Zurich University
of Applied Sciences

School of
Engineering
InIT Institute of Applied
Information Technology

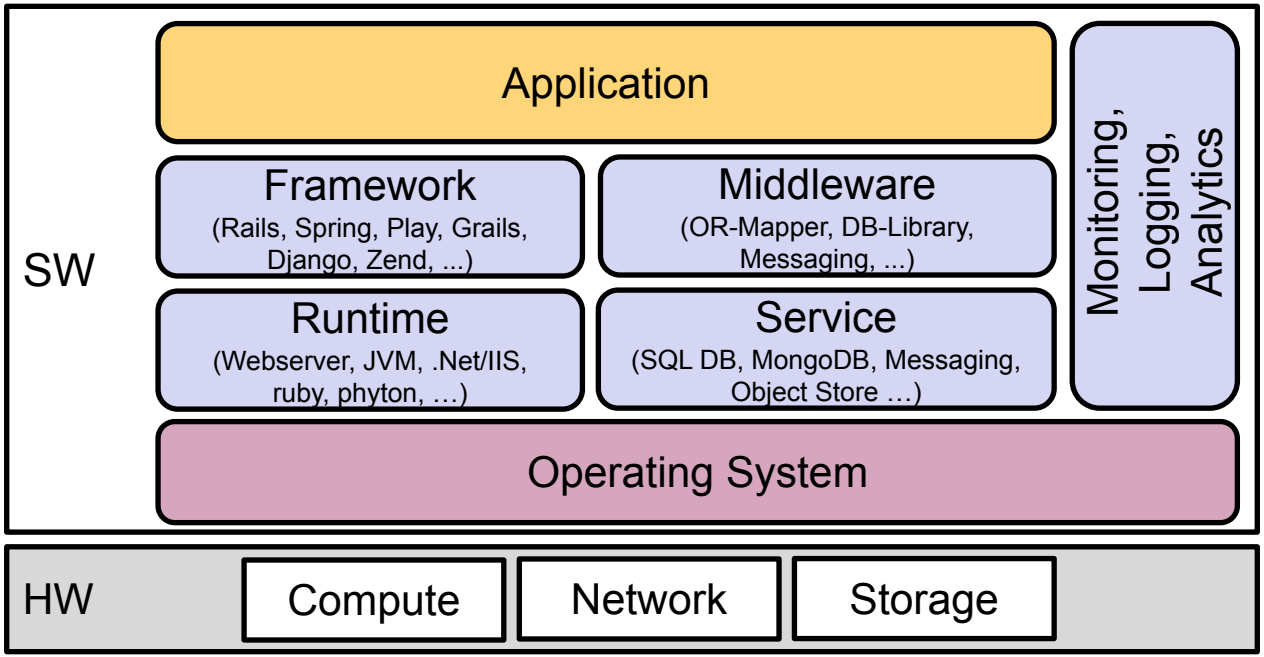
RUNT - Runtime Environments

Prof. Dr. Thomas M. Bohnert
Christof Marti

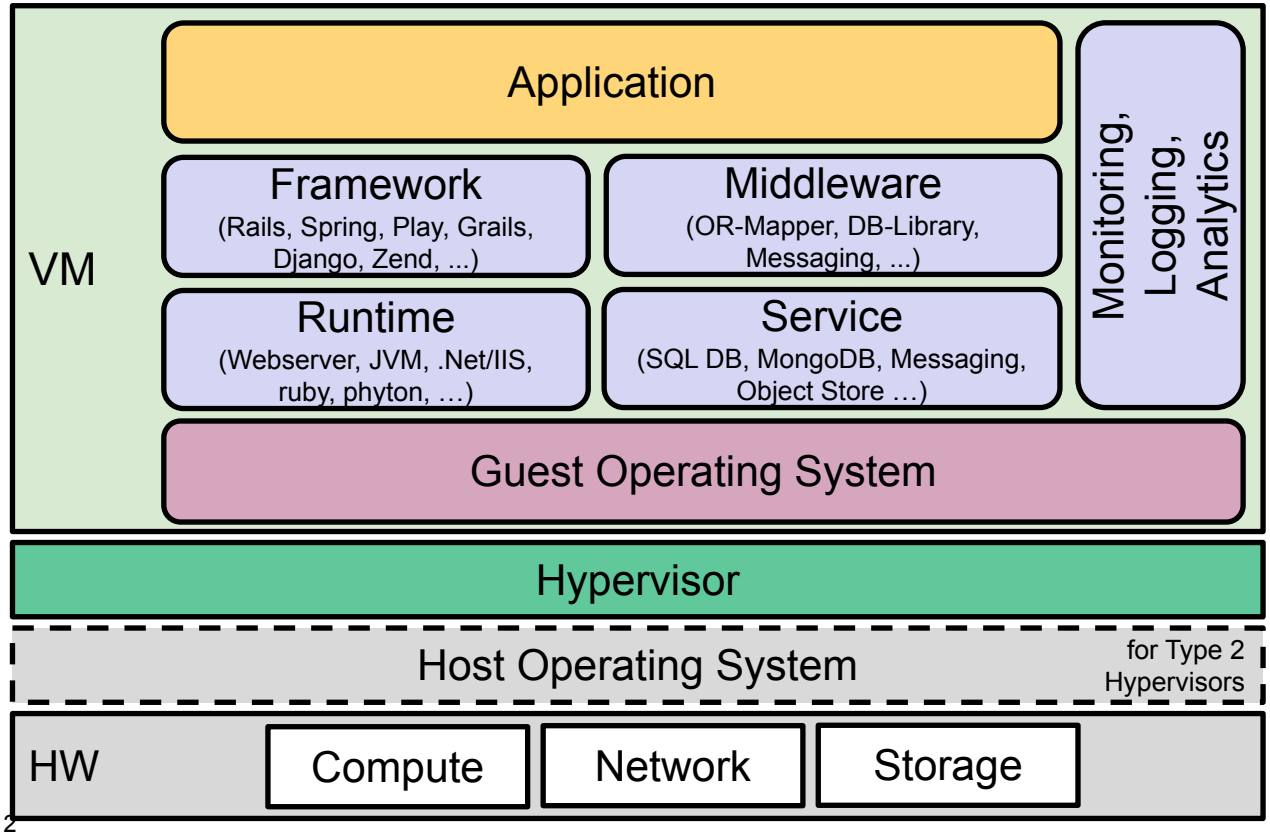
Content

- Application Stacks
- Containers and Runtimes
- Docker images, Multi-Stage images
- Buildpacks - the Heroku / CloudFoundry / CNCF way

