CS130: Software Engineering

Lecture 4 Build Systems
Deployment

https://forms.gle/1zy8cx72rkF5Cyc

A word: What builds you up?
A tweet: A comical frustration trying to compile/run software.

Assignment 1 Postmortem

Assignment 1 ~~Postmortem~~ Retrospective

What went well?

What did not go so well?

- Windows?
- Bash?
- git / Gerrit workflow
- Debugging problems on Piazza
- Code reviews take time

Empty config

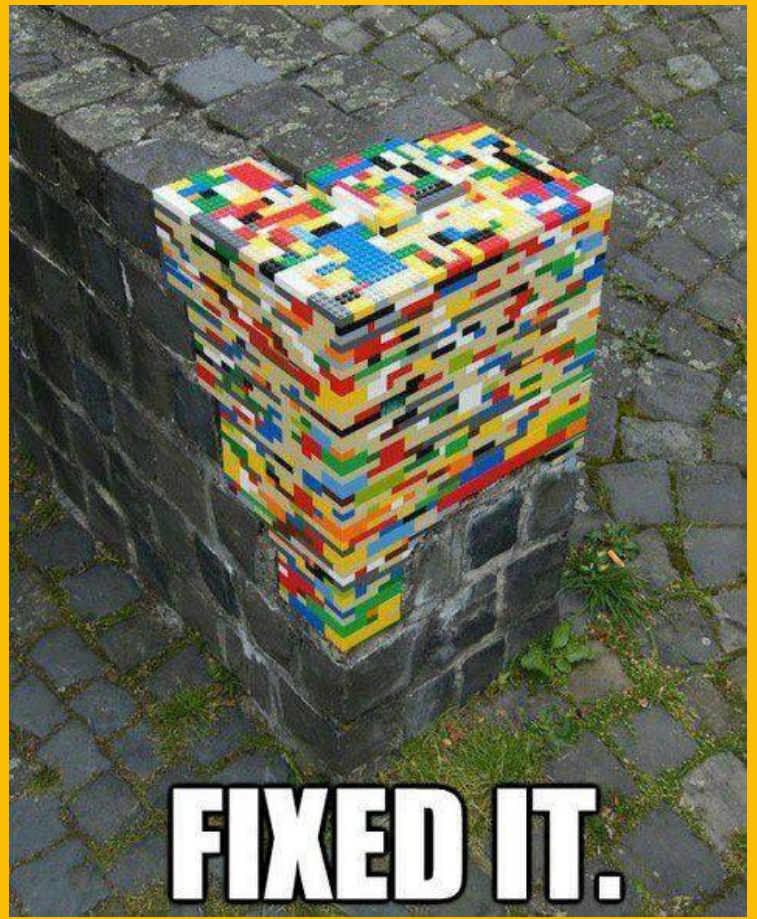
What should your parser do on an empty config?

Blockers

- Any step in your weekly work that is required before you can do any of the future steps
- As an IC (Individual Contributor, i.e. not the TL)
  - After you have decided what piece of the assignment you are responsible for, immediately think through the whole chunk and identify points where you may get blocked.
  - Make sure you get through those sections early in the week!
- As a TL
  - Your #1 priority every day is making sure your team is not blocked
  - Code reviews, effective planning at weekly meeting, etc.

Other things

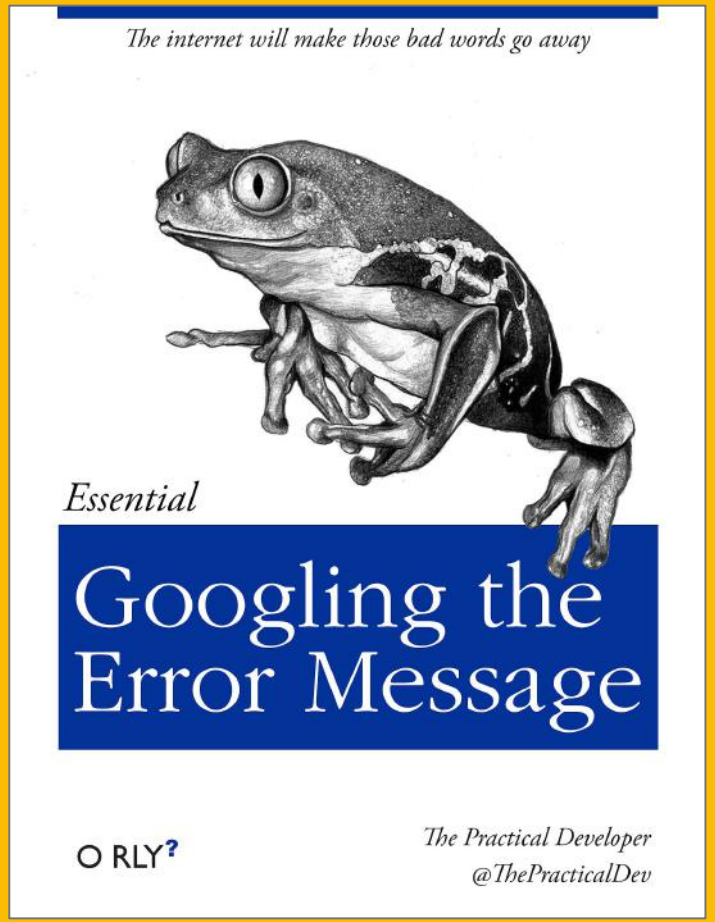
- Ask questions effectively!
- Assignments are intentionally open-ended...
- Everything has Pros and Cons!
- Not everything given to you will be perfect, improve it!
- Practice, practice, practice
- “Safe” and “risky” actions



Assignment 2: In progress

Knowledge you will need

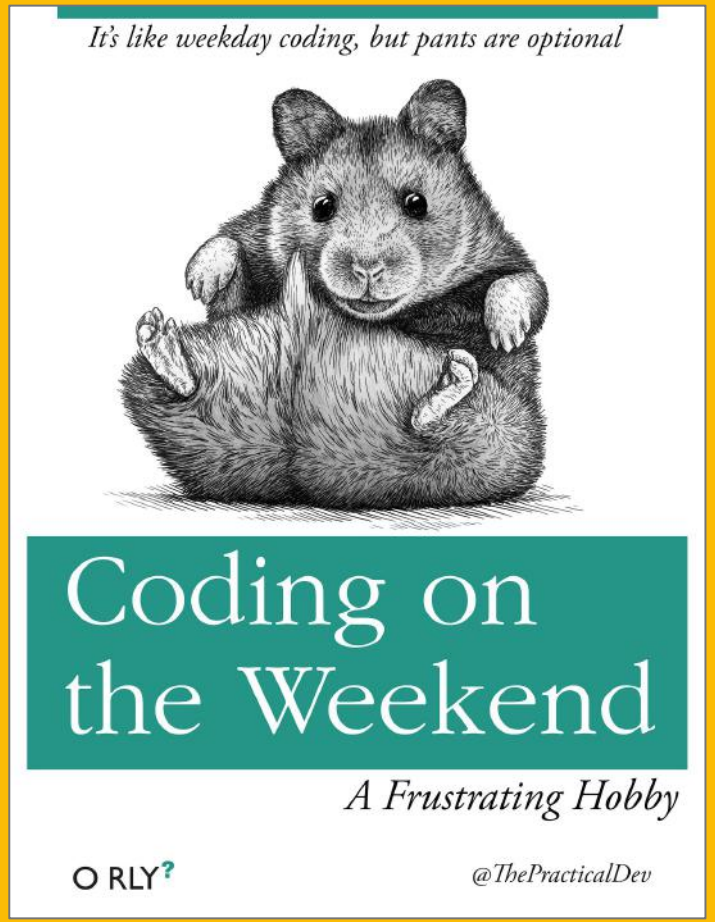
- Git + Gerrit (review)
  - Interactive tutorial
  - CS 130 Guide
- Boost library (new)
  - Sample code
  - Online docs
- HTTP (review)
  - Previous lecture
  - Online spec



Read lots of documentation!

