


Laissez tomber vos Dockerfile,
adoptez un Buildpack !

Julien Wittouck –  @CodeKaio

Vous êtes au bon endroit si

- Vous êtes un·e Dev 
- Vous êtes 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

```
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"]
```

 Dockerfile

Salut 🖐️

Moi c'est Julien

Freelance @CodeKaio

Associé @Ekité

Teacher @univ-lille

🧑 Speaker (DevFest Lille –
Sunny Tech – Cloud Nord)

🔍 Reviewer Cloud Nord



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 27 may 2022 🤮

www.docker.com/blog

 Dockerfile

```
1 FROM eclipse-temurin:17-jdk-focal
2
3 WORKDIR /app
4
5 COPY .mvn/ .mvn
6 COPY mvnw pom.xml ./
7 RUN ./mvnw dependency:go-offline
8
9 COPY src ./src
10
11 CMD [ "./mvnw", "spring-boot:run" ]
```

Le *Hello World* Spring Boot, selon Docker 27 may 2022 🤮

www.docker.com/blog

 Dockerfile

```
1 FROM eclipse-temurin:17-jdk-focal
2
3 WORKDIR /app
4
5 COPY .mvn/ .mvn
6 COPY mvnw pom.xml ./
7 RUN ./mvnw dependency:go-offline
8
9 COPY src ./src
10
11 CMD [ "./mvnw", "spring-boot:run" ]
```

Le *Hello World* Spring Boot, selon Docker 27 may 2022 🤮

www.docker.com/blog

 Dockerfile

```
1 FROM eclipse-temurin:17-jdk-focal
2
3 WORKDIR /app
4
5 COPY .mvn/ .mvn
6 COPY mvnw pom.xml ./
7 RUN ./mvnw dependency:go-offline
8
9 COPY src ./src
10
11 CMD [ "./mvnw", "spring-boot:run" ]
```

Le *Hello World* Spring Boot, selon Docker 27 may 2022 🤮

www.docker.com/blog

 Dockerfile

```
1 FROM eclipse-temurin:17-jdk-focal
2
3 WORKDIR /app
4
5 COPY .mvn/ .mvn
6 COPY mvnw pom.xml ./
7 RUN ./mvnw dependency:go-offline
8
9 COPY src ./src
10
11 CMD [ "./mvnw", "spring-boot:run" ]
```

Le *Hello World* Spring Boot, selon Docker 27 may 2022 🤮

www.docker.com/blog

 Dockerfile

```
1 FROM eclipse-temurin:17-jdk-focal
2
3 WORKDIR /app
4
5 COPY .mvn/ .mvn
6 COPY mvnw pom.xml ./
7 RUN ./mvnw dependency:go-offline
8
9 COPY src ./src
10
11 CMD [ "./mvnw", "spring-boot:run" ]
```

Le *Hello World* Spring Boot, selon Spring Boot 8 feb 2023 🤔

spring.io/guides

 Dockerfile

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


Le *Hello World* Spring Boot, selon Spring Boot 8 feb 2023 🤔

spring.io/guides

 Dockerfile

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

Le *Hello World* Spring Boot, selon Spring Boot 8 feb 2023 🤔

spring.io/guides

 Dockerfile

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

Le *Hello World* Spring Boot, selon Spring Boot 8 feb 2023 🤔

spring.io/guides

 Dockerfile

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

Le *Hello World* Spring Boot, selon Spring Boot 8 feb 2023 🤔

spring.io/guides

 Dockerfile

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

Le *Hello World* Spring Boot, selon Spring Boot 8 feb 2023 🤔

spring.io/guides

 Dockerfile

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

Le *Hello World* Spring Boot, selon Spring Boot 8 feb 2023 🤔

spring.io/guides

 Dockerfile

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




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 ?

Darren Shepherd  @ibuildthecloud

(former @Rancher_Labs, k3s creator)

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*

You really can't build a properly good docker image without multi stage because you don't want your compiler in the final image. < | >

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


Les images Docker

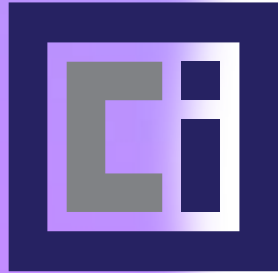


Images 🐳 Docker ?

Images  ~~Docker~~ ?

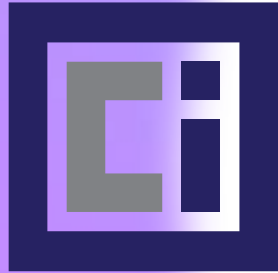
Images  ~~Docker~~ OCI ?

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 :

mon-appli:1 d1669123f2
eclipse-temurin:17-jdk 2ec73b48ae
ubuntu:jammy 9dbb3ddf83

distrib + runtime/middleware + code

JSON + tar.gz = ❤️

✈️ manifest.json

```
1 {
2   "schemaVersion": 2,
3   "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
4   "layers": [
5     {
6       "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
7       "size": 30430275,
8       "digest": "sha256:d1669123f28121211977ed38e663dca1a397c0c001e5386598b96c8
9     },
10    {
11      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
12      "size": 17038759,
13      "digest": "sha256:2ec73b48ae406646223453ca41d5d6b7cb739853fb7a44f15d35a31
14    },
15  ]
}
```