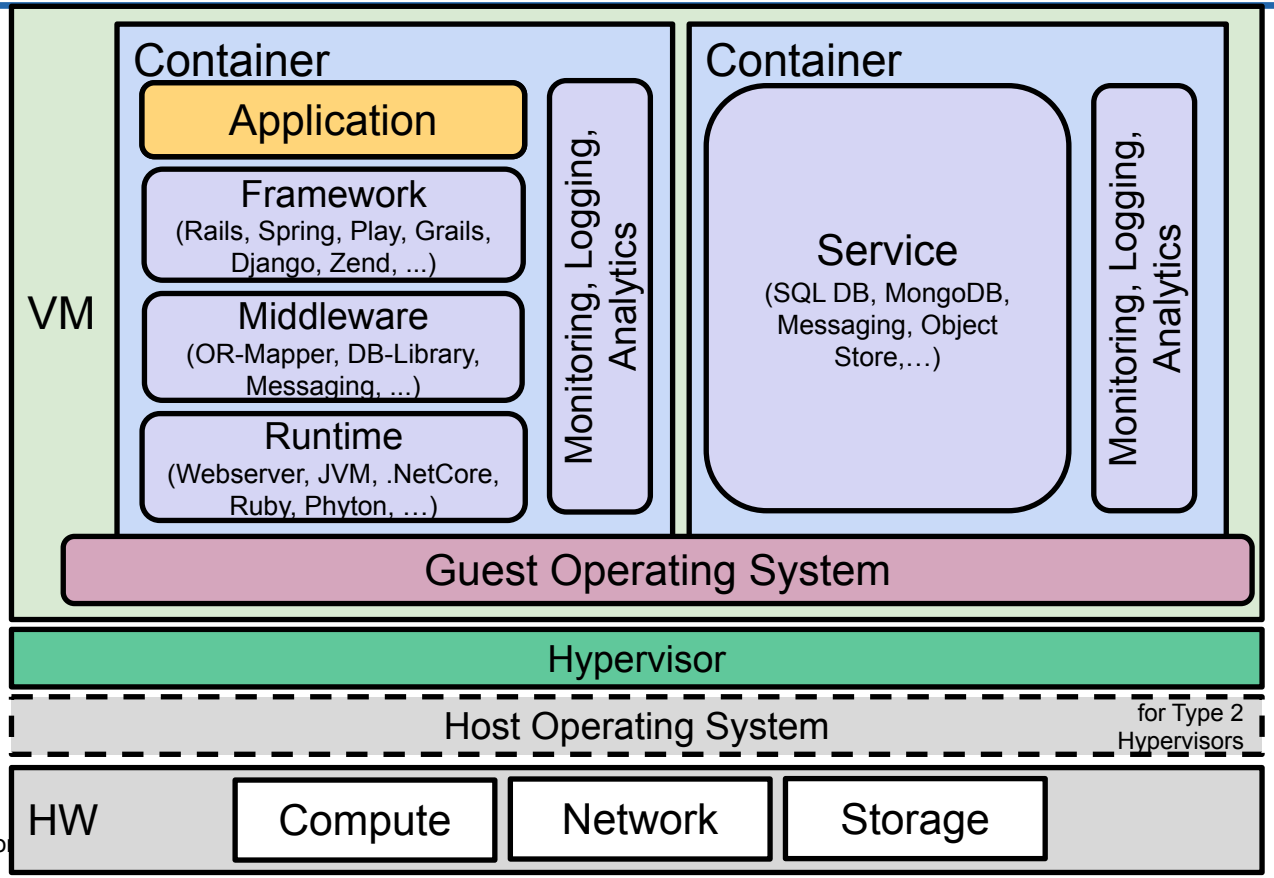
On Premise Application Stack



HW-Virtualized Application Stack



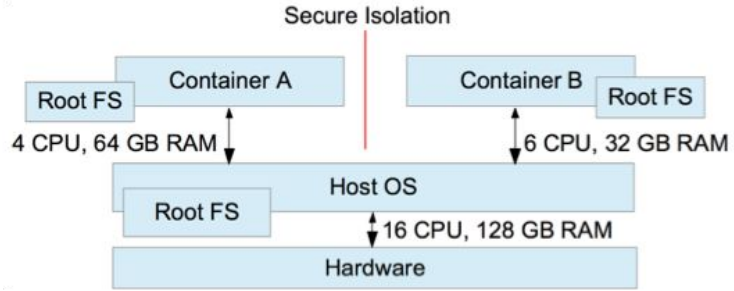
OS-Virtualized Application Stack



remove VM plus this part for Bare-Metal OS-Virtualization

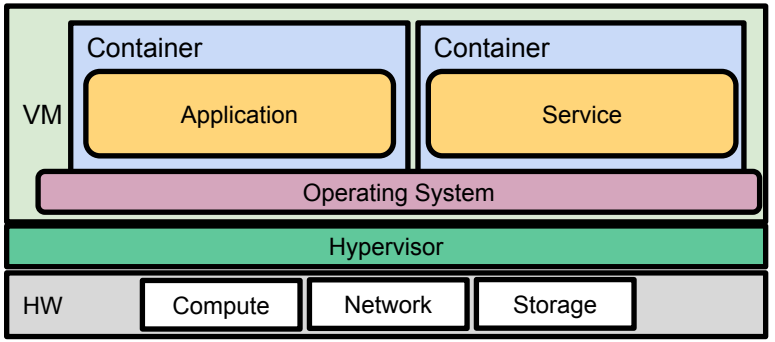
OS Level Virtualization (recap from CCP1)

- In OS-Level virtualization, a process or a set of processes is called a container.
- A Container is run directly on the Host or Guest OS and has the following characteristics:
  - cgroups: Access to resources (CPU/RAM/IO) is:
    - limited, prioritized, accounted
  - namespaces: Visibility of available resources is limited
    - Which users, processes, networks, mountpoints, ... are visible to the process?
  - chroot: Root filesystem is independent of other containers and Host/VM OS
    - Each container has its own "/"
  - LSM (Linux Security Modules): Container are isolated from each other securely
    - unwanted communication is generally prevented.



Containers as units of management

- Building management APIs around containers rather than machines shifts the "primary key" of operations from machine to application. This has many benefits:
  a. it **relieves** application **developers** and ops teams from worrying about specific details of machines and operating systems;
  b. it provides the infrastructure team **flexibility to roll out new hardware** and **upgrade operating** systems with minimal impact on running applications and their developers;
  c. it **ties telemetry** collected by the management system (e.g., metrics such as CPU and memory usage) **to applications** rather than machines, which dramatically improves application monitoring and introspection, especially when scale-up, machine failures, or maintenance cause application instances to move.



Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes, Google Inc.
Borg, Omega, and Kubernetes Lessons learned from three container-management systems over a decade, ACM Queue March 2016, http://queue.acm.org/detail.cfm?ref=rss&id=2898444

OCI – Open Container Initiative

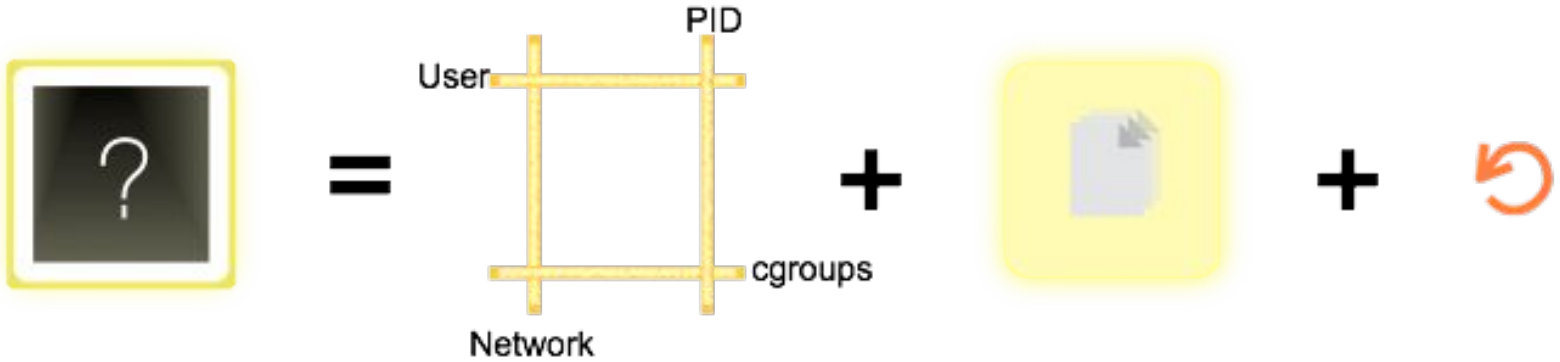
The Open Container Initiative is governance organization specifying open industry standards around container formats and runtimes:

- OCI Image specification
  - File-Format of a filesystem bundle and how to unpack it on disk
    - Image Manifest (metadata, contents & dependencies),
    - Filesystem (layer) serialization,
    - Image configuration (arguments, environments, …)
- OCI Runtime specification
  - How to start and run the image in a OS container
  - Two implementations: runC (Docker), ~~rkt~~ (CoreOS/RedHat)

OPEN CONTAINER INITIATIVE

https://opencontainers.org

Runtime Environment



isolation

contents

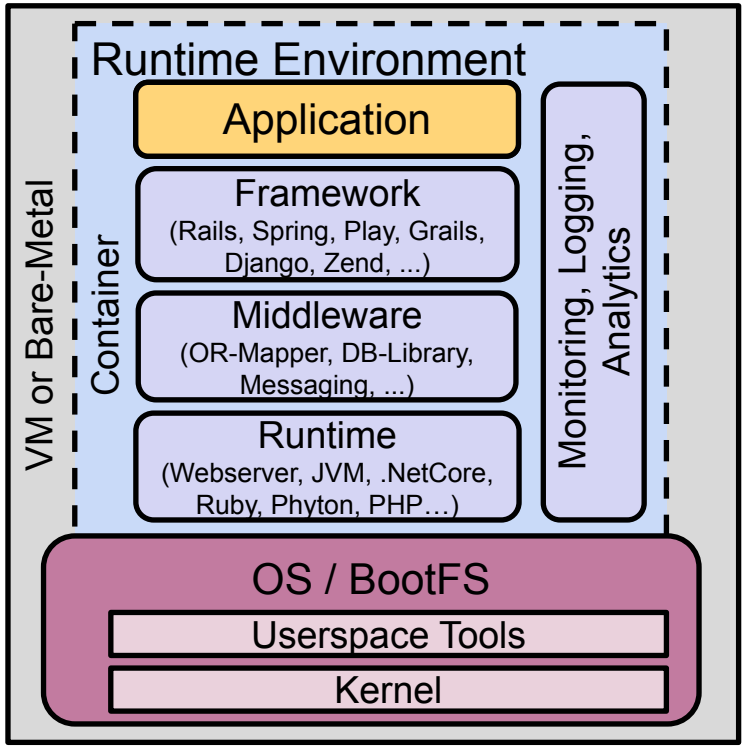
- Base OS / DIST
- Runtime
- Middleware
- Frameworks
- Application

processes

- Start command(s)
- poss. watchdog
- log-collector

Runtime Environment

- The **Runtime** is the component executing the Application
  - Interpreter: e.g. Ruby, Python, Node, PHP
  - Virtual Machine: e.g. JVM, .Net Core
- The **Runtime Environment** contains all required components to run the Application
  - OS / BootFS (Kernel and Userspace Tools)
  - **Runtime**, Monitoring, Analytics
  - Middleware, Frameworks
- Typically built starting from a **Base-Image** (stemcell, stack) containing the more static **OS dependent** components like the BootFS, OS Kernel, Monitoring, Logging and often the Runtime*)
- Dynamically adding the **Application dependent** components like Middleware, Frameworks, Libraries and finally the compiled Application



