Container Images From Zero

Building Them Bit by Bit

Joe Thompson, Clarity Business Solutions

Intro; Let's Talk

Who's that at the podium?

Almost 30 years in IT (Kubernetes-enabled since 2015)

Past employers: HashiCorp, Mesosphere, CoreOS, Red Hat, among others; currently a Cloud Native Architect for Clarity Business Solutions

Pronouns: he/him
Blood type: Caffeine-positive
Pop-culture references center of gravity: around 1989

How to get in touch:

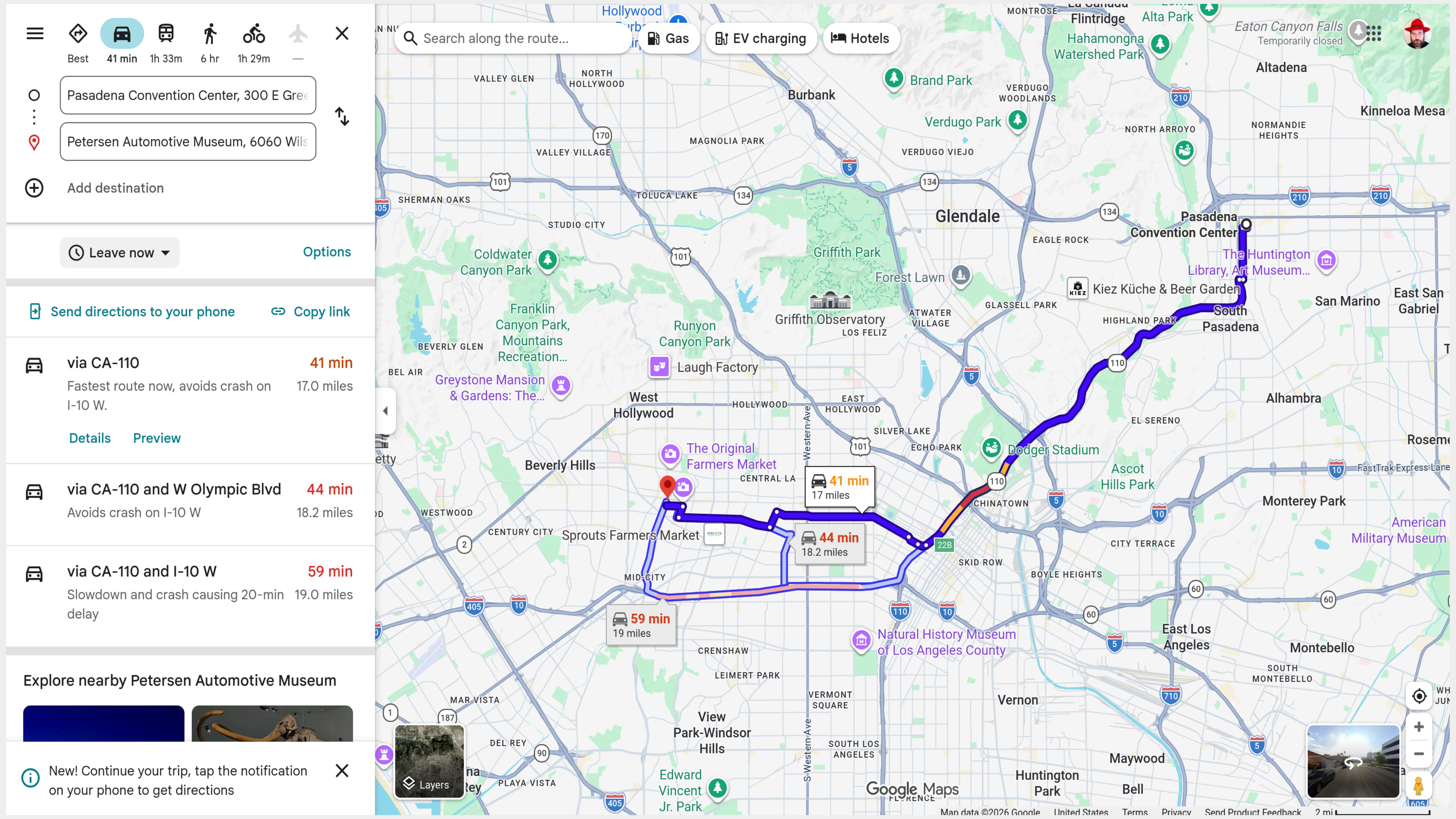
- Kubernetes Slack: @kensey
- LinkedIn

Now about the most important people here -- you!

- Who's here at their first SCaLE?
- How many container image maintainers in the room?
  - How many of you are enjoying it?
- Who's in for heisting the 1989 Batmobile from the Petersen Auto Museum?

It's totally doable!