Knowledge you will need

- Docker (new)
  - This lecture
- Google Cloud Web + SDK (new)
  - `gcloud` tutorials
  - CS 130 Guide
  - This lecture



Start early!

Build primer

Quick vocab

“Build”

“Deploy”

Quick vocab

“Build” - Compile your code

(Source code → Executable binary)

“Deploy” - Get your code running

(Run an executable)

In the beginning...

- Run compiler from the command line directly
- Inputs are a few .cc files
- Output is a single executable
- Need to know all the flags

Run:

```
$ g++ config_parser.cc \
    config_parser_main.cc \
    -std=c++0x -g -Wall \
    -o config_parser
```

One step up...

- Wait, what was that flag? It was working yesterday. Grr…
- Write it down

build.sh:

```
#!/bin/bash
g++ config_parser.cc \
  config_parser_main.cc \
  -std=c++0x -g -Wall \
  -o config_parser
```

Run:

```
$ ./build.sh
```

Uh-oh. My laptop has a different compiler...

```
#!/bin/bash

case `uname` in

 Linux) g++ config_parser.cc config_parser_main.cc \
   -std=c++0x -g -Wall -o config_parser;;

 Darwin) clang++ config_parser.cc config_parser_main.cc \
   -std=c++11 -g -Wall -stdlib=libc++ -o config_parser;;

 *) echo "Unknown operating system";;

esac
```

Hmm...I should really have some unit tests.

```
GTEST_DIR="gtest-1.7.0"

g++ -std=c++0x -isystem ${GTEST_DIR}/include \
  -I${GTEST_DIR} -pthread -c ${GTEST_DIR}/src/gtest-all.cc

ar -rv libgtest.a gtest-all.o

g++ -std=c++0x -isystem ${GTEST_DIR}/include \
  -pthread config_parser_test.cc config_parser.cc \
  ${GTEST_DIR}/src/gtest_main.cc libgtest.a \
  -o config_parser_test
```

Lesson 1

Builds should be one simple step

```
$ ./build.sh
```

Lesson 2

Builds should be repeatable

Different engineers, Different times → Same output

Lesson 3

Remember: Valuable things go in revision control

Being able to build your project is valuable

Therefore, build scripts go in revision control

But...

- Bash scripts can be tricky to write
- bash scripts aren’t very readable
- bash scripts aren't easily maintainable

Let’s consider a different tool: make

Building the config parser

Inputs: config_parser.h, config_parser.cc, config_parser_test.cc, config_parser_main.cc

Building the config parser

Inputs: config_parser.h, config_parser.cc, config_parser_test.cc, config_parser_main.cc

Outputs: config_parser_test (a test binary), config_parser_main (a binary)

Building the config parser

Inputs: config_parser.h, config_parser.cc, config_parser_test.cc, config_parser_main.cc

Intermediates: config_parser.o, config_parser_test.o, config_parser_main.o

Outputs: config_parser_test (a test binary), config_parser_main (a binary)

Building the config parser

Inputs: config_parser.h, config_parser.cc, config_parser_test.cc, config_parser_main.cc

↓ Compile

Intermediates: config_parser.o, config_parser_test.o, config_parser_main.o

↓ Link

Outputs: config_parser_test (a test binary), config_parser_main (a binary)

Exposing intermediates in g++

Compile and link in one step:

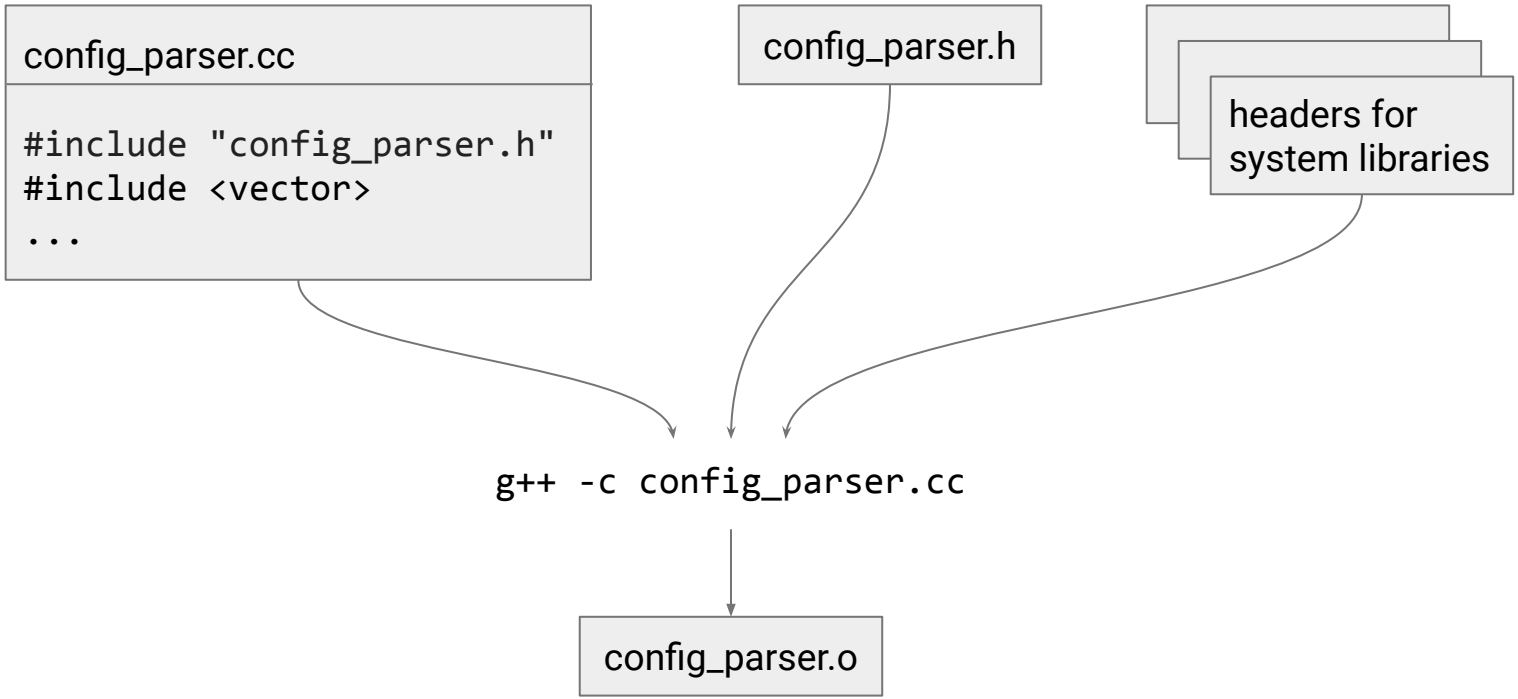
```
$ g++ foo.cc -o foo
```

In two steps:

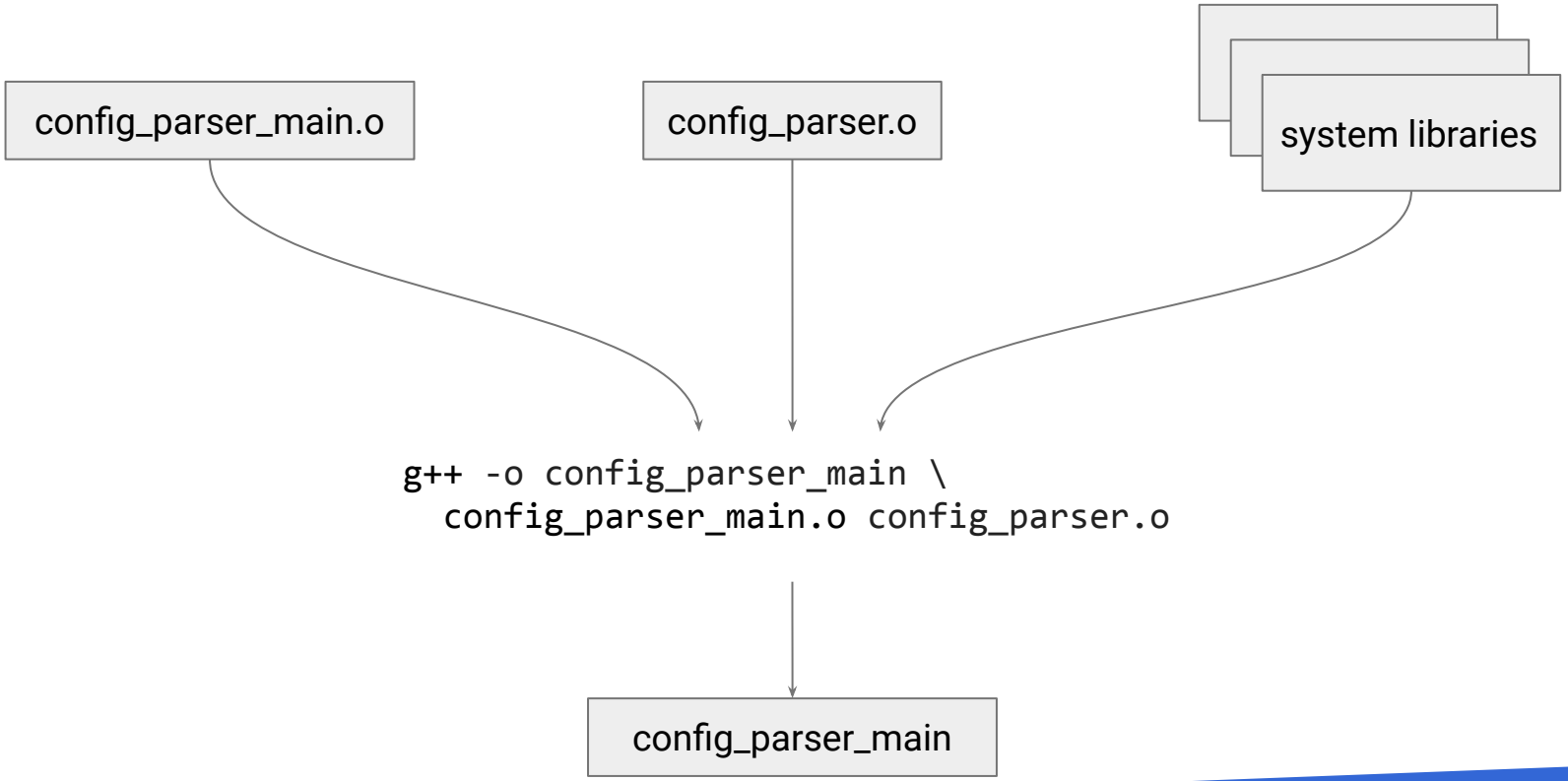
Compilation: `$ g++ -c foo.cc # produces foo.o`