*) May also be added dynamically, e.g. when version is depending on Application

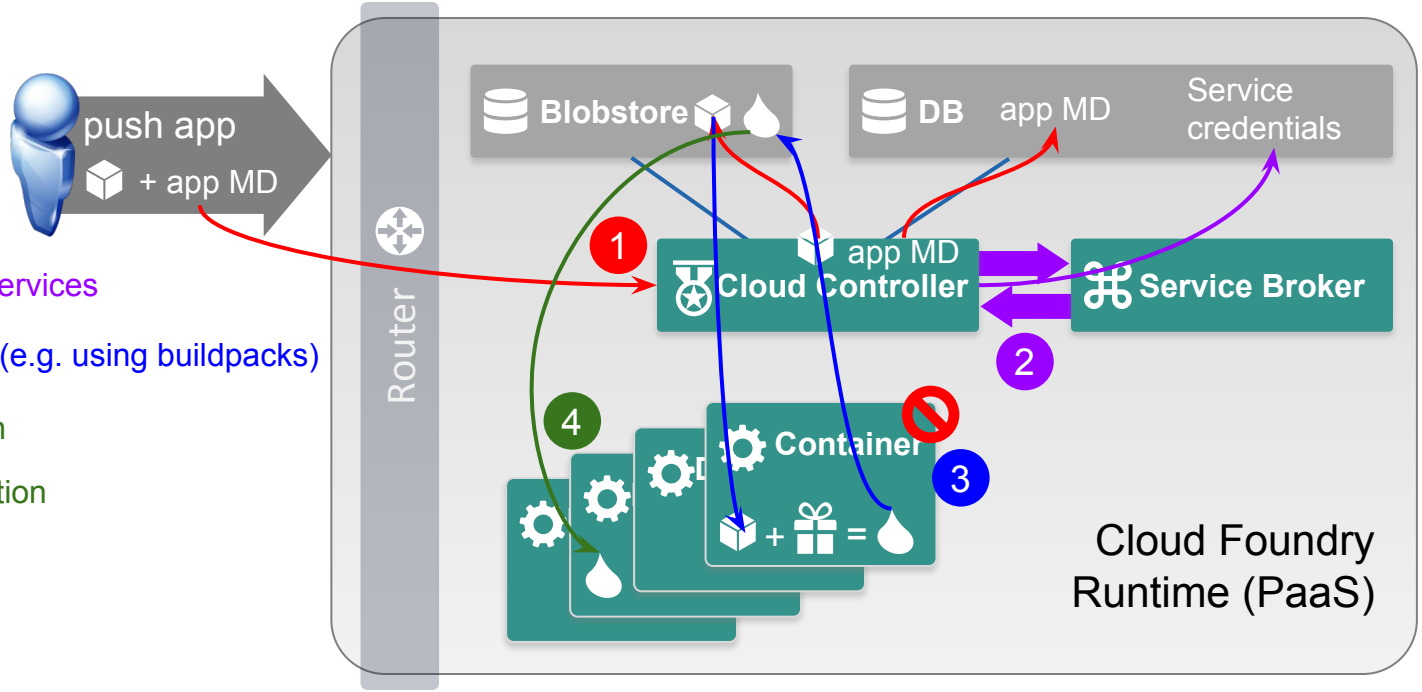
Building Runtime Environment

- Most PaaS systems provide a separate process of packaging application in runtime environments (containers / VMs)
  - This process is called **Staging**
- Recall the **essential steps of the software automation pipeline**
  - Resolving the application's runtime dependencies
  - Automated setup and configuration of runtime, middleware frameworks, monitoring
  - compilation of app (if required)
  - packaging of app

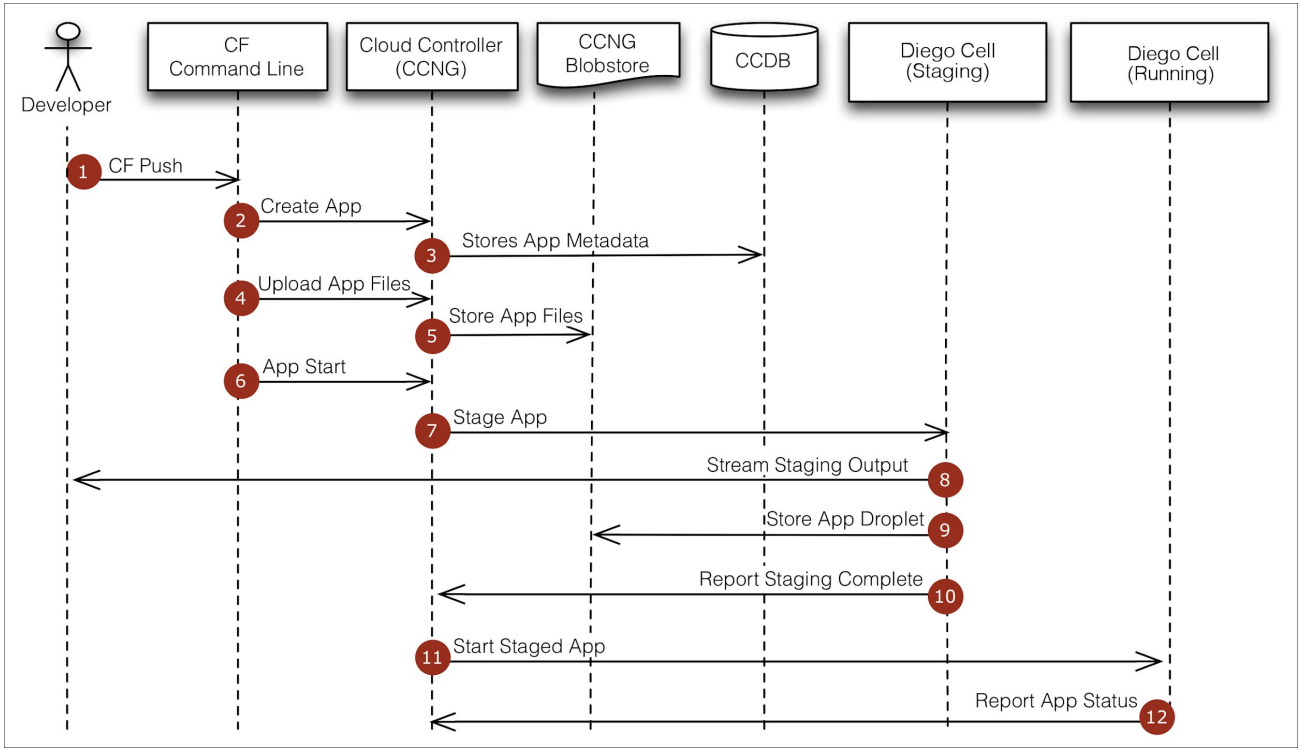
Example: Cloud Foundry Staging

1. Upload app bits and metadata
2. Create and bind services
3. Stage application (e.g. using buildpacks)
4. Deploy application

   Provision application



Example: Cloud Foundry Staging using Buildpacks



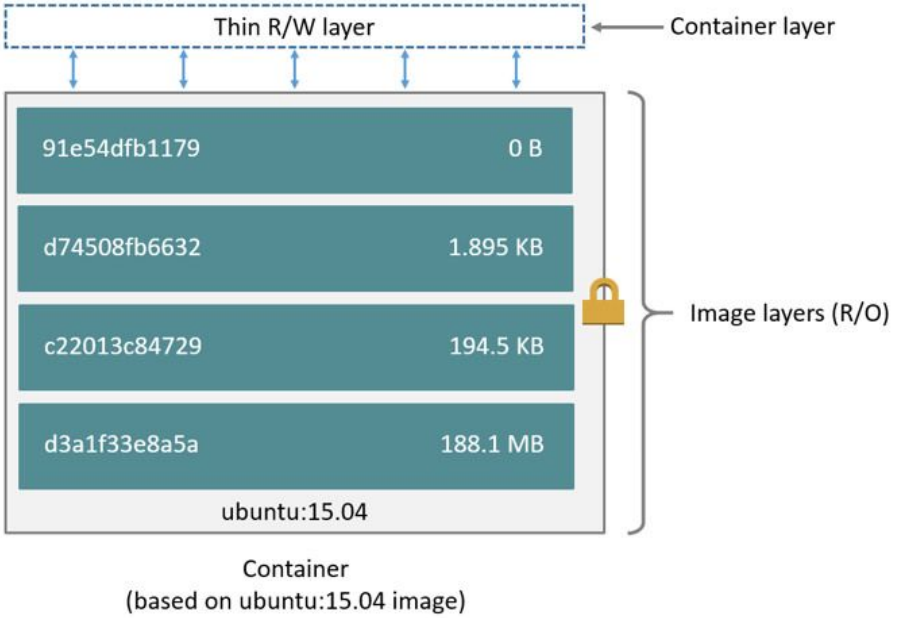
Runtime Environment Builder Tools

In the following we cover common ways to **create Runtime Environments**:

- Dockerfiles
  - use plain Docker build workflow using Dockerfiles
  - e.g. in CI/CD pipeline, or using automation tools like Ansible, Puppet, ...
- Buildpacks (Heroku / Pivotal / CNCF)
  - v1&2: uses simple scripts to build the app environment in a base container (stack)
  - CNCF: process to creates layered OCI images on top of a base image
- Builder Images (Source To Image – s2i)
  - predefined workflow (builder images) to create OCI images including applications available in source code

Staging using Docker (CCP1)

- Docker uses a layered filesystem
  - Each layer is called an image and overlays the previous layer (called parent image).
  - Layers can be traversed until the bottom of the image stack where the final image is called **base image**.
  - Base images contain the Kernel, virtualization tools, the bootfs and user-space tools (e.g. from Ubuntu)



- When a container is launched from an image, Docker mounts a read-write filesystem on top of any layers below.
  - Because each container has its own thin writable container layer and all data is stored in this container layer, multiple containers can share the same image and have their own data state

Create Images using Dockerfile (CCP1)

- The recommended, portable, way to create custom Docker images
- By default is a text called `Dockerfile`
- Follows a very simple syntax:
  - `FROM` an existing image
  - `RUN` a command (e.g. `apt-get install` …)
  - `ADD` a (e.g. config) file from the host machine
  - `EXPOSE` a port by default when the user selects the `-P` option
  - `CMD` to run a command by default when the container is started and the operator does not override it
  - `ENTRYPOINT` the command that will always be executed when the container starts
- An image is built using:
  - `docker build -t <MY_CONTAINER_IMAGE_NAME> .`
  - run from the folder where the `Dockerfile` is.