On appelle ça un *Manifest* d'image

JSON + tar.gz = ❤️

 manifest.json

```
1 {
2   "schemaVersion": 2,
3   "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
4   "layers": [
5     {
6       "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
7       "size": 30430275,
8       "digest": "sha256:d1669123f28121211977ed38e663dca1a397c0c001e5386598b96c8
9     },
10    {
11      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
12      "size": 17038759,
13      "digest": "sha256:2ec73b48ae406646223453ca41d5d6b7cb739853fb7a44f15d35a31
14    },
15  ]
}
```

On appelle ça un *Manifest* d'image

JSON + tar.gz = ❤️

✈️ manifest.json

```
1 {
2   "schemaVersion": 2,
3   "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
4   "layers": [
5     {
6       "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
7       "size": 30430275,
8       "digest": "sha256:d1669123f28121211977ed38e663dca1a397c0c001e5386598b96c8
9     },
10    {
11      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
12      "size": 17038759,
13      "digest": "sha256:2ec73b48ae406646223453ca41d5d6b7cb739853fb7a44f15d35a31
14    },
15  ]
}
```

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:17.0.8.1_1-jdk

 config.json

```
1 {
2   "architecture": "amd64",
3   "config": {
4     "Hostname": "",
5     "Domainname": "",
6     "User": "",
7     "AttachStdin": false,
8     "AttachStdout": false,
9     "AttachStderr": false,
10    "Tty": false,
11    "OpenStdin": false,
12    "StdinOnce": false,
13    "Env": [
14      "PATH=/opt/java/openjdk/bin:/usr/local/sbin:/usr/local/bin:/usr",
15      "JAVA_HOME=/opt/java/openjdk"
```

The background of the slide features a bronze statue of 'The Thinker' by Auguste Rodin, depicting a man in a state of deep contemplation, sitting on a stone block with his chin resting on his hand. The statue is set against a blurred background of a stone wall.

L'instant philosophie

Heureusement qu'ils n'ont pas choisi YAML

docker image inspect



```
docker image inspect eclipse-temurin:17.0.8.1_1-jdk | bat -l json
```

> **bash**



Une image OCI

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

L'instant philosophie

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

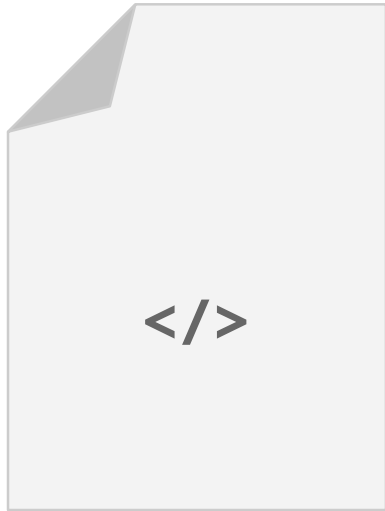
ça veut aussi dire, qu'on peut modifier une image, juste en allant modifier son manifest

