KubeCon | CloudNativeCon

Europe 2018

Going from Source to Image

Introduction to Key Concepts

Ben Parees(Red Hat)

Matt Moore (Google)

Steve Speicher (Red Hat)

Agenda

- What do we mean by source to image?
- Why would you want it?
- Available approaches
  - **Dockerfile-based:** Docker, Kaniko, img, buildah
  - **Source-based:** s2i, buildpacks, “FTL”
- Discussion

What is it?

- Generally, a mechanism for going from application source to a runnable application image
- Ideally, by abstracting/hiding the details of image construction from an application developer
  - Don’t give them permissions+complexity they don’t need
  - Give them tools that consume the artifacts they already know how to work with (source code, not dockerfiles)

Why do I want it?

- Kubernetes runs images
- You need to build images
- Why not build images on Kubernetes?
  - One less piece of infrastructure to maintain
  - Need a secure mechanism to do so
- Attend our Deep Dive session to discuss how these technologies can be brought to Kubernetes

Approaches - Docker Daemon

- Have an accessible docker daemon
- Run “docker build”
- Run “docker push”
- Pros
  - It’s what every developer knows how to do today
- Cons
  - Handing out docker socket privileges on shared systems isn’t viable

Approaches - Daemon-less Dockerfile Builds

- Consume a Dockerfile, but build image **without** a docker daemon
  - Run a container directly and invoke Dockerfile steps directly, committing result
  - … or just construct image layers directly
- Pros
  - Docker build-like experience (just write a Dockerfile)
  - Potentially more control over image layers (combine or shard)
  - Aim is for greater security
- Cons
  - Dockerfile fidelity
    - May lag in supporting the latest Dockerfile syntax extensions
  - Different approaches to image layer construction
    - How many layers are in the final image, what is in a given layer

Approaches - Daemon-less Dockerfile Builds

- **buildah**
  - Runs a container directly (no container runtime daemon)
  - Simulates Dockerfile evaluation within the container
  - Commits the final container state (no layers)
- **img**
  - *"You could even probably use buildah as unprivileged if you use the same instructions from the unprivileged mounting section below."* -- img README.md
- **kaniko**
  - Simulates Dockerfile evaluation within an empty container
    - Designed to be run within a container orchestrator (e.g. on-cluster)
  - 100% user-mode: no syscalls to start/stop containers or snapshot filesystem.
  - Directly publishes resulting image to Registry

Approaches - Dockerfile-less builds

- User input is source / intent: “I want to run a Node.js web server”
  - Builder makes it so.
  - Sometimes incorporates detection.
- Pros:
  - Less configuration
  - Tools can intelligently build layers, better/safe layer caching.
  - More serviceable by vendors
  - Docker image best practices can be codified into tools
- Cons:
  - Less flexible
  - Very fragmented across vendors, no real standard.

Approaches - Dockerfile-less builds

- Source to Image
  - User provides source, source gets built+layered into an application image
  - Dependent on ecosystem of framework/language builder images
- Buildpacks
  - Popularized by Heroku, moving towards containers.
  - User provides source, “build” produces “droplet”, “export” produces container image
- FTL
  - Purpose-built source to image builders per-language, goal is layer-per-dependency
  - Insight: turn build incrementality into deploy incrementality.
- Bazel
  - Google’s OSS build system, supports declarative image builds
  - Used for user-mode Docker image builds for 3+ years

Observations

- Clear trend toward securing the image build process
  - Often in support of on-cluster builds
- Clear interest in better image layering strategies
  - Better control over how to group layers, not arbitrary delineation in the Dockerfile
- Clear gap for building images directly from source (in a standard way)

Discuss!

References

- Buildah - https://github.com/projectatomic/buildah
- Img - https://github.com/genuinetools/img
- Kaniko - https://github.com/GoogleContainerTools/kaniko
- S2I - https://github.com/openshift/source-to-image
- Buildpacks - https://github.com/sclevine/packs
- FTL - https://github.com/GoogleCloudPlatform/runtimes-common/blob/master/ftl
- JIB - https://github.com/GoogleContainerTools/jib
- Bazel - https://github.com/bazelbuild/rules_docker#language-rules

Backup

Approaches - buildpacks

- Invented by Heroku, also adopted by Cloud Foundry / Deis
- Lazy binding of application logic to language runtimes
  - Former: “Slugs” (Heroku) or “Droplets” (Pivotal)
  - Latter: “Stacks”
- Build
  - Runs within container to produce “slug”
- Export
  - Enables “rebasing” by rerunning the export.
  - Binds the “slug” to a “stack” and publishes to registry.

Approaches - Google “FTL” builders

- Google Container Builder images for Appengine / Functions.
- Turns source into images following idiomatic conventions
  - Python: `pip install`, Node.js: `npm install`, …
  - VERY purpose-built and restricted in capabilities.
- Insight: turn build incrementality into deploy incrementality.
- No Docker
  - Assembles image layers directly against Docker Registry API
  - Enables caching and a variety of neat optimizations.

Approaches - Bazel + rules_docker

- Bazel
    - OSS form of Google's internal build system
    - Very explicit dependency representation
- rules_docker
    - Declarative container image construction
    - Inspiration and reference implementation for aspects of "FTL"
- No Docker
    - Unprivileged, reproducible, and verifiable docker builds for 3+ years at Google.
    - Supports cross-construction of Docker images (e.g. OSX => Linux)
    - … no support for "`RUN`" => hard for most to adopt.

Approaches - S2I

- CLI tool for building images
- User provides source, someone provides the s2i builder image
  - S2I builder image knows how to convert source to a runnable state
  - S2I commits the new container after conversion and pushes the image
- First class support within OpenShift
- Can be made to run on Kubernetes
- Wraps access to the docker daemon
  - Users have no direct access to the daemon
  - Can control what user id performs the image assembly (no root)

Dimensional Analysis

- Privileged vs Root vs Unprivileged vs VM-based isolation
  - Also user privileged vs wrapped privileged
- Performance implications
  - Is the host layer cache shared between builds? Across nodes?
  - Are layers reusable between builds for assembly + push
    - E.g. Layer squashing effectively destroys layer reuse
- User experience/API/Input
  - Dockerfile
  - Application Source
    - E.g. s2i, buildpacks, but someone must construct those builders/buildpacks