Linking: `$ g++ foo.o -o foo`

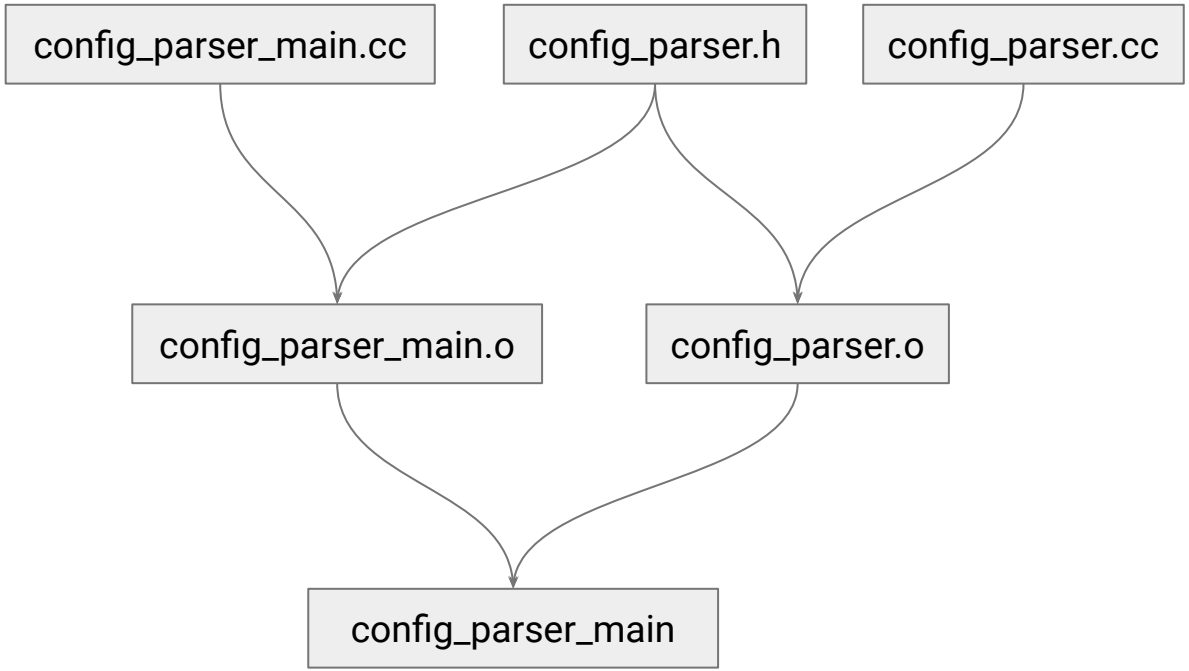
Compilation



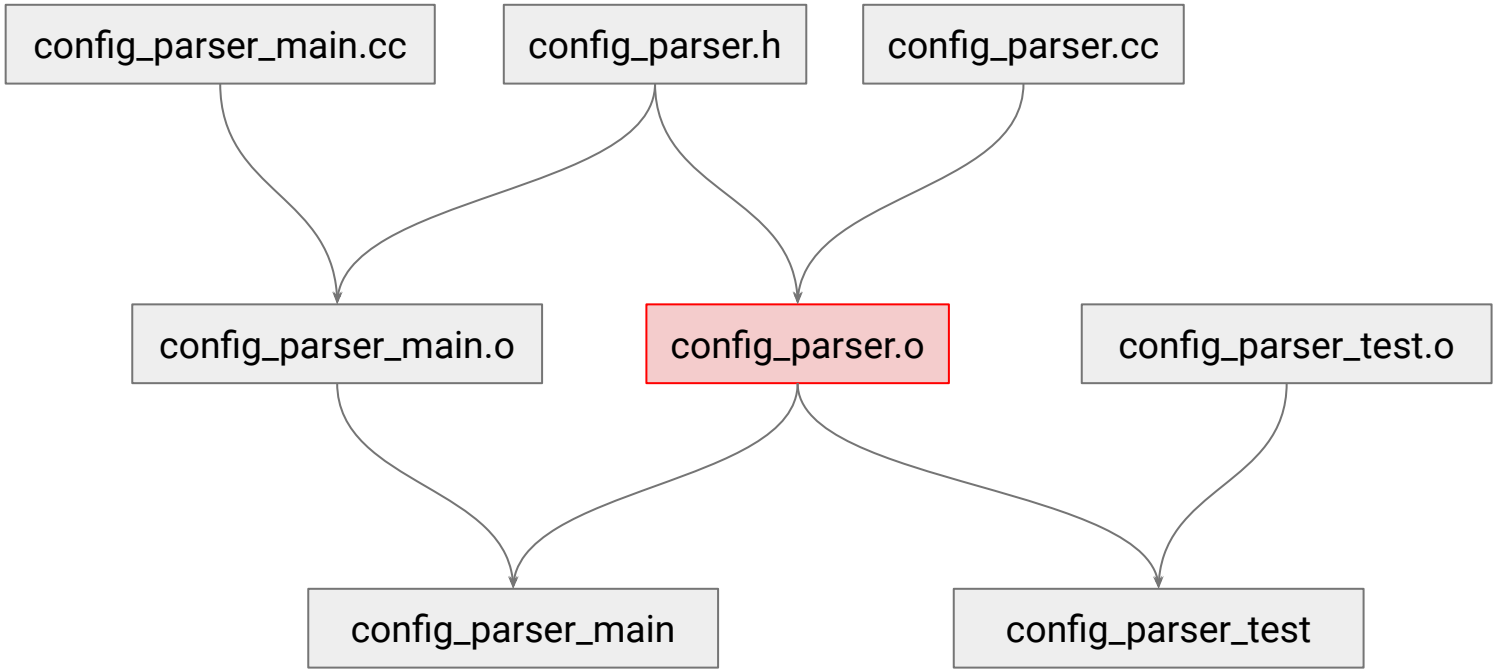
Linking


config_parser_main.o
config_parser.o
system libraries
g++ -o config_parser_main \
config_parser_main.o config_parser.o
config_parser_main

A dependency graph



We can re-use intermediates



Config parser Makefile, version 1

```
config_parser_main: config_parser_main.o config_parser.o
  g++ -o config_parser_main config_parser_main.o config_parser.o \
  -std=c++11 -Wall -Werror


config_parser.o: config_parser.cc config_parser.h
  g++ -c config_parser.cc -std=c++11 -Wall -Werror


config_parser_main.o: config_parser_main.cc config_parser.h
  g++ -c config_parser_main.cc -std=c++11 -Wall -Werror
```

Config parser Makefile, version 2

```
CXXFLAGS=-std=c++11 -Wall -Werror

config_parser_main: config_parser_main.o config_parser.o
    $(LINK.cc) $^ $(LDLIBS) -o $@


config_parser.o: config_parser.cc config_parser.h
    $(CXX) $(CXXFLAGS) -c $<


config_parser_main.o: config_parser_main.cc config_parser.h
    $(CXX) $(CXXFLAGS) -c $<
```

Who makes the Makefiles?

- Makefiles can get real complex, real fast
- Difficult to:
  - Encode logic into Makefile
  - Detect compilers
  - Detect and build with installed packages

Solution: Generate Makefiles!

GNU Build System

- Autoconf + Automake
- Notable files:
  - Makefile.am
  - configure
  - Makefile.in
  - config.h
- Run:
  - `$ ./configure`
  - `$ make`
- Creates GNU-compatible Makefiles

Also see:

https://en.wikipedia.org/wiki/GNU_Build_System

CMake

- Single tool
- Notable files:
  - CMakeLists.txt
  - build/
  - A ton of generated files
- Run:
  - `$ cd build`
  - `$ cmake ..`
  - `$ make`
- Supports multiple output formats

Also see:

https://en.wikipedia.org/wiki/CMake

CMake examples

Define library:

```
add_library(config_parser src/config_parser.cc) # libconfig_parser.a
```

Define executable:

```
add_executable(config_parser_test tests/config_parser_test.cc) # config_parser_test
target_link_libraries(config_parser_test config_parser gtest_main)
```

Add GTest tests:

```
gtest_discover_tests(config_parser_test WORKING_DIRECTORY
    ${CMAKE_CURRENT_SOURCE_DIR}/tests)
```

CMake examples

Add non-GTest tests:

```
add_test(NAME web_test
    COMMAND ${CMAKE_CURRENT_SOURCE_DIR}/tests/web_test.sh -s "$<TARGET_FILE:webserver>"
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/tests)
```

Include external source package:

```
add_subdirectory(/usr/src/googletest googletest)
```

Include external installed (compiled) library:

```
find_package(Boost 1.50 REQUIRED COMPONENTS signals system)
```

Build system wishlist

Build system wishlist

- A good build system has many desirable features
- Unfortunately, some of them are contradictory

Wishlist: Correct

The build should be a faithful representation of the current state of your source code.

What if a file has changed somewhere, but the build system don't realize it?

- Easy to happen in GNU Make.
- `make clean` to the rescue
- GMake detects changes by timestamps.
  Hashing is slower, but more accurate.

Wishlist: Correct

The build should be as close as possible to what you're going to release

Very hard to achieve!

- Debug vs. optimized builds?
- Instrumented binaries (e.g. to find memory bugs, coverage)
- Extra logging?

Wishlist: Correct

```
#include <assert.h>
main() {
 int a = 1;
 assert(a++);  // Looks like a function, but it's a macro.
 std::cout << a << std::endl;
}
```

```
$ g++ -g test.cc && ./a.out
2

$ g++ -DNDEBUG -Os test.cc && ./a.out
1
```

Wishlist

✓ Correct

Wishlist: Hermetic

Different users, different environments → same result

- What if you're on different OSes?
- What if you're using different compilers?
  - Do you check your compiler into version control? Almost never, but Google does
- What if your shells are configured differently?
  - Different .bashrc, different umask, etc.
- Possible solutions:
  - Build in a virtual machine
  - Build in a Docker container

Wishlist

- ✓ Correct
- ✓ Hermetic

Wishlist: Flexible

- Change compilation and linking options
- Support other languages
- Support custom build rules

Wishlist

- ✓ Correct
- ✓ Hermetic
- ✓ Flexible

Wishlist: Easy to use

- Build in one step
- Easy to set up for your project
- One obvious correct way to do things (Make fails here!)

Wishlist

- ✓ Correct
- ✓ Hermetic
- ✓ Flexible
- ✓ Easy to use

Wishlist: Automatable

Builds should happen automatically, and frequently

- Nightly?
- After every submit? ("continuous build")
- Whenever you save a file you're editing?

Wishlist

- ✓ Correct
- ✓ Hermetic
- ✓ Flexible
- ✓ Easy to use
- ✓ Automatable

Wishlist: Fast

The more builds you want to do, the faster it needs to be

- Tell GMake to run 4 build commands at a time (`-j4`)
  - `make -j4`
- Build cluster
- Cache intermediates. Only build what has changed.