Les buildpacks

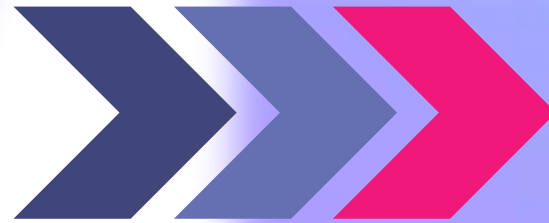




Buildpacks.io



source



without Dockerfiles

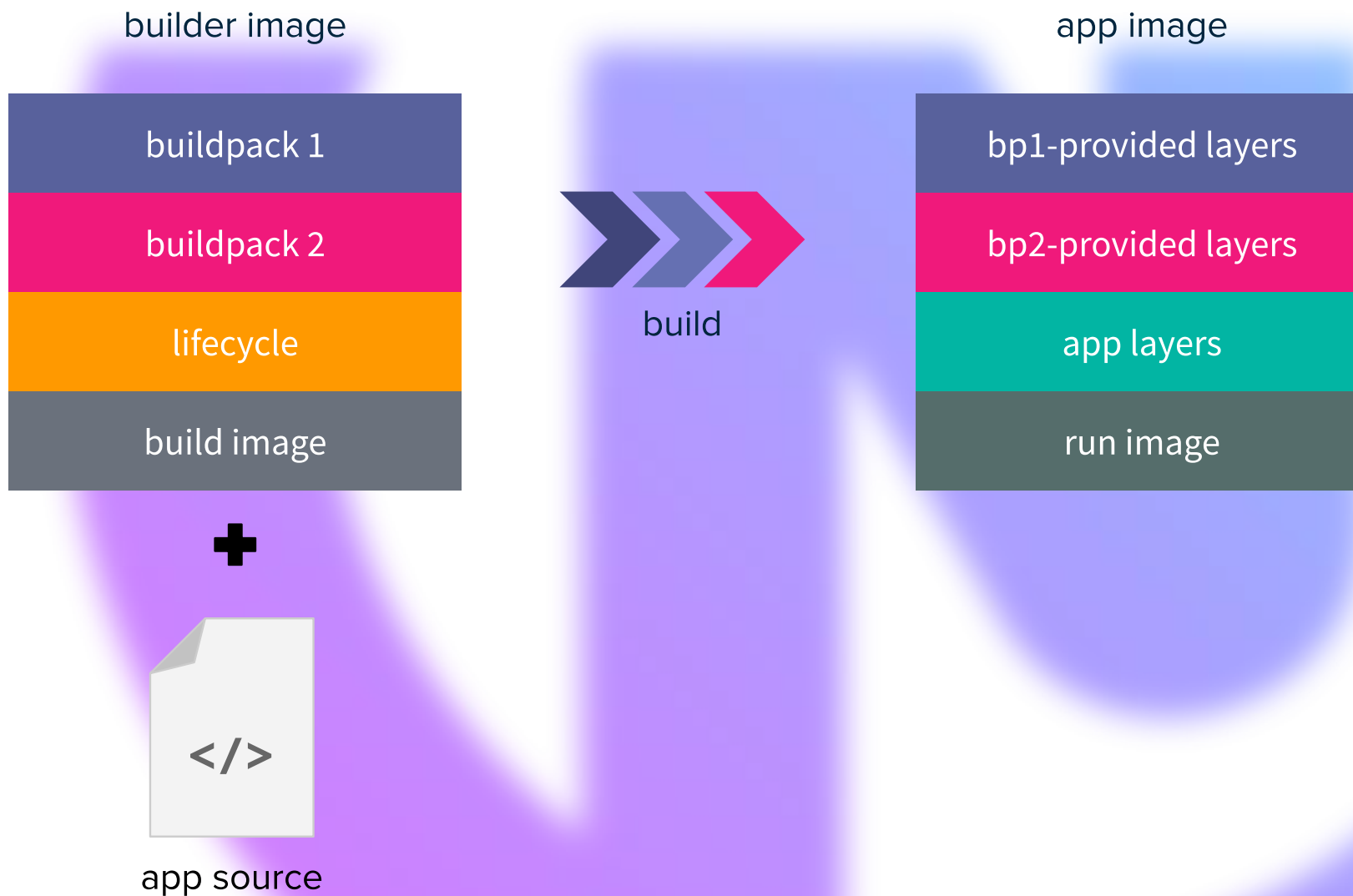


OCI image

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 ?



UN

builder

construit une image

buildpack

contribue à une ou plusieurs layers dans la construction

build image

image de base utilisée par le builder

run image

image de base des futures images applicatives

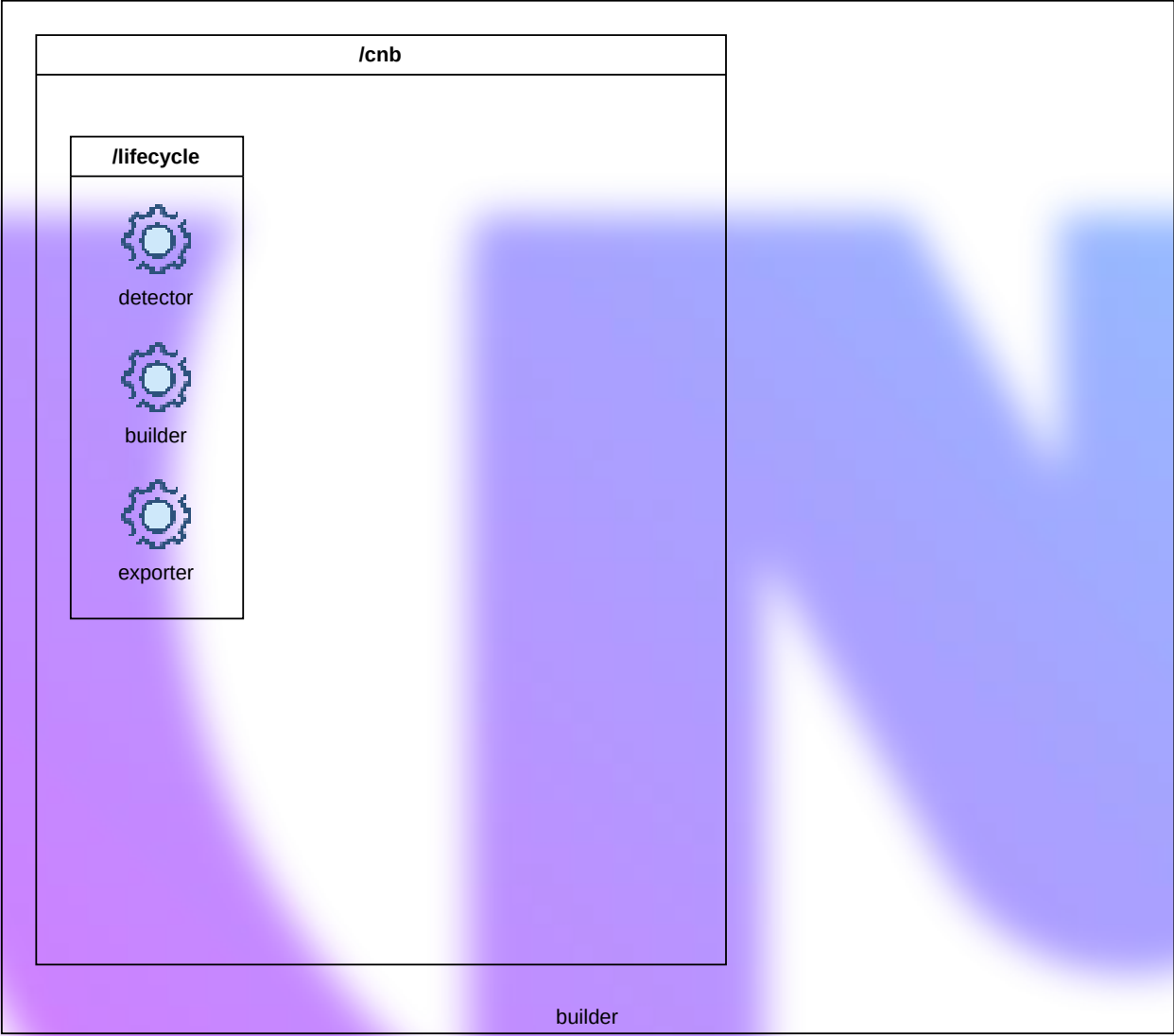
Un buildpack est composé de 2 binaires :