Simple Example:

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y
 nginx
EXPOSE 80 443
ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

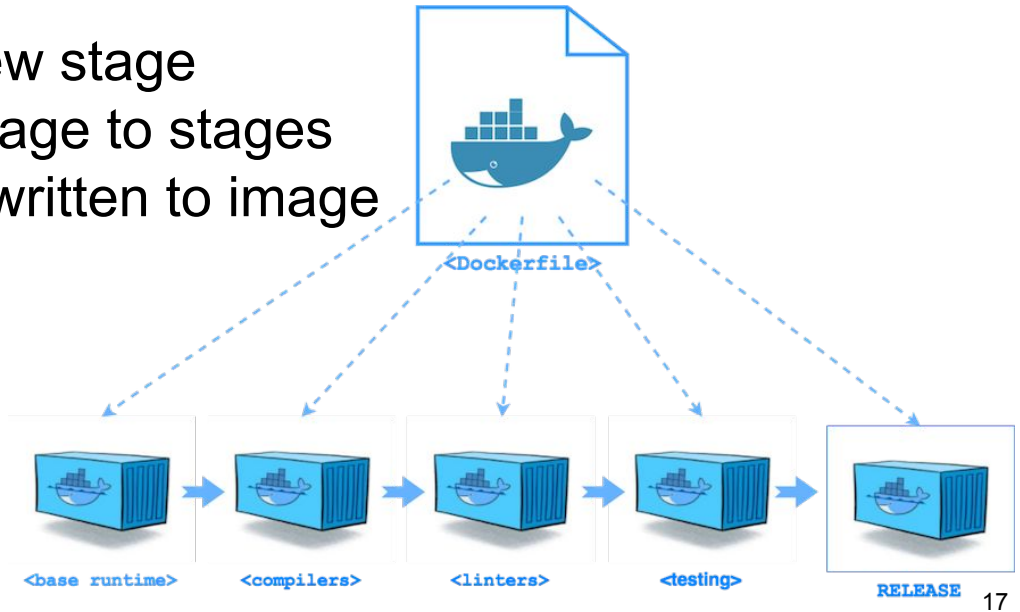
Difference:

- ENTRYPOINT: command that will always be executed when the container starts.
- CMD: arguments that will be given to the ENTRYPOINT.

Note that there are other forms and additional differences for CMD and ENTRYPOINT: https://goinbigdata.com/docker-run-vs-cmd-vs-entrypoint/

Multi-Stage Dockerfiles

- Runtime / Release image size should be small
- Build & Testing tools are large and not required at runtime
- Separate stages in Dockerfile
- Each FROM-block starts a new stage
- Copy content / results from stage to stages
- Only last processed stage is written to image



Multi-Stage Dockerfiles


```
# Stage 0
FROM maven:3.8.5-openjdk-11-slim as build
ENV MAVEN_OPTS="-XX:+TieredCompilation -XX:TieredStopAtLevel=1"
WORKDIR /app
# fetch dependencies
COPY pom.xml .
RUN mvn dependency:go-offline
# build source
COPY ./src ./src
RUN mvn clean install -Dmaven.test.skip=true

# Stage 1
FROM openjdk:11.0.10-jre-slim as run
COPY --from=build /app/target/*.jar /app/spring-boot-application.jar
ENTRYPOINT ["java", "-Xmx300m", "-Xms300m", "-XX:TieredStopAtLevel=1", "-noverify", "-jar",
"/app/spring-boot-application.jar"]
EXPOSE 8080
```

Stop at a specific stage: `docker build --target build -t ccp2/sample-maven:latest .`

Are Dockerfiles good for runtime management?

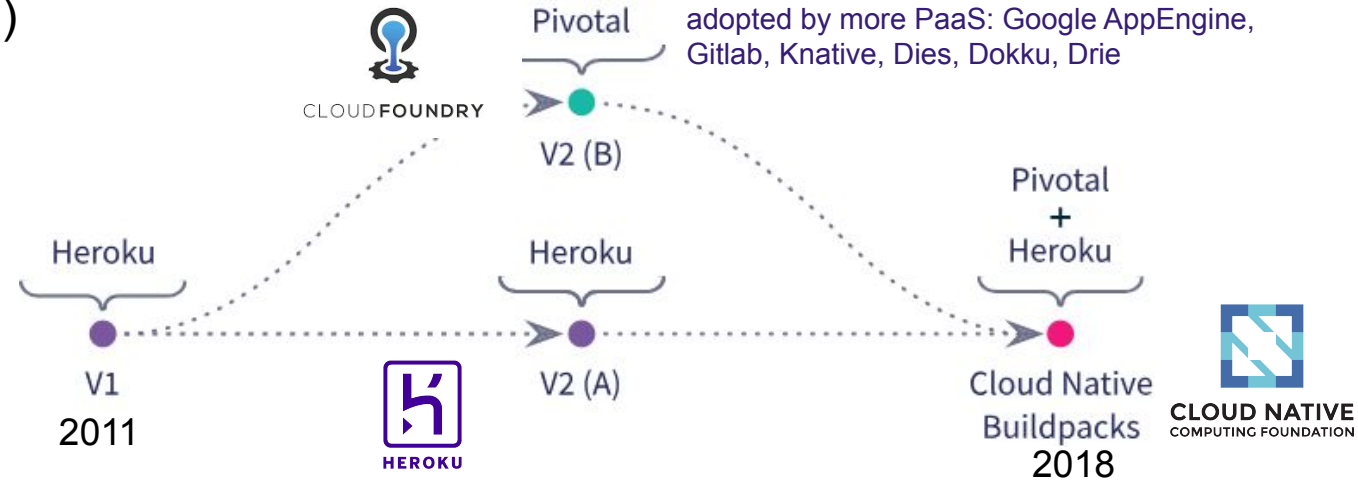
- Advantages using Dockerfiles:
  - Very flexible
  - Very extensible
  - A lot of base images exists
  - Easy to extend and adopt
- Disadvantages
  - Difficult to control image sources
  - Allows also dangerous functions
  - A lot of work to build generic base images
  - Images can get huge

Staging using Buildpacks

Buildpacks.io

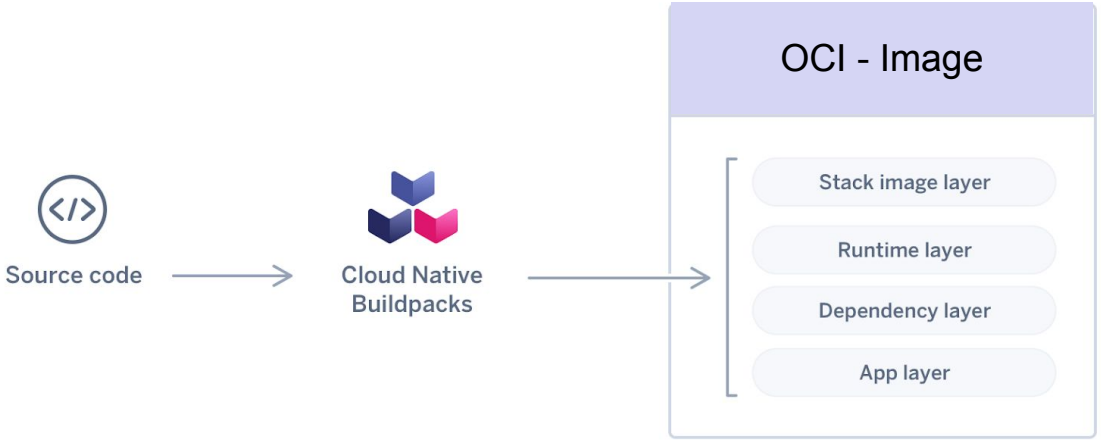
Zurich University of Applied Sciences
School of Engineering
InIT Institute of Applied Information Technology

- Invented by Heroku to create a generic framework to **build runtime environments**.
  - Simplify app deployment - focus on your code
  - lesser files to manage → small images → quicker app deploys
  - Produces self-contained, runnable application artefacts (blobs/archives, images)