Wishlist

- ✓ Correct
- ✓ Hermetic
- ✓ Flexible
- ✓ Easy to use
- ✓ Automatable
- ✓ Fast

Wishlist: Manages dependencies

Very important for "fast" and "scalable"

In GMake, you have to declare your dependencies:

- foo.cc: `#include "bar.h"`
- Makefile:
  - `foo.o: foo.cc foo.h bar.o`
- But now you have dependencies in two places
- Implicit rules
  - Easier to maintain complex makefiles
  - Don't always manage dependencies very well

Wishlist: Manages dependencies automatically

- Read my code, figure out the dependencies
- Very language-dependent
- Example: ekam
  - Works by exploration
    - .h file: Might want to add current directory to #include path
    - .cc file: Try compiling it. What goes wrong? Missing header?
    - .o file: Does it contain main()? Try linking it into a binary
  - Experimental, C++ only
  - If it can't figure something out, it's hard to intervene
- CMake does this for you!

Wishlist

- ✓ Correct
- ✓ Hermetic
- ✓ Flexible
- ✓ Easy to use
- ✓ Automatable
- ✓ Fast
- ✓ Manages dependencies automatically

Wishlist: Integrated

- Build and test when you send a code review. Display results to reviewer
- Save test results to a database you can query
  - "How many times was this test run in the past week?"
- Works with your packaging and release system
- Can query dependencies
  - "What targets depend on X?"
  - "How does X depend on Y?"

(These are all examples from Google)

Wishlist

- ✓ Correct
- ✓ Hermetic
- ✓ Flexible
- ✓ Easy to use
- ✓ Automatable
- ✓ Fast
- ✓ Manages dependencies automatically
- ✓ Integrated

Wishlist: Scalable

- Supports large codebases
- Supports many users

Wishlist

- ✓ Correct
- ✓ Hermetic
- ✓ Flexible
- ✓ Easy to use
- ✓ Automatable
- ✓ Fast
- ✓ Manages dependencies automatically
- ✓ Integrated
- ✓ Scalable

Break

Building at scale

Scale and change rate at Google

- 95,000 developers, all over the world
- 2B lines of code (86 TB)
- Every week, 15M lines and 250,000 files changed
  - Linux kernel: 15M lines, 40,000 files
- 16,000 changes per hour by developers (4 changes per second)
  - Another 24,000 changes per hour by automated systems (10 changes per second)
- Repository gets 800k QPS at peak, mostly from build and test systems.

Build scalability the typical way

Constrain the change rate of your dependencies

- Separate repositories
- Long-lived release branches

For example:

- Your project depends on a library built by another team.
- Other team builds, tests, releases the library to you once a quarter.
- Side effect: When you get a new release of the library, you spend a week fixing things that have broken.

Build scalability

"Multi-repo"

- Separate repositories
- Release branches
- Library built and released ~quarterly
- Some manual testing possible
- Spend a week fixing things after release
- Builds are fast because you only build YOUR code

Google (“mono-repo”)

- One repository
- Development on HEAD
- You always have the latest version
- Tests must be automated
- Fix things as soon as they are broken
- Have to build the whole codebase to test your code.

Google's problem

We want:

- Fast builds for engineers
- Build and test every commit
- Have to build the whole source tree

A fast compiler is ~20k lines/second → 1 day to build the source tree

We have 10 changes per second

The problem, continued

- You can't ever build from scratch.
- You can't "make clean".

Reasons to "make clean"

- Un-tracked dependencies.
- Builds aren't reproducible.

Google's solution

- Each commit changes only a small part of the source tree.
- If our builds are always correct, we can cache the results

Result:

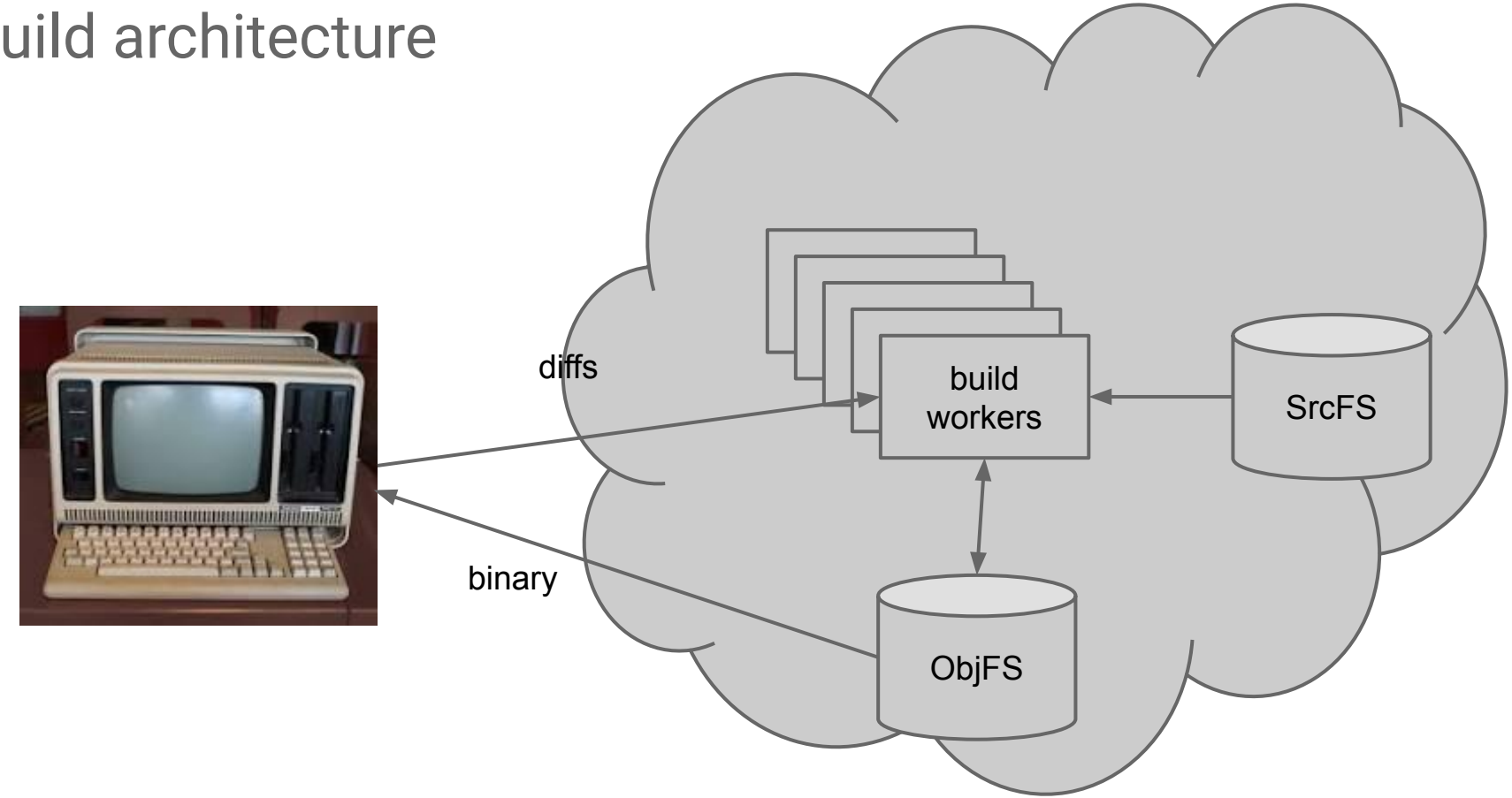
- >90% cache hit rate
- 10k CPU cores
- 50 TB RAM
- 1 PB 7-day cache

But: Achieving correctness is really hard!

