# Why are there no good build systems? An introduction to Bazel

Aleksandra Rylund Glesaaen

June 1st 2018

SA2C Tech Chat

# Why do we need build systems?

- Essential for big projects
- Lowers build times
- Incentivises building often
  - Makes finding bugs easier
  - Lowers barrier to testing everything always
- Allows for automation (CI/CD etc)
- A good system allows you to get on with what you are actually there for

# What do I want from a build system?

- Modular
  - Encourages reuse
  - Simplifies dependencies
  - I just really like folders
- Easy to use
- Easy to maintain
- Easy to configure
- Portable
- Extendible to new programming languages
- Handles external dependencies cleanly
- Lends itself to CI/CD

# Let us talk about make

It is the standard for *NIX based development.

Basically shell + dependencies, the rest you have to make yourself; I have too many copies of this snippet:

```
SRCS := $(wildcard src/*.cpp)
DEPS := $(SRCS:src/%.cpp=.deps/%.d)

$(DEPS): .deps/%.d: src/%.cpp | .deps
  @$(CXX) $(CXXFLAGS) -MM -MP -MT \
    bin/$(patsubst src/%.cpp,%,$<) -MF $@ $<

-include $(DEPS)
```

# Make and recursion

*Recursive Make Considered Harmful*

Miller, P.A. (1998), AUUGN Journal

# Make and recursion

- Hard to get build order right
  *Requires a lot of tweaking*
- Inter-directory dependencies very difficult
  *Builds too much or too little*
- Dependencies dropped to decrease build times
  *Users need to run clean periodically*

The *RMCH* paper claims that make does not have a design flaw, I disagree.