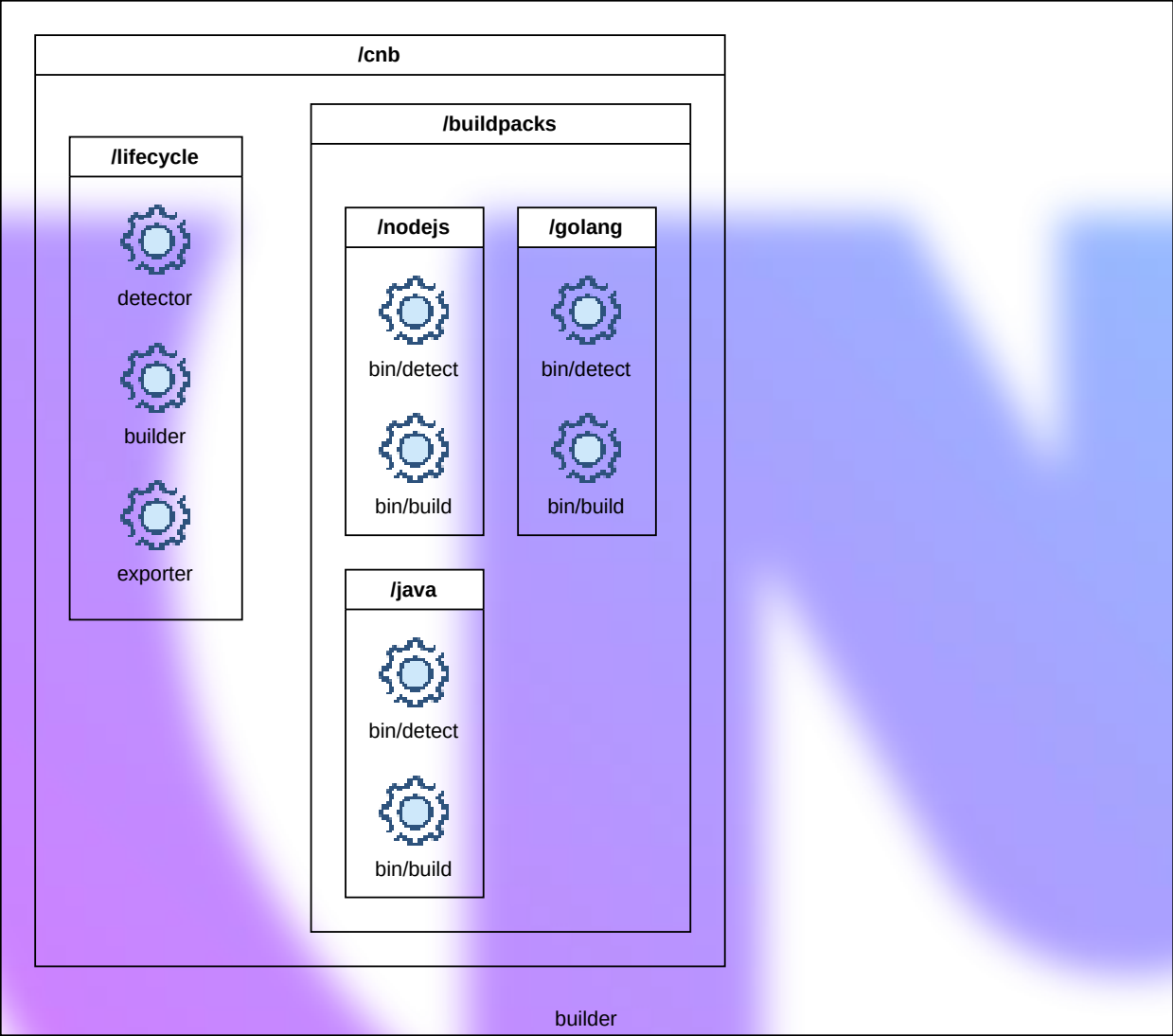
- `/bin/detect` : indique si le buildpack doit être activé
- `/bin/build` : contribue à la construction d'une ou plusieurs layers