What is a Buildpack

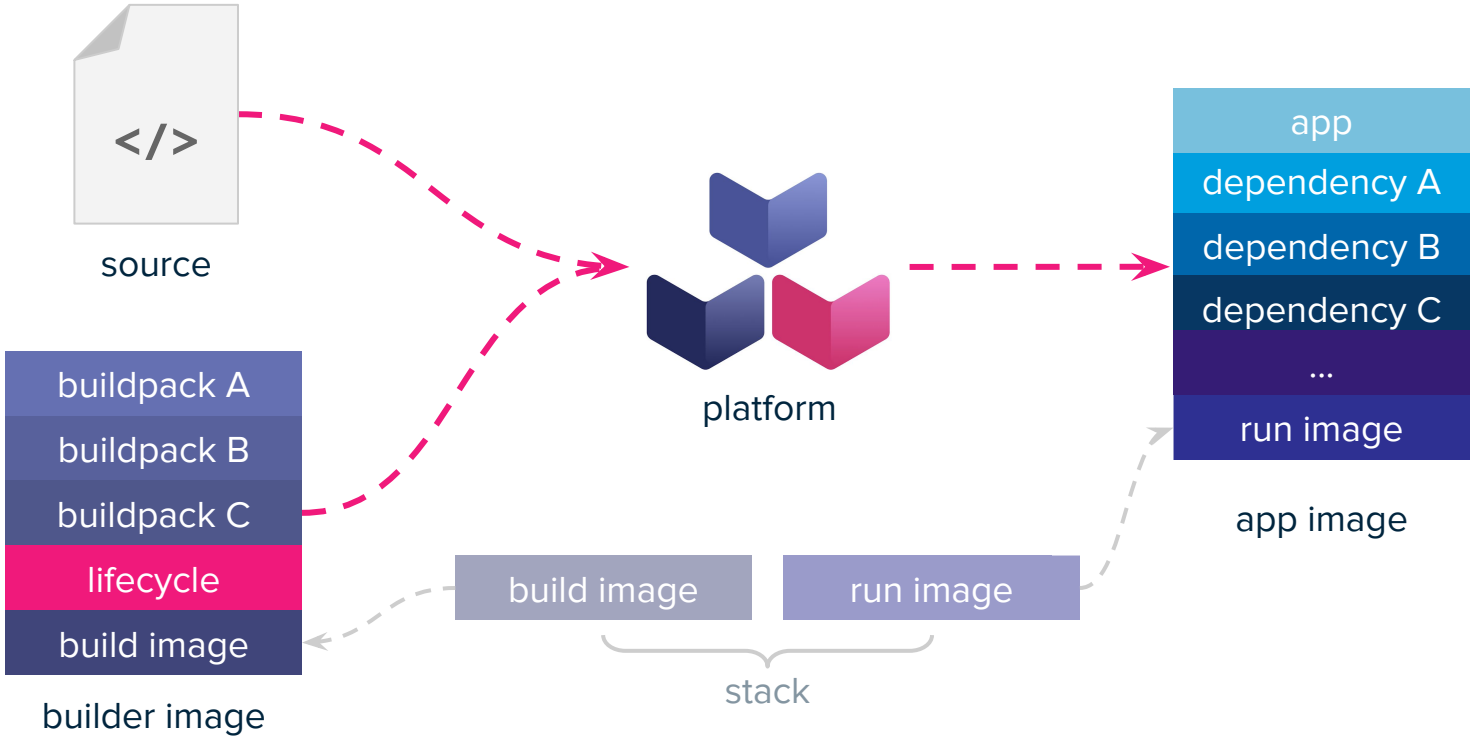
- Process & Toolbox to examine source code, determine dependencies and formulate and execute a plan to build and run your application.



OCI App image that has

- a reproducible build
- metadata that can be inspected
- logical mapping of layers to components

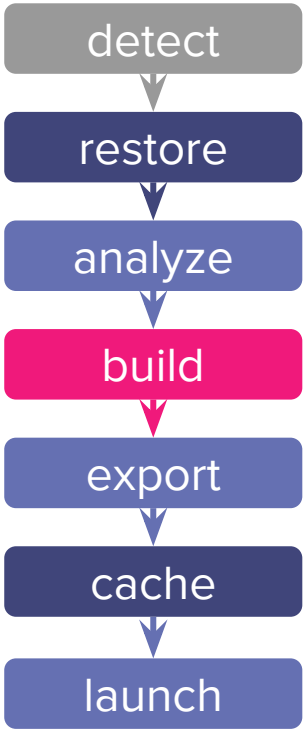
Components


source
platform
app
dependency A
dependency B
dependency C
...
run image
app image
buildpack A
buildpack B
buildpack C
lifecycle
build image
builder image
build image
run image
stack

Components

- ## Builder Image
  Image that contains all the components necessary to execute a build. A builder image is created by taking a build image and adding a lifecycle, buildpacks, and files that configure aspects of the build including the buildpack detection order and the location(s) of the run image.
- ## Buildpack
  A buildpack is a unit of work that inspects your app source code and formulates a plan to build and run your application.
- ## Platform
  A platform uses a lifecycle, buildpacks (packaged in a builder), and application source code to produce an OCI image. E.g. CLI-Tool (pack), CI-CD-Plugin (Tekton), Application Platform - PaaS (kpack)
- ## Stack
  A stack is composed of two images that are intended to work together:
  - The build image of a stack provides the base image from which the build environment is constructed. The build environment is the containerized environment in which the lifecycle
    (and thereby buildpacks) are executed.
  - The run image of a stack provides the base image from which application images are built.

Lifecycle



Finds an ordered group of buildpacks to use during the build phase

Restores layers from the cache

Restores files that buildpacks may use to optimize the build and export phases.

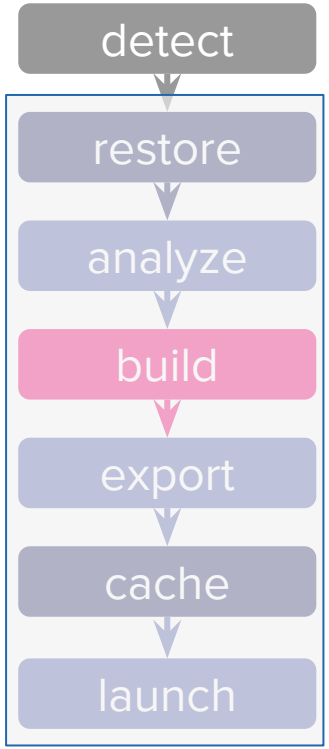
Transforms application source code into runnable artifacts that can be packaged into a container.

Creates the final OCI image.

Caches possible artefacts for reuse

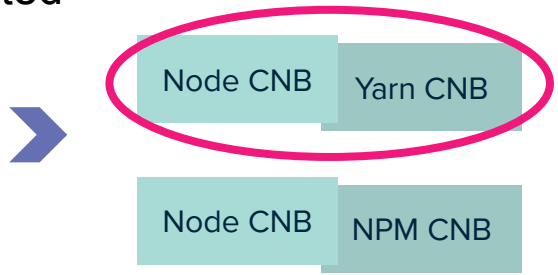
Entrypoint for the OCI image. Responsible for launching the application process.

Lifecycle: Detect

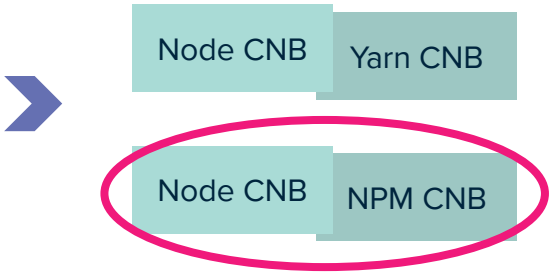


- Tests groups of buildpacks against source, in order (via each buildpack's detect binary)
- First group that passes is selected

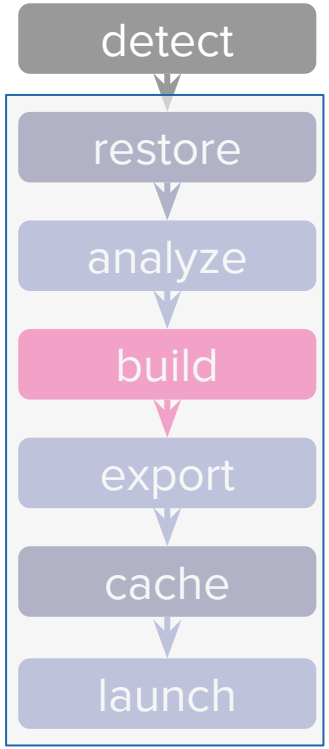
```
src/
  package.json
  yarn.lock
  ...
```



