paketo-ba

diff -s <(docker image inspect petclinic:demo | jq '.[].RootFS') <(docker image in
```

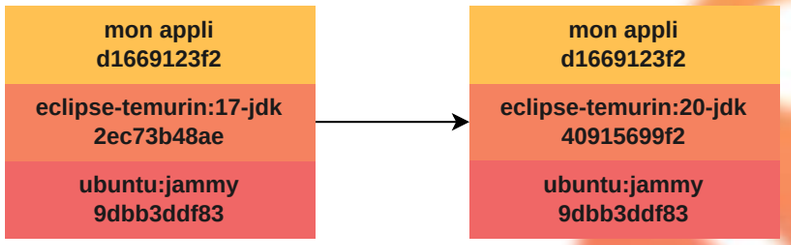
Reproductibilité des builds ? 🤯

Les images buildées sur une CI et sur le poste développeur ont la même signature !

Les images buildées par deux pipelines de CI consécutifs sont identiques !

Le rebase d'image

Oui, ça veut bien dire ce que vous imaginez !



Le rebase d'image

> bash

```
pack rebase petclinic:demo --run-image <new-base-image>
```

Le rebase d'image

Se fait au niveau du ✈️ *Manifest*

Implémenté par la plupart des builders sur la couche "distribution"

Permet de modifier les layers basses d'une image, sans avoir besoin de la reconstruire

Permet de patcher rapidement une image, sans rebuild

Créer son propre builder 

Choisir son image de base

Réutiliser des buildpacks existants

Implémenter les langages manquants

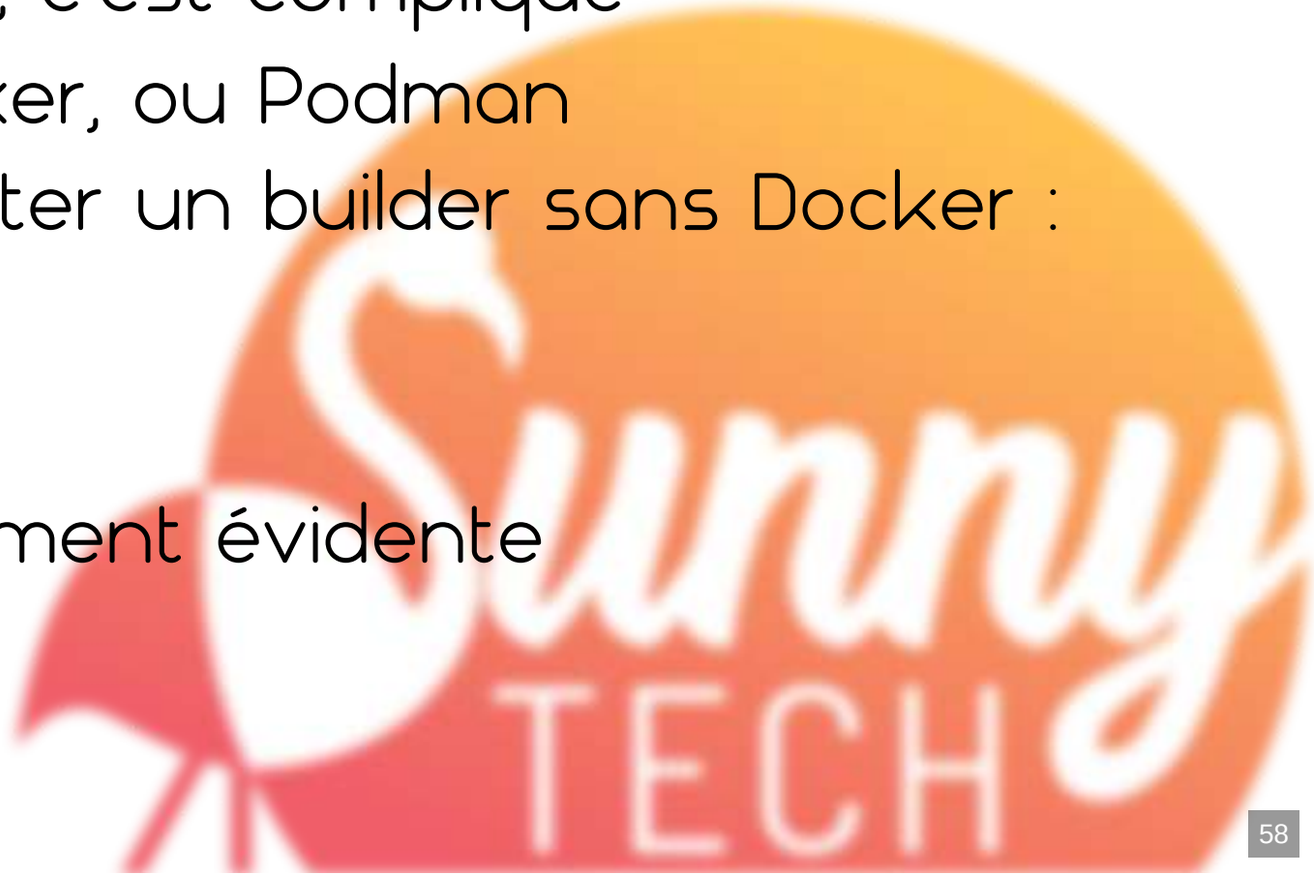
Take-aways

Les buildpacks c'est cool 😎

- Un builder pour supporter tous vos langages
- Génère des SBOM pour votre RSSI/RSO préféré
- Gestion du cache intégrée
- Facile d'utilisation
- Implémente les bonnes pratiques de layering, sécurité
- Reproductibilité des builds 🤯
- Rebase d'images 🤯

Mais... 😢

- Créer son propre builder, c'est compliqué
- Le CLI `pack` utilise Docker, ou Podman
- Mais y'a moyen d'exécuter un builder sans Docker :
  - `kpack`
  - `tekton`
- Customisation pas forcément évidente



Merci pour votre attention !

Un petit feedback ? =>

Julien Wittouck – @CodeKaio