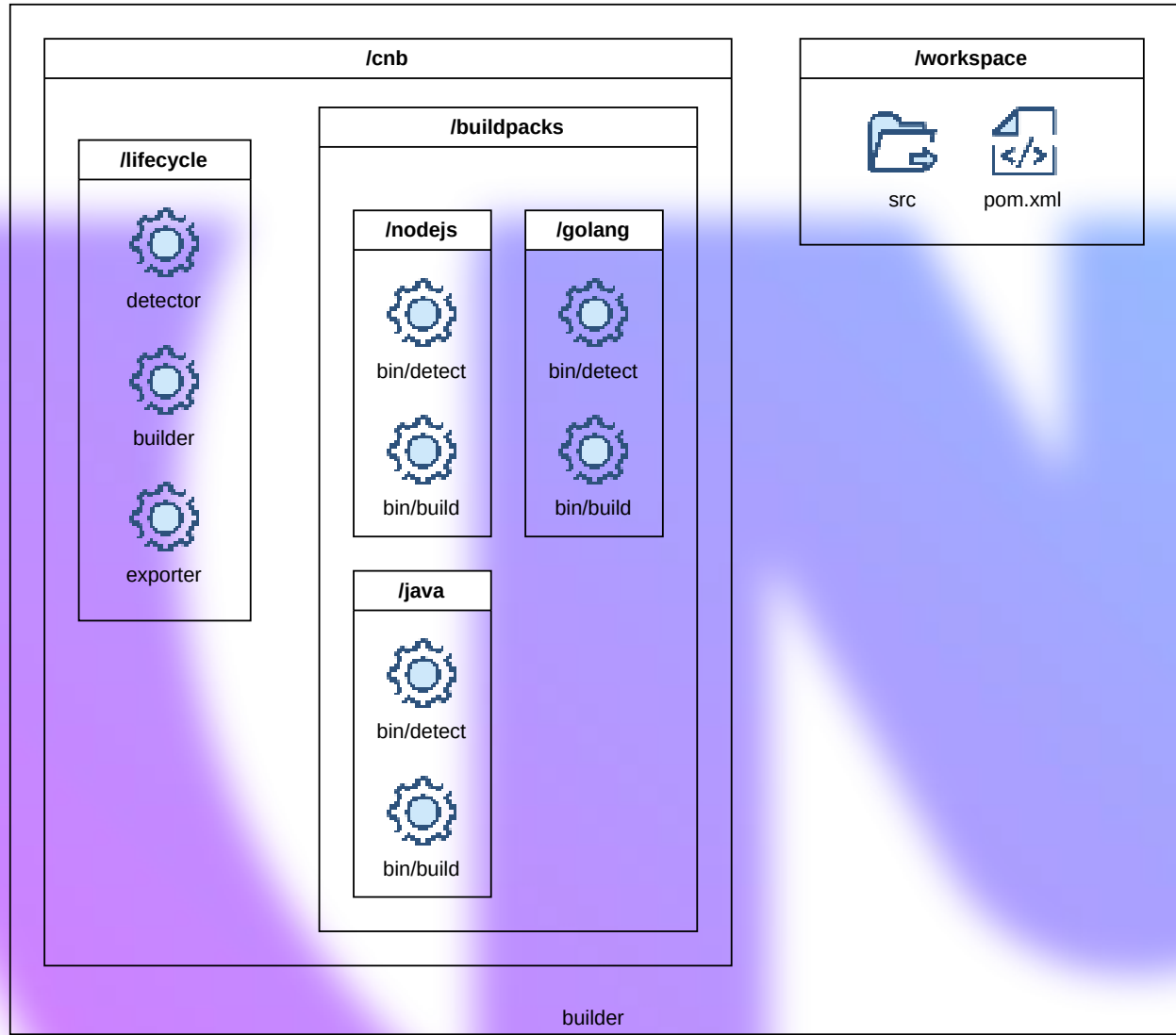
Le builder :

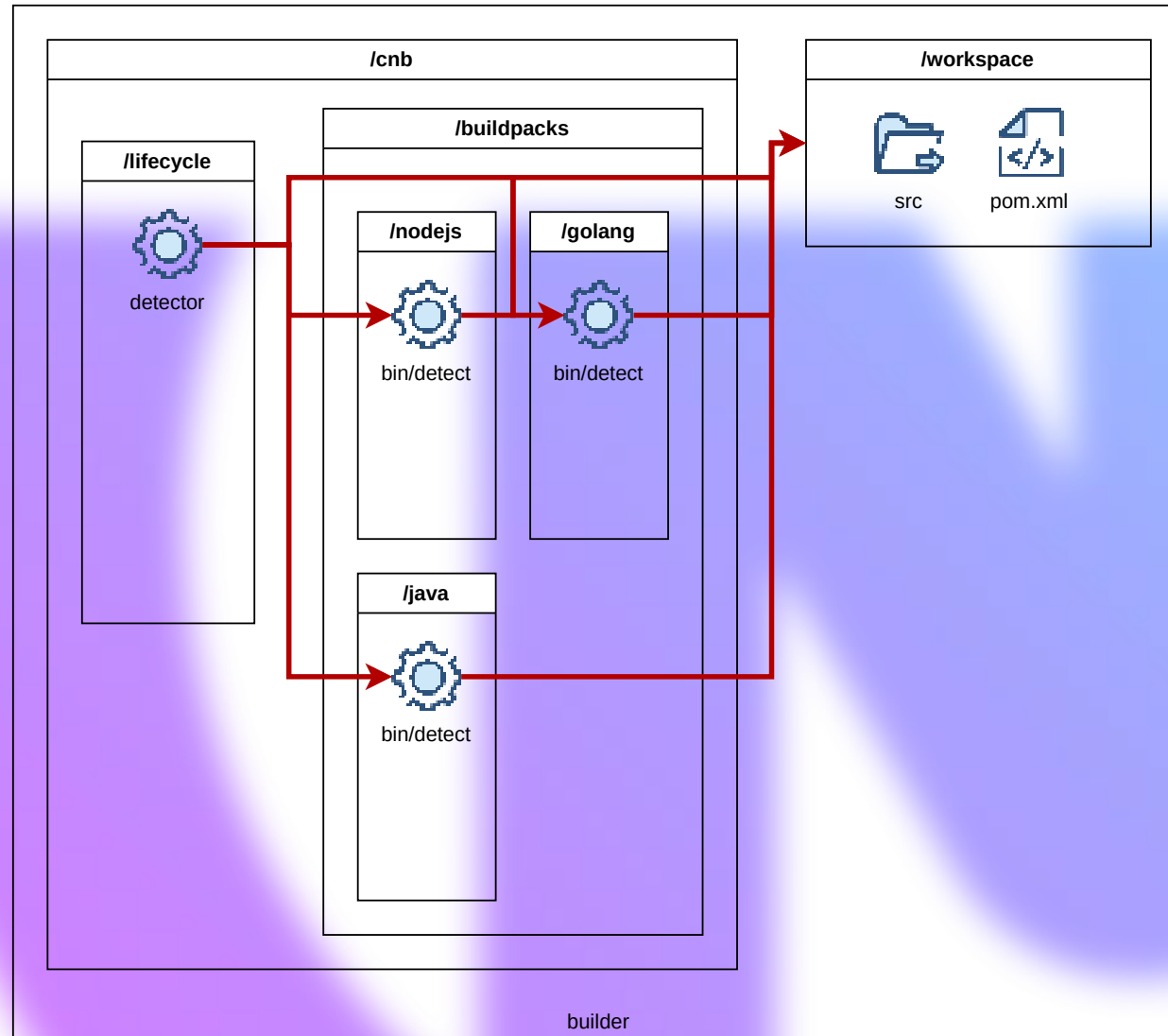
1. Code source dans un répertoire `/workspace`
2. Interroge chaque buildpack avec `/bin/detect`
3. Exécute tous les buildpacks qui doivent être exécutés avec `/bin/build`
4. Chaque buildpack contribue une ou plusieurs layers dans `/layer`
5. Les layers dans `/layer` sont exportées pour créer une image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