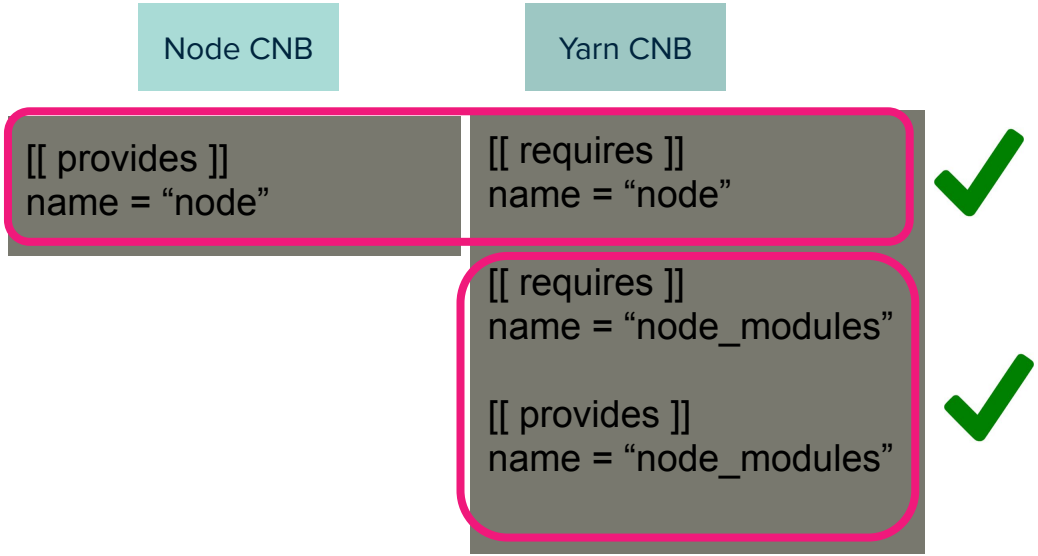

```
src/
  package.json
  ...
```



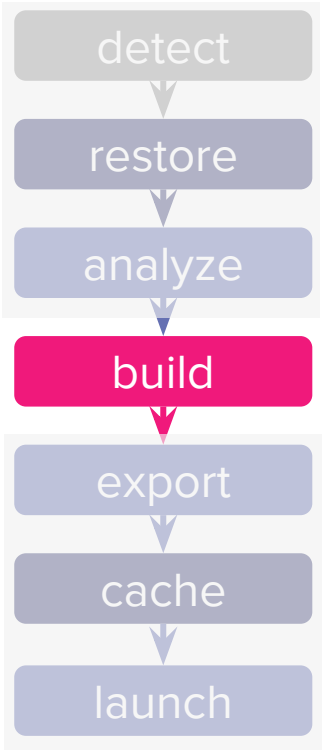
Lifecycle: Detect - Build Plan



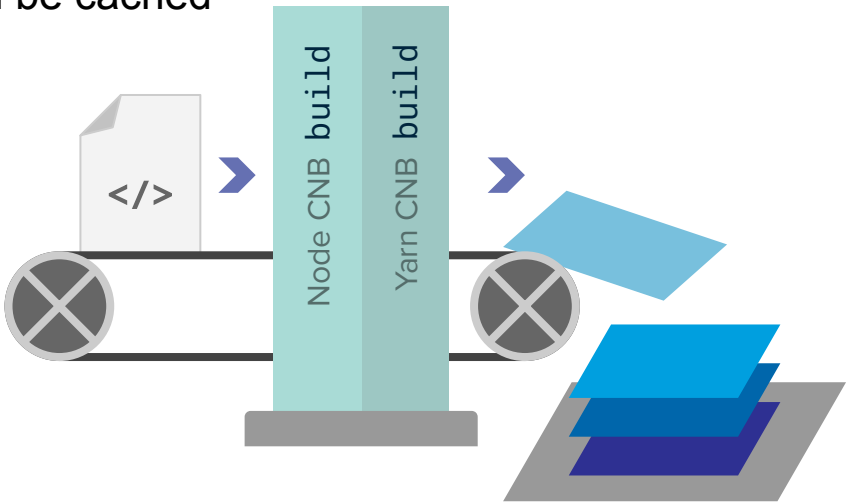
- Buildpacks detect in parallel
- The build plan allow buildpacks to coordinate during detection
- Composability allows for easy customization or extension



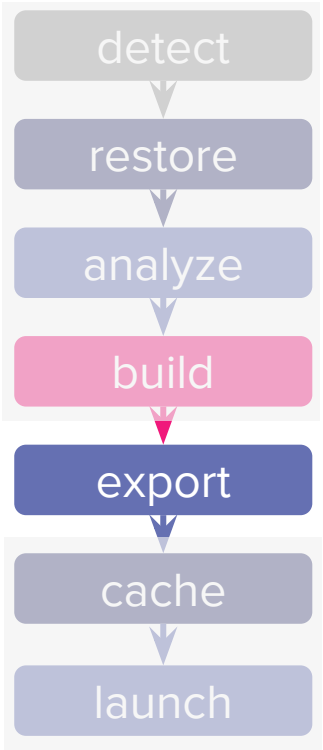
Lifecycle: Build



- For previously-selected group, executes each buildpack's build command in order
- Recall: build gathers dependencies, compiles app (if needed), and sets launch command
- Specifies in a layer.toml file, if this layer is used for build or runtime and if it should be cached

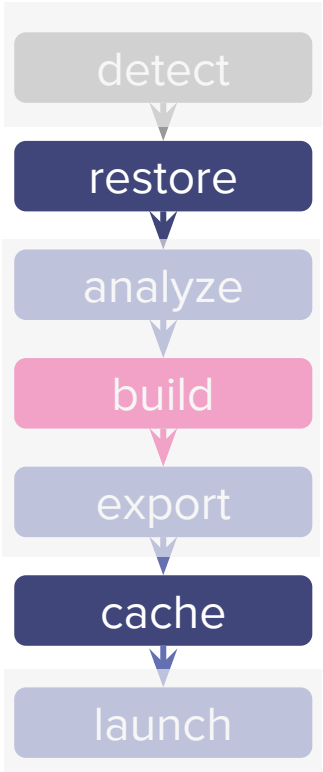


Lifecycle: Export

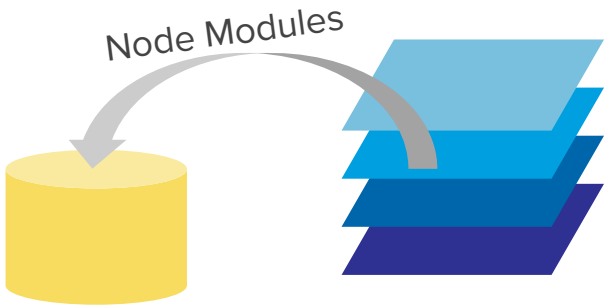


- Assembles final layers into image
- Combines information from analyze phase to ensure only changed layers are update

Lifecycle: Cache & Restore

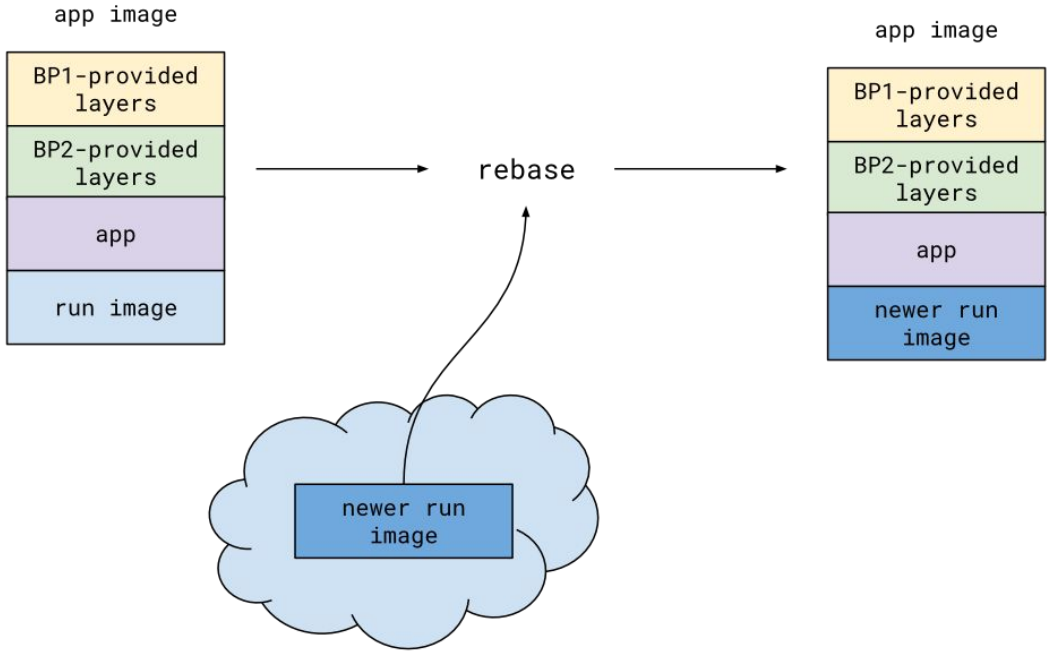


- Caches any necessary dependencies
- Retrieved on next build's restore phase



Rebasing

- Allows developers or operators to rapidly update an app image when its stack's run image has changed.
- Avoids the need to fully rebuild the app.
- Security patches by the Operator without access to source code



buildpacks.io pack trusted builders / buildpacks

- Default buildpacks (depends on runtimes supported by platform provider)