Store everything in the cloud

- Engineers work on a small part of the code.
  - CitC stores and snapshots your changes in the cloud
  - FUSE file system makes it transparent (looks like local files you've checked out)
- Send diffs to the cloud to compile
- Fetch back binaries (don’t care about intermediate object files)
- Re-use other people’s builds

Build architecture



Deployment

What is deployment?

What is deployment?

Development

Production

- Server: Running "in the cloud"
- Application: Running on users' phones, computers, etc.

History

Once upon a time…

To run a web server, you must:

- Buy hardware
- Find a place to put your hardware
- Get a network connection
- Install and configure an OS
- Install and configure your software

History

Once upon a time…

To run a web server, you must:

- Buy hardware
- ~~Find a place to put your hardware~~
- ~~Get a network connection~~
- Install and configure an OS
- Install and configure your software

1997-2000: Datacenters

History

Once upon a time…

To run a web server, you must:

- ~~Buy hardware~~
- ~~Find a place to put your hardware~~
- ~~Get a network connection~~
- ~~Install and~~ configure an OS
- Install and configure your software

1997-2000: Datacenters

2000s: Hosting providers, virtualization ("computer rental")

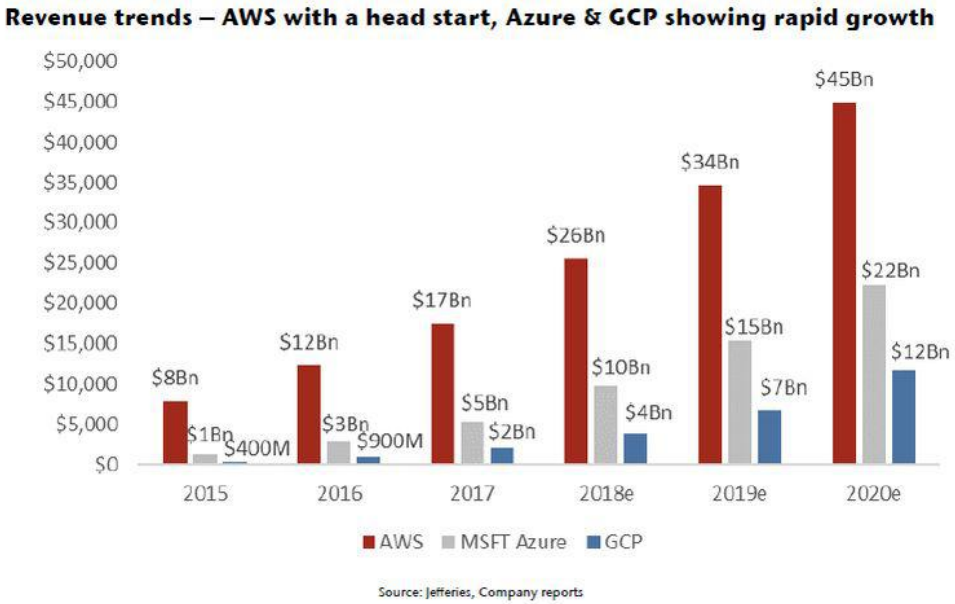
Where are we now?

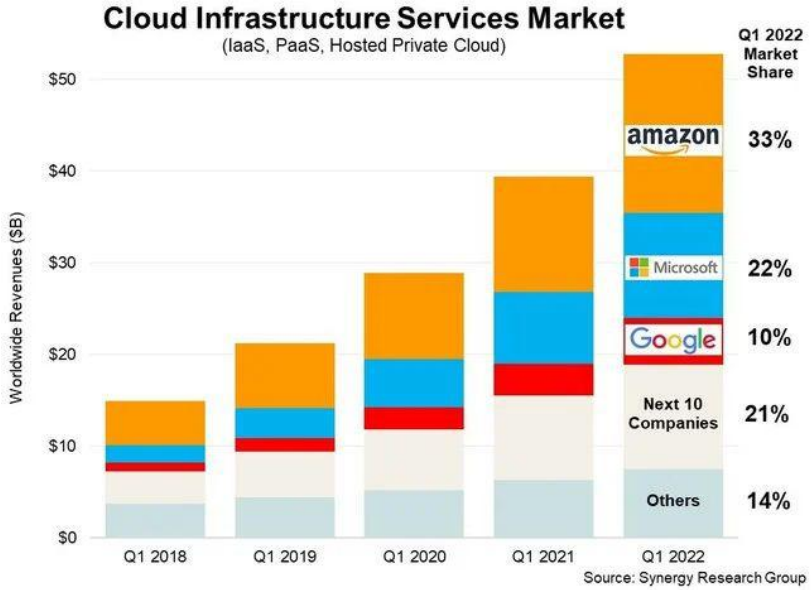
Recent past:

- "Here's a VM image. Please run 200 instances for me"
- Get billed per machine.

Current:

- "Here's my webserver image. Please run enough instances for me."
- Get billed for CPU, RAM, storage.





Birth of a $1T industry? (source)

Observations

- Deployment requires packaging
- Deployment requires isolation and resource management

Virtual machines provides both, but at high overhead

Is there a better way?

An aside on virtualization

- Widely-used term in CS, but hard to define.
- Easiest to define as the antonym of "actual":
  - Virtual memory vs. actual memory
  - Virtual machine vs. actual machine
- Pre-dates computers
  - Turing machine: A virtual computer
  - Universal turing machine:
    A way to run virtual Turing machines. "VMWare for turing machines"

Examples of virtualization

- Virtual functions
  - Specify an interface without an implementation
- Memory virtualization
  - Present a large address space independent of RAM
- Storage virtualization
  - Make many disks on many computers appear as one
- JVM
  - Consistent data, memory, computation model on different architectures
- Machine virtualization

Machine virtualization

Technology	CPU	Hardware	OS	Overhead
Emulation	Different	Different	Different	Higher
OS-level virtualization	Same	Different	Different	
Paravirtualization	Same	Different	Different	
Hypervisor	Same	Same	Different	
Containerization	Same	Same	Same	Lower

Containerization

A restricted view of the underlying OS

- See group of related processes.
- Manage and limit resource usage.
- Separate or restricted filesystem.
- Restrict system calls.

Linux kernel magic

- Namespaces
- cgroups
- OverlayFS
- seccomp

Docker

- Utilities and glue for Linux containerization
- A way to build filesystems in layers
- A package format
  - Basically, a filesystem and config
- A package repository

But wait...I'm running Docker on Mac/Windows!