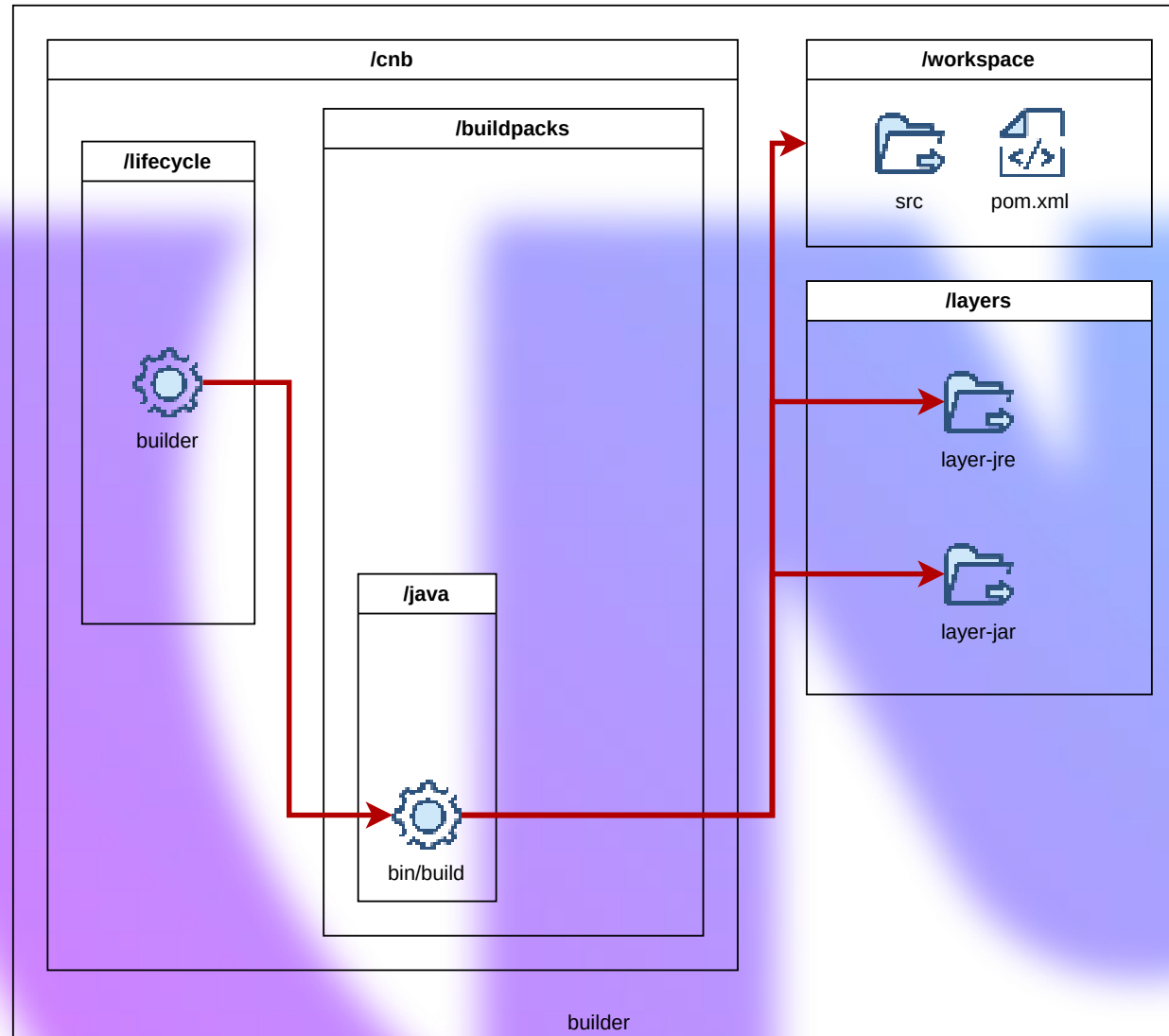
UN

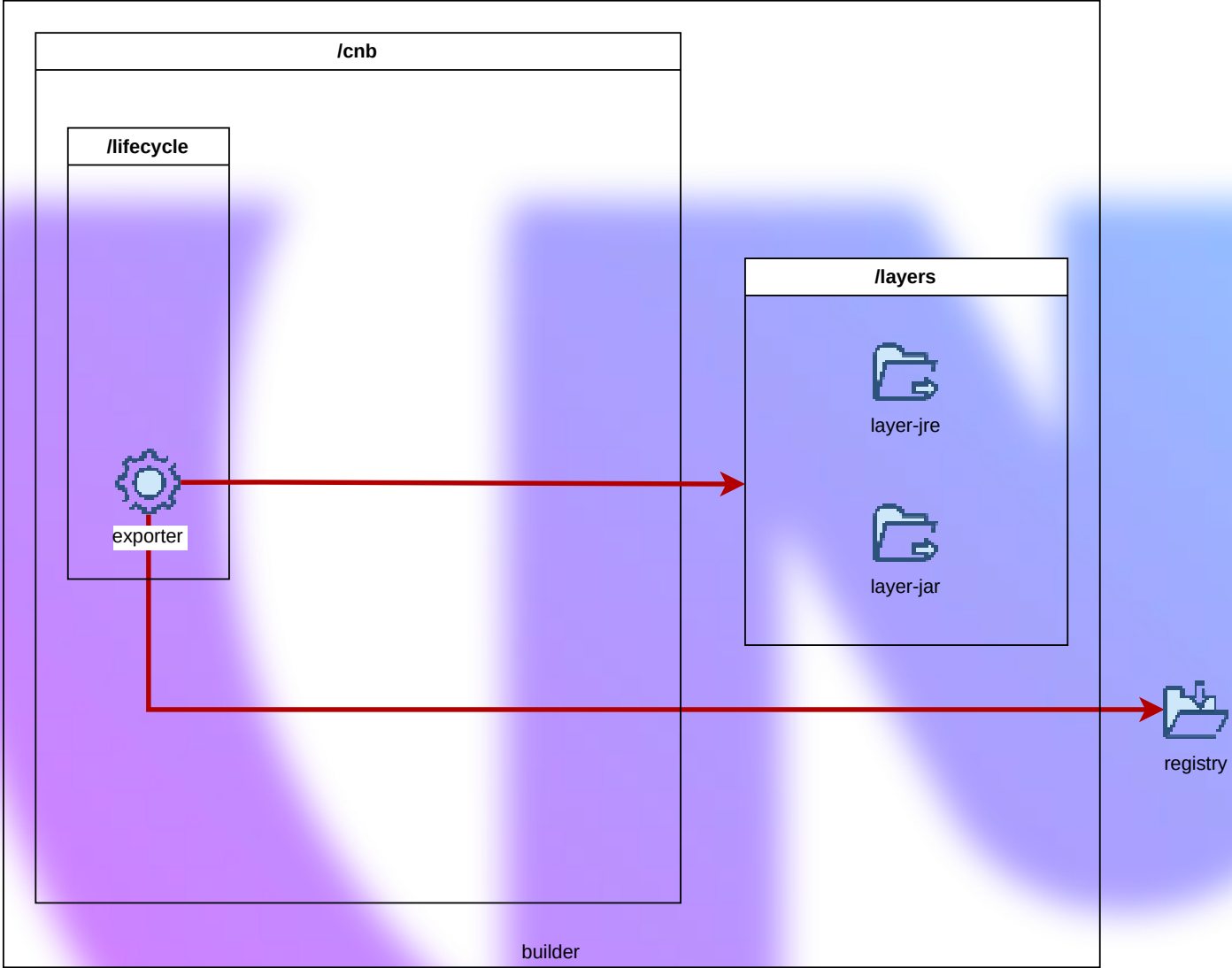












Mais aussi

- 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.

```
# 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
```

> bash

Les builders de la communauté

Paketo

(Cloud-Foundry / VMWare + Pivotal)

```
# 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
```

> bash

Les builders de la communauté

Heroku

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

> bash

Google

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

> bash

Construction d'image !

```
# 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
```

> bash

L'image produite

metadata
launcher
/app/BOOT-INF/classes
/app/META-INF
/app/BOOT-INF/lib
SBOM
jdk
ca-certificates
ubuntu:bionic

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 minutes
petclinic	demo	c3448bd3	315MB	43 years

Une image produite pour une appli NodeJS

metadata
launcher
/app
SBOM
node_modules
node runtime
ca-certificates
ubuntu:bionic

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



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

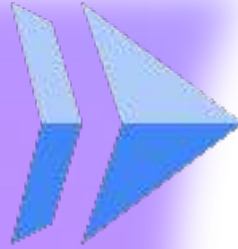


Google App
Engine

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



Google App
Engine

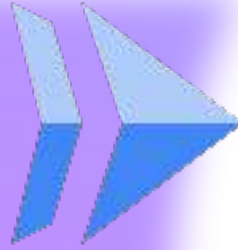


Google Cloud
Run

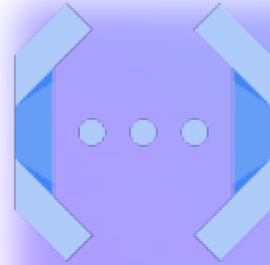
Qui l'utilise en production ? Est-ce que c'est mature ?