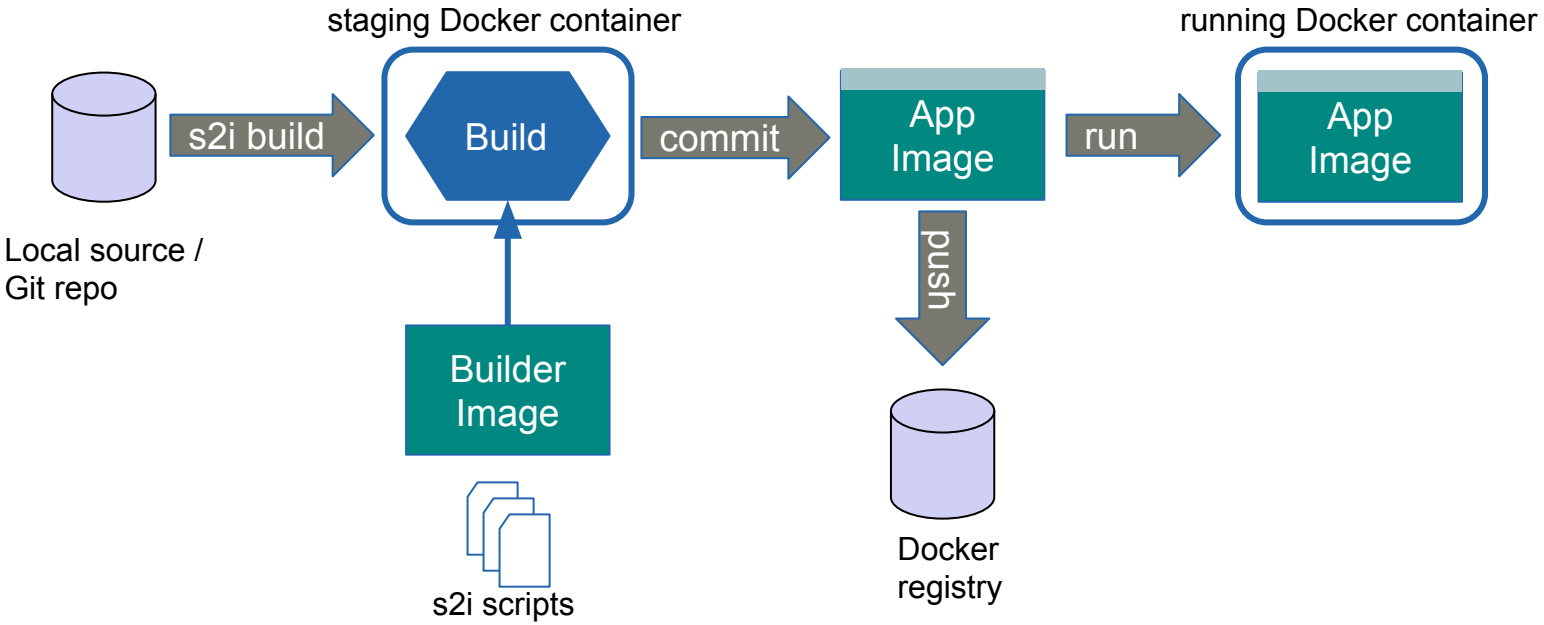
```
$ pack builder suggest
Google:            gcr.io/buildpacks/builder:v1  Ubuntu 18 base image
                   with buildpacks for .NET, Go, Java, Node.js, and Python

Heroku:            heroku/buildpacks:20          Base builder for Heroku-20 stack, based on ubuntu:20.04

Paketo Buildpacks: paketobuildpacks/builder:base     Ubuntu bionic base image
                    with buildpacks for Java, .NET Core, NodeJS, Go, Python, Ruby, NGINX and Procfile
Paketo Buildpacks: paketobuildpacks/builder:full     Ubuntu bionic base image
                    with buildpacks for Java, .NET Core, NodeJS, Go, Python, PHP, Ruby, Apache HTTPD,
                    NGINX and Procfile

Paketo Buildpacks: paketobuildpacks/builder:tiny   Tiny base image (bionic, distroless-like run image)
                    with buildpacks for Java, Java Native Image and Go
```

Annex

Source-to-image (s2i)

- Source-to-image (s2i) is a tool for building reproducible, executable Docker images (Runtime Environments)
  - From “source code to complete and runnable image”
  - Output is ready to use with **`docker run`**
- Philosophy
  - Simplify process of creating usable image of applications for most use cases
    - application source + builder image results in runnable application image
  - Support incremental builds
  - Provide support for verification
  - Use native Docker primitives

s2i - basic workflow



s2i - basic workflow

- Invocation
  - `s2i build <directory/git repo> <builder-image> <output-container-image>`
- What does this do?
  - creates a container based on the Builder Image and injects tar file containing:
    - The **application source** in `/tmp/src (within the container)`, excluding any files specified in `.s2iignore`
    - If existing (see incremental build) from previous runs, **build artifacts** to `/tmp/artifacts` (within the container)
  - sets the environment variables from `.s2i/environment` (optional)
  - starts the container and runs its **`assemble`** script (part of the Builder Image, script that runs the build process)
  - commits the container
    - sets the CMD for the output image to be the **`run`** script
    - tags the image with the name provided

s2i - anatomy of a builder image

- s2i supports the following scripts provided as part of a Builder Image (placed in `/usr/local/bin/s2i/... see next slide`):
  - **`assemble`** (required) - builds and/or deploys the source to appropriate directory within container, download dependencies, inject configuration
    - can be complex, depending on application build process
  - **`run`** (required) - start script to run the assembled artifacts
    - used when resulting image is started with docker run
  - **`save-artifacts`** (optional) - extracts the artifacts / dependencies from a previous build for use in the next incremental build
  - **`usage`** (optional) - displays builder image usage information
- Typical use case is to build parent layers (runtime) into build image with dockerfiles and final layer (application) on top with s2i

s2i - example Dockerfile to create a builder image

Zurich University of Applied Sciences
School of Engineering
InIT Institute of Applied Information Technology

```
# reveal-js-builder-image
FROM ubuntu
```

Use `ubuntu` source image

```
ENV TERM linux
RUN apt-get update
RUN apt-get install -qqy dialog apt-utils
RUN apt-get install -yqq git nodejs npm #runtime and buildtool
RUN ln -s /usr/bin/nodejs /usr/bin/node #setup links to executable

COPY ./s2i/bin/ /usr/local/bin/s2i #copy s2i-builder scripts

RUN useradd -ms /bin/bash processuser

USER processuser

EXPOSE 8000

CMD ["usage"]
```

`apt-get` dependencies required to build and run application

Copy `assemble`, `run`, `usage`, `store-artifacts` to builder img

Don't run as `root`

Default usage is to tell user how to use this builder image

s2i - example assemble script


```
#!/bin/bash -e
# NodeJS-based application

# If the 'reveal-js-builder-image' assemble script is
# executed with the '-h' flag, print the usage.
if [[ "$1" == "-h" ]]; then
      exec /usr/libexec/s2i/usage
fi

echo "---> Installing application source..."
PRES_DIR=$HOME/presentation
mkdir $PRES_DIR
cp -Rf /tmp/src/. $PRES_DIR #sources incluing html/java-script

echo "---> Building application from source..."
cd $PRES_DIR
npm install grunt-cli #node package, NodeJS build tool
echo "-----> Installing node.js dependencies - this may take a while..."
npm install #resolves all NodeJS specific dependencies
```

Help instructions

Copy files from `/tmp/src` into working directory

Set up `grunt`, installing all dependencies

s2i - example run script


```
#!/bin/bash -e
#
# S2I run script for the 'reveal-js-builder-image' image.
# The run script executes the server that runs your application.
#

PRES_DIR=/home/processuser/presentation
cd $PRES_DIR
export PATH=$PATH:$PRES_DIR/node_modules/grunt-cli/bin
exec grunt serve --hostname='0.0.0.0' --port=8000 #starts server with HTML/JS
files
```

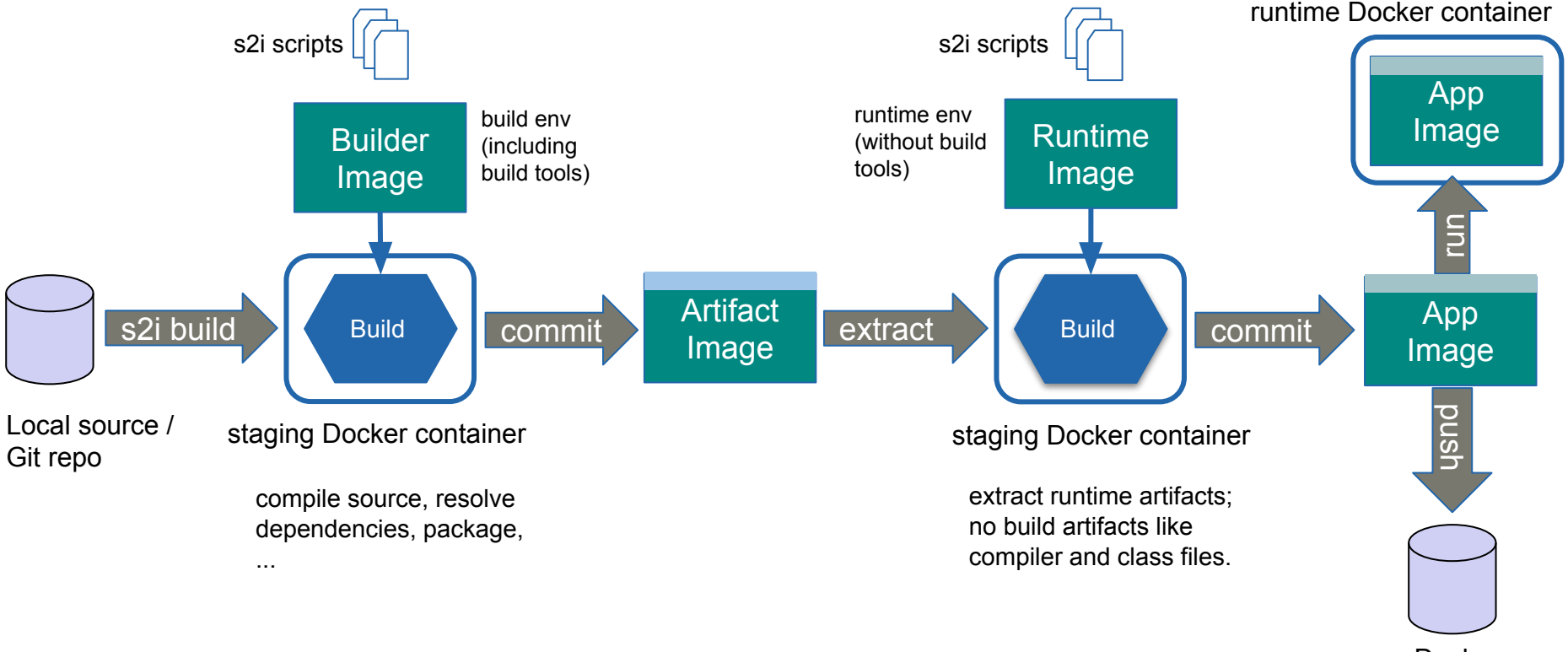
Enter presentation directory, setup `PATH` and run `grunt`

s2i - managing artifacts

- s2i supports incremental builds
  - artifacts from previous build used in current build (e.g. compiled output)
  - dependencies not known a priori (maven, npm, … app artifacts) cannot be included in builder image
  - assemble script requires logic to detect dependencies if present and obtain them if not
- s2i enables artifacts from previous build to be used in current build
  - detects if previous build exists (container with same name)
  - makes copy of container
  - runs save-artifacts from previous build before setting up environment for new build
- Also supports more advanced build process - different build and run containers

s2i - extended builds (builds considering artifacts from previous builds)