- Seriously, it's like a 45-minute drive



Best 41 min 1h 33m 6 hr 1h 29m —

Pasadena Convention Center, 300 E Gre

Petersen Automotive Museum, 6060 Wil

Add destination

Leave now Options

Send directions to your phone Copy link

via CA-110 41 min
Fastest route now, avoids crash on I-10 W. 17.0 miles
Details Preview

via CA-110 and W Olympic Blvd 44 min
Avoids crash on I-10 W 18.2 miles

via CA-110 and I-10 W 59 min
Slowdown and crash causing 20-min delay 19.0 miles

Explore nearby Petersen Automotive Museum

New! Continue your trip, tap the notification on your phone to get directions

  - And our getaway vehicle is **the Batmobile!**

Container prehistory

In the beginning was the chroot

- "Change root" -- as in the filesystem root
- Somewhat more restrictive than just setting file permissions -- not only can't the process read the file, it can't even see it
- **Only** restricts the view of the filesystem, though -- not access to the network or process info
    - Although containers are much more common now, chroots were used for a long time to protect host filesystems from processes that might otherwise be tricked into overwriting or exposing critical file contents

Go directly to (FreeBSD) jail

- BSD improved on the concept of a chroot to further isolate processes from each other and the host -- "BSD jails"
- Uses copies of (part of) the host OS and can host multiple jails
  - "Operating system virtualization"
  - Extremely common in shared-hosting environments like FTP (and later web) hosting
- This is isolation strong enough that although people weren't calling them "containers" yet, it was recognizably heading that direction

You have entered... the Solaris Zone

- Sun took jails one step further and created something even stronger called Solaris Zones
- Probably the most developed expression of this was Joyent's Triton, which initially competed with Kubernetes
- Then Kubernetes won the orchestration wars, so that's about as far as that got
  - ...maybe?

Meanwhile, at the Halls of Linux...

- Chroots evolved into various forms of "operating-system virtualization" as a path toward (and eventually a lightweight alternative to) hardware virtualization
  - Hardware virtualization as we know it today really only became feasible in 2006-ish as CPU makers added virtualization assistance to their instruction sets
  - How well do I recall the pain of realizing my Intel CPU was not blessed with the VT instructions...
- Eventually LXC appeared and started to provide a platform on which other projects could build
- Docker ultimately won the container wars (...more or less) and then the Open Container Initiative wrote the spec for the victory

Labs 0 & 1: Image Prehistory; Container images == tarballs

Deep thoughts on the nature and purpose of things

A step back: What *is* a container?

- This is actually a really important question because it influences how you will want to build them
- If a container is a way of running *many complete systems* on a host, you're in an "operating-system virtualization" mindset and may prefer "thicker" containers
- If it's a way to run *many shared services* on a host, you're likely to prefer someting much slimmer
- People get very opinionated on this
  - For what it's worth I tend to agree with most in the cloud-native community that containers are a poor way of providing a general-purpose environment
  - That said... I've seen it done for reasons that were good at the time

Lab 2: Container images as minimal distros

Image building with dedicated tools

Tools are a convenience

- Part of what's nice about the OCI image spec is that it makes building things tool-independent
- It also of course makes a wide array of tools possible

Build config and artifacts

- Most tools use (or can use) some version of a "Dockerfile" to build a container image
- Builds can start from scratch or from an existing "base" image (useful for building images that have only a few differences from each other)
- Conceptually the same as Linux distribution package build file -- "here are some artifacts; do these things with them and bundle it all up neatly"
- Some tools (e.g. HashiCorp Packer) that predate Docker but were extended to build OCI images still use their own manifest format and terminology, but the basic idea is still the same

Image structure

- Images contain one or more layers (basic concept: "content-addressable storage")
- Multi-layered images are an optional construct and build tools can default to creating them or not
- "Images" now may also be an array of pointers that identify actual arch-specific images (which can still have one or more layers)
- Image layering is one of the factors that enable some base image build strategies to work at all

Image distribution

- Images are typically hosted in some form of archive called a registry
- This is not mandatory -- an image can be hosted in multiple registries or none at all
- The destination registry (or registries), if any, will typically not be specified in the build info, but as a tag applied after build
- After build, the build tool (or another tool) can push the image to the registries that will host it

Lab 3: Using OCI image building tools

Image building strategies compared

What base image? The pure upstream approach

- If your image customizations are always built on an upstream source and are minimal enough, you may not need to build images at all
- Override the entrypoint and mount volumes to do the transformations you need
- **Pro:** You benefit immediately from improvements upstream without needing to build anything
- **Con:** If changes are not very simple this can quickly get difficult to maintain or operate

Bare-bones: every image is a base image

- Built from scratch, minimal files included
- **Pro:** Very small per-image, more secure
- **Con:** may cause more storage consumption in the long run
  - If layers are not used, then every rebuild takes up the entire size of the image in storage
  - If layers are used but packaging isn't, updating the build file can become tedious
- **Con:** often harder to debug
- **Con:** security scanners often do not work well on these images because of lack of OS package metadata

Multiple independent base images

- Base images are built per-project or per-team for their specific tools
- **Pro:** Gives you some degree of efficiency and convenience, as well as taking advantage of storage layering, while isolating the effect of base image changes
- **Con:** Base images will tend to proliferate; governance and maintenance can quickly become a burden

Image tree

- One or a few "trunk" images that have "branch" images, which have branch images of their own, and so on out to the "leaf" images
- **Pro:** Changes to the "trunk" and "branch" images will flow through naturally as images further out in the tree are rebuilt; layering will make space usage very efficient
- **Con:** The blast radius of any errors or breaking changes in a "trunk" or branch image with many children can be huge

Wrapping up

Final thoughts

- **Do what works for you**
- If it stops working, do something else
- Remember the point of all this is supposed to be to make your life *easier*

Further reading and resources

- The OCI image spec
- Podman and Buildah
- Packer
- Buildroot

Thank you!

Slides

Q&A + Labs 4 & 5: Monoliths vs. layers; Inspecting the image in action