Google App Engine



Google Cloud Run

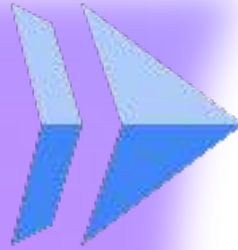


Google Cloud Functions

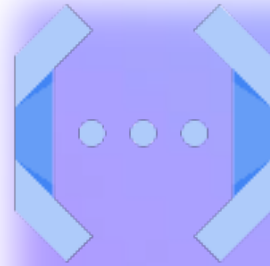
Qui l'utilise en production ? Est-ce que c'est mature ?



Google App Engine



Google Cloud Run



Google Cloud Functions

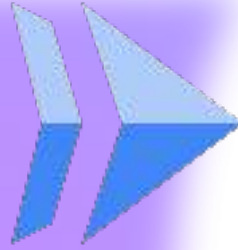


Azure Container Apps

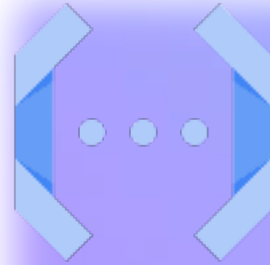
Qui l'utilise en production ? Est-ce que c'est mature ?



Google App Engine



Google Cloud Run



Google Cloud Functions



Azure Container Apps

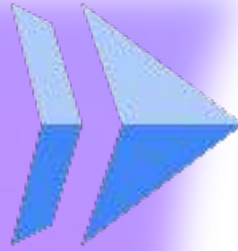


Gitlab Auto DevOps

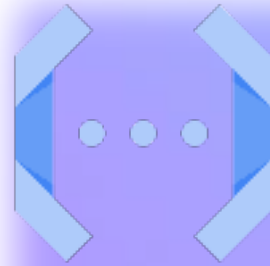
Qui l'utilise en production ? Est-ce que c'est mature ?



Google App Engine



Google Cloud Run



Google Cloud Functions



Azure Container Apps



Gitlab Auto DevOps



KNative

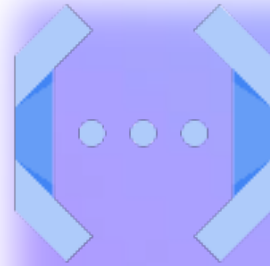
Qui l'utilise en production ? Est-ce que c'est mature ?



Google App Engine



Google Cloud Run



Google Cloud Functions



Azure Container Apps



Gitlab Auto DevOps



KNative

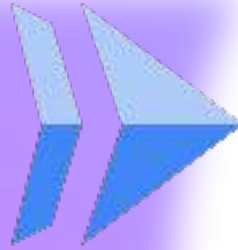


Dokku

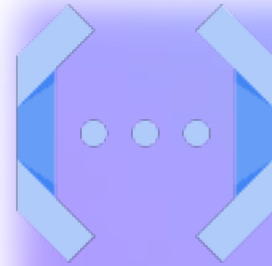
Qui l'utilise en production ? Est-ce que c'est mature ?



Google App Engine



Google Cloud Run



Google Cloud Functions



Azure Container Apps



Gitlab Auto DevOps



KNative



Dokku

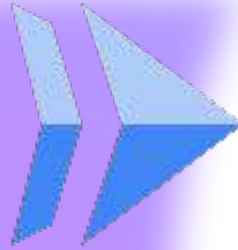


Spring Boot
(`mvn spring-boot:build-image`)

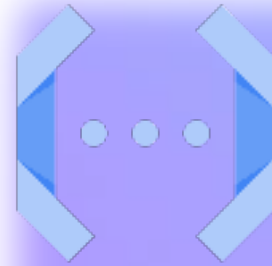
Qui l'utilise en production ? Est-ce que c'est mature ?



Google App Engine



Google Cloud Run



Google Cloud Functions



Azure Container Apps



Gitlab Auto DevOps



KNative



Dokku



Spring Boot
(`mvn spring-boot:build-image`)



Riff

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 RSSO qui vont être contents 😊

```
# extraction du SBOM  
pack sbom download petclinic:demo -o sbom  
tree sbom
```

> bash

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.

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

{ JSON

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

> bash

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

Reproductibilité des builds ? 🤔

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

> bash

```
# comparaison des layers  
docker image inspect petclinic:demo | jq '.[].RootFS'  
docker image inspect rg.fr-par.scw.cloud/cloud-nord-buildpacks/petclinic:paketo-base | jq '.[].RootFS'  
  
diff -s <(docker image inspect petclinic:demo | jq '.[].RootFS') <(docker image inspect rg.fr-par.scw.cloud/cloud-nord-buildpacks/petclinic:paketo-base | jq '.[].RootFS')
```

> bash

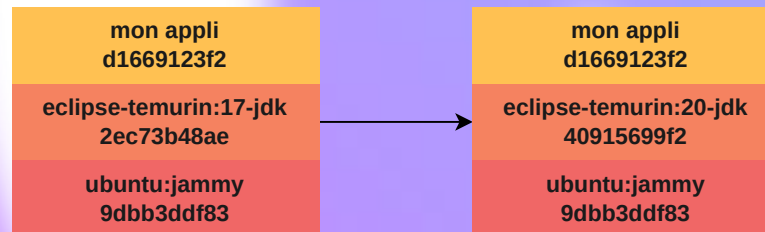
Reproductibilité des builds ? 🤖

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

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

> bash

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

Extensions des buildpacks



Permet d'étendre des builder / buildpacks sans avoir besoin de les modifier !

Installer un binaire dans la 'Build' ou 'Run' image

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
- Pas de support des architectures type ARM (aucun builder disponible)
- 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