- Moby Linux VM

Docker concepts

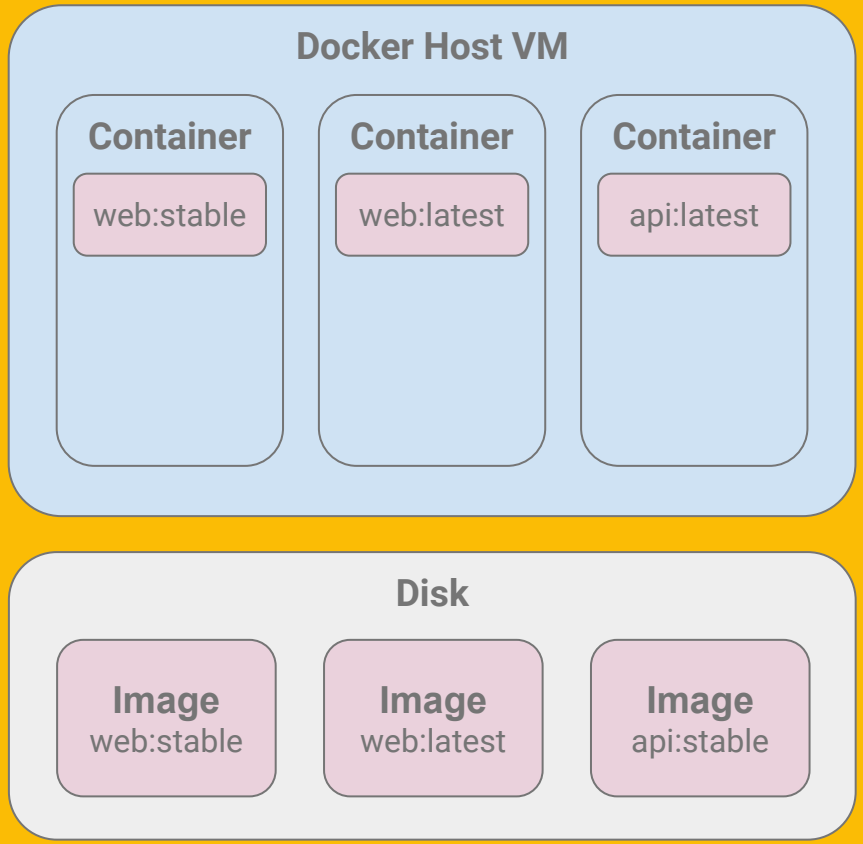
Image

- Snapshot of OS, defined by commands, stacked in layers
- Created by `docker build`
- Defined by Dockerfile
- Base is another image
  - e.g. ubuntu:latest
- Can be tagged with versions
  - e.g. :latest

Docker concepts

Container

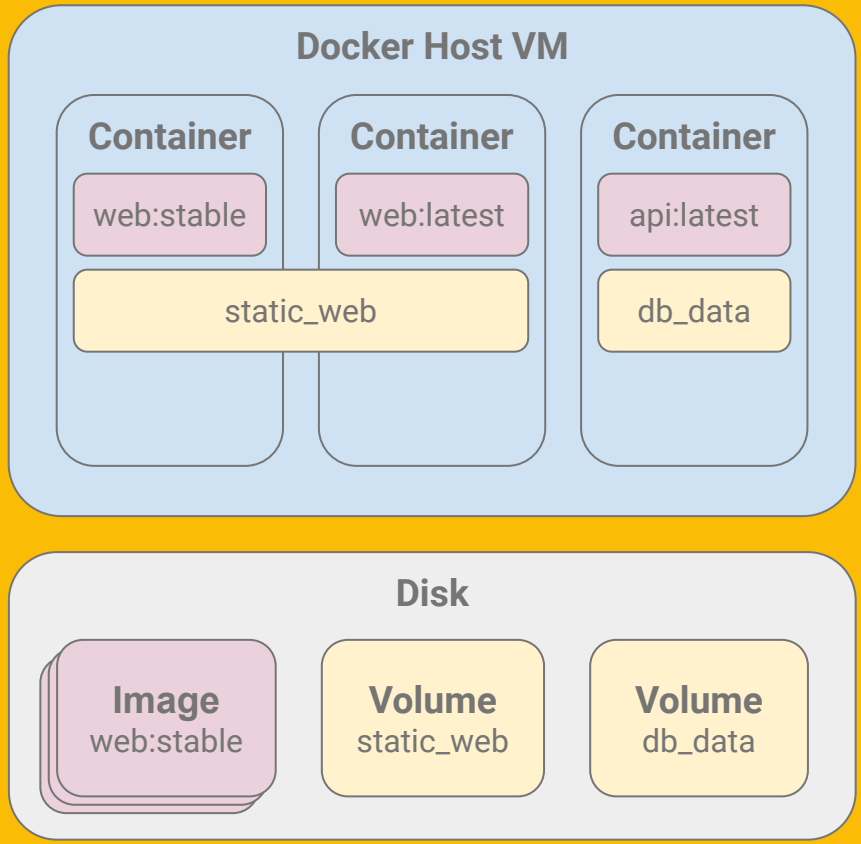
- Running instance of an image
- Created by `docker run`



Docker concepts

Volume

- Virtual disk image
- Created by:
  - `docker run -v/--mount`
  - `docker volume`
- Persists across restarts
- Shareable between containers

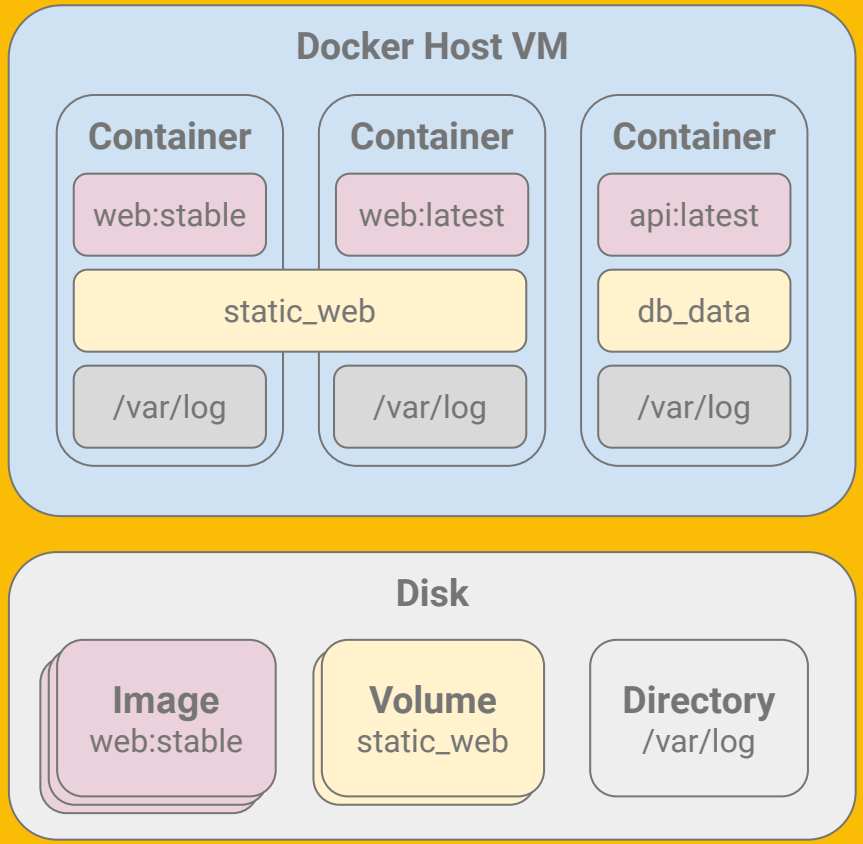


Docker concepts

Bind mount

- Mount host FS in container
- Created by:

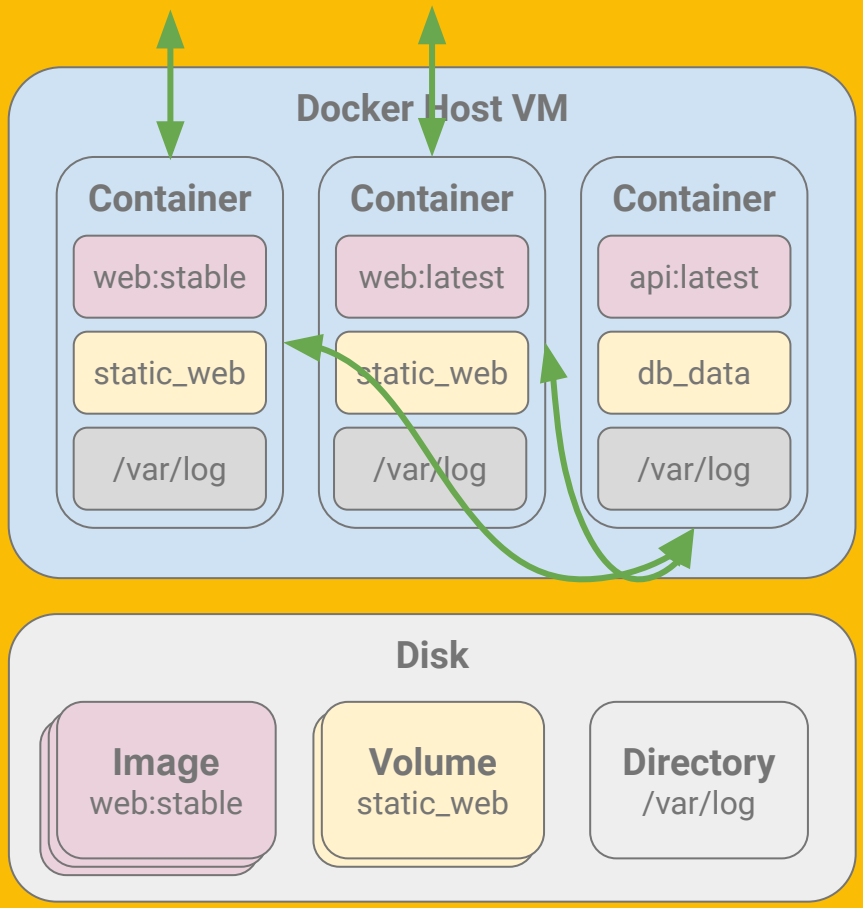
  `docker run -v/--mount`
- Share data between host and container
- Volume that lives in host FS



Docker concepts

Port

- Expose ports inside container to:
  - Host
    - use 127.0.0.1
  - Internet
    - use 0.0.0.0
- Define in:
  - Image (Dockerfile)
    - For documentation
  - Container (`docker run`)
    - For implementation



See also:

https://docs.docker.com/engine/docker-overview/

Example: `base.Dockerfile`

```
# Get the base Ubuntu image from Docker Hub
FROM ubuntu:latest as base

# Update the base image and install build environment
RUN apt-get update && apt-get install -y \
    build-essential \
    cmake \
    ...
```

Example: Dockerfile

```
# Define builder stage
FROM my-proj:base as builder

# Share work directory
COPY . /usr/src/project
WORKDIR /usr/src/project/build

# Build and test
RUN cmake ..
RUN make
RUN ctest --output-on_failure
```

Example: Dockerfile

```
# Define deploy stage
FROM ubuntu:latest as deploy

# Copy server output binary to "."
COPY --from=builder /usr/src/project/build/bin/webserver .

# Copy config files
COPY conf/* conf/

# Expose port 80
EXPOSE 80

# Use ENTRYPOINT to specify the binary name
ENTRYPOINT ["./webserver"]

# Use CMD to specify arguments to ENTRYPOINT
CMD ["conf/deploy.conf"]
```

Container registries

- Storage for built container images
- Local storage:
  - `docker image ls`
- Remote storage:
  - `docker push`
  - `docker pull`
  - Also other tools
- Implicit pull when docker build needs a base image

- Docker Hub
  - Default registry
  - Lots of public images
- Google Container Registry
  - Private to your project
  - Integrated with gcloud tools
- AWS EC2 Container Registry
- Azure Container Registry
- More...

Google Cloud Build

- Define series of build steps in cloudbuild.yaml
- Start a build with some directory/files as the source
  - `gcloud builds submit`
- Build is performed in the cloud
- Build is performed hermetically
  - No context from previous builds!

- Output
  - Build logs
    - Console
    - Web UI
  - Build artifacts
    - Container images
    - Compiled outputs

Build steps

- **Build docker image**

```
- name: 'gcr.io/cloud-builders/docker'
  args: [
    'build',
    '-f', 'docker/Dockerfile',
'-t', 'gcr.io/$PROJECT_ID/mascots:latest',
    '.'
  ]
```

Build steps

- **Build base image**
- Build docker image

```
- name: 'gcr.io/cloud-builders/docker'
  args: [
    'build',
    '-f', 'docker/base.Dockerfile',
    '-t', 'mascots:base',
  '-t', 'gcr.io/$PROJECT_ID/mascots:base',
    '--cache-from',
    'gcr.io/$PROJECT_ID/mascots:base',
    '.'
  ]
```

Build steps

- **Pull base image**
- Build base image
- **Push base image**
- Build docker image

```
- name: 'gcr.io/cloud-builders/docker'
  entrypoint: 'bash'
  args:
    - '-c'
    - |
      docker pull \
        gcr.io/$PROJECT_ID/mascots:base \
        || exit 0
```

(Build base image here)

```
- name: 'gcr.io/cloud-builders/docker'
  args: [
    'push',
    'gcr.io/$PROJECT_ID/mascots:base'
  ]
```

Whole config

- Steps:
  - Pull base image
  - Build base image
  - Push base image
  - Build docker image
- Images:
  - :base
  - :latest

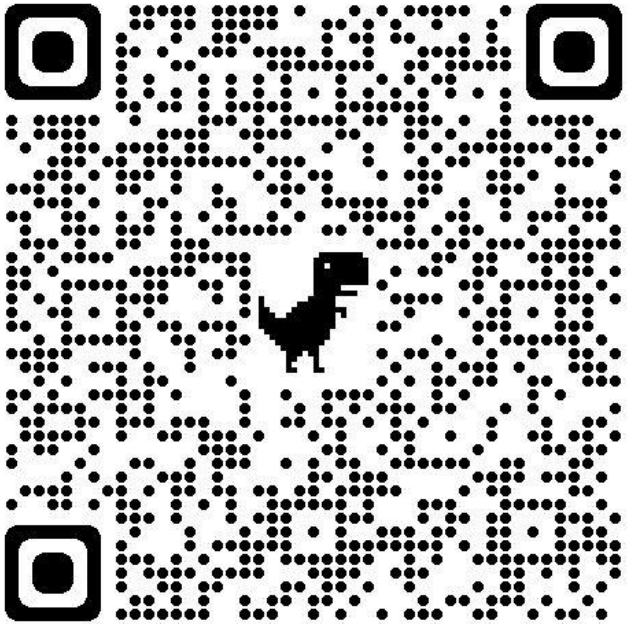
```
steps:
(All steps here)

images: [
  'gcr.io/$PROJECT_ID/mascots:base',
  'gcr.io/$PROJECT_ID/mascots:latest'
]
```

See also:

https://cloud.google.com/cloud-build/docs/how-to
http://www.yamllint.com/

https://bit.ly/38oLgHM

- A word: How do you feel about deploying a webserver?
- A tweet: What fun thing could your server do?



Coming up