Zurich University of Applied Sciences
zh aw School of Engineering
InIT Institute of Applied Information Technology

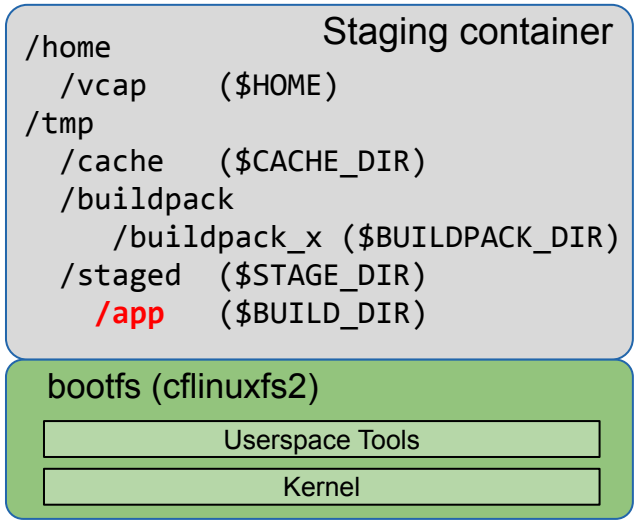


Anatomy of a buildpack (API) V2

- A buildpack essentially consists of three executable files:
  - **`bin/detect`: determines whether to apply buildpack to an app (Analysis)**
    - detect typical patterns of the application for the supported runtimes
    - e.g look for package.json for node.js, Gemfile for ruby, setup.py for python
  - **`bin/compile`: perform the transformation steps on the app (Droplet)**
    - process/compile the source, resolve dependencies
    - download runtime environments (JRE, ruby, go, phyton…) and middleware
    - configure runtime and middleware (create/adapt config files)
    - copy executables, dependencies, config and resources to the correct folders
  - **`bin/release`: provide metadata back to the runtime (Metadata)**
    - return a yaml file (to stdout) containing the command to run the application

Buildpack Staging Workflow - (1) prepare environment

- Developer: pushes application (`cf push`)
  - application bits (diffs) are uploaded to platform ➡ blobstore
- Platform: create **Staging Container** and copy application content
  - Uses a generic Linux kernel with a rootfs (default: `cflinuxfs3` based on Ubuntu 18.04)
  - $HOME is the user home (/home/vcap)
    - where the application will be deployed at runtime (see Deploying the Droplet)
  - $CACHE_DIR used to cache artifacts between multiple buildpack runs
  - $BUILD_DIR initially contains uploaded app bits at the end of the process it should contain the entire self-contained application (including all artifacts)



Buildpack Staging Workflow - (2) Analysis

- if no explicit buildpack is specified → start buildpack detection
  - go through the **ordered list** of default buildpacks
  - download each buildpack to `/tmp/buildpack/<name>` and call its detect script: **`$ bin/detect $BUILD_DIR`**
    - detect if the buildpack is able to process the app
    - on success
      - print buildpack name (version,...) to stdout
      - return exit value 0 (success), otherwise exit with return value ≠ 0
  - the **first** buildpack returning 0 (success) is selected
- if buildpack is specified
  - download the buildpack to /tmp/buildpack/... (`bin/detect` is not called)

Buildpack Staging Workflow - (3) Droplet

- run the compile script of the selected buildpack
  - **`$ bin/compile $BUILD_DIR $CACHE_DIR`**
  - process the app in $BUILD_DIR
    - process/compile the source, resolve dependencies
    - download runtime executables (JRE, ruby, go,…) and middleware
    - configure runtime and middleware (create/adapt config files)
    - copy executables, dependencies, config and resources to the correct folders
    - The content of `$CACHE_DIR` will made available whenever staging is run again
  - at the end `$BUILD_DIR` should contain the configured runtime, runnable app code and all dependencies
  - on success return exit code 0 (success), otherwise ≠ 0

Buildpack Staging Workflow – (4) Metadata

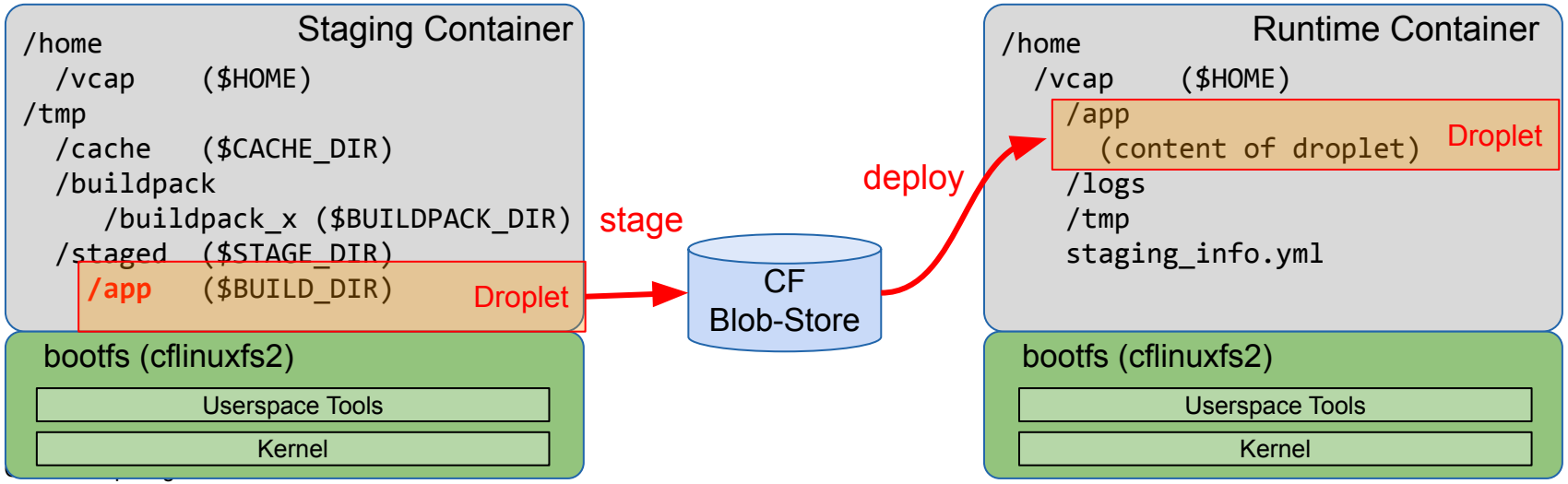
- run the release script of the selected buildpack
- **`$ bin/release $BUILD_DIR`**
  - output a YAML string containing metadata to run the application

```
addons: []                    // list of heroku addons
config_vars: {}               // config vars to set on heroku
default_process_types:
  web: <start command>        // e.g. bundle exec rackup config.ru -p $PORT
```

  - CloudFoundry only supports process type `web`, ignores `addons` and `config_vars`
  - `<start command>` is only used if no run command is declared in CLI or manifest
  - The command is executed using: `bash -c <start command>`
- store archive in Blob-Store
  - create an archive file of `$BUILD_DIR` and upload it to the CF Blob-Store

Buildpack - Deploying the Droplet

- When an Application instance is deployed a **Runtime Container** is created
  - runtime container is (as) similar (as possible) to the staging container
  - Droplet from Blob-Store is copied and unpacked to $HOME/app
  - staging_info.yml contains results of staging process (e.g. start command)



Buildpack - Example of a Custom Buildpack


```
heroku-buildpack-markdown/
├── README.md
├── bin
│   ├── compile
│   ├── detect
│   └── release
└── opt
    └── Markdown.pl
```

bin/detect

```
#!/bin/sh
# this pack is valid for apps with a index.md
# in the root
if [ -f $1/index.md ]; then
  echo "Markdown"
  exit 0
else
  exit 1
fi
```

bin/release

```
#!/bin/sh
cat << EOF
---
default_process_types:
  web: python -m SimpleHTTPServer \$PORTEOF
```

```
/home
   /vcap (aka $HOME)
/tmp
   /cache (aka $CACHE_DIR)
/tmp
   /buildpacks/
      /heroku-buildpack-markdown(aka $BUILDPACK_DIR)
/tmp
   /staged (aka $STAGE_DIR)
      /app (aka $BUILD_DIR)
```

bin/compile

```
#!/usr/bin/env bash
set -eo pipefail
BIN_DIR=$(cd $(dirname $0); pwd)
ROOT_DIR=$(dirname $BIN_DIR)
PERL=`which perl`
echo "-----> Found an index.md"
echo "-----> Markdown to HTML"
MARKDOWN_FILES=`find $1 -name "*.md"`
for f in $MARKDOWN_FILES; do
  NEWFILE=${f%.*}.html
  echo "       $f -> $NEWFILE"
  echo -e "<html>\n<body>" > $NEWFILE
  $PERL $ROOT_DIR/opt/Markdown.pl $f >> $NEWFILE
  echo -e "</body>\n</html>" >> $NEWFILE
  rm $f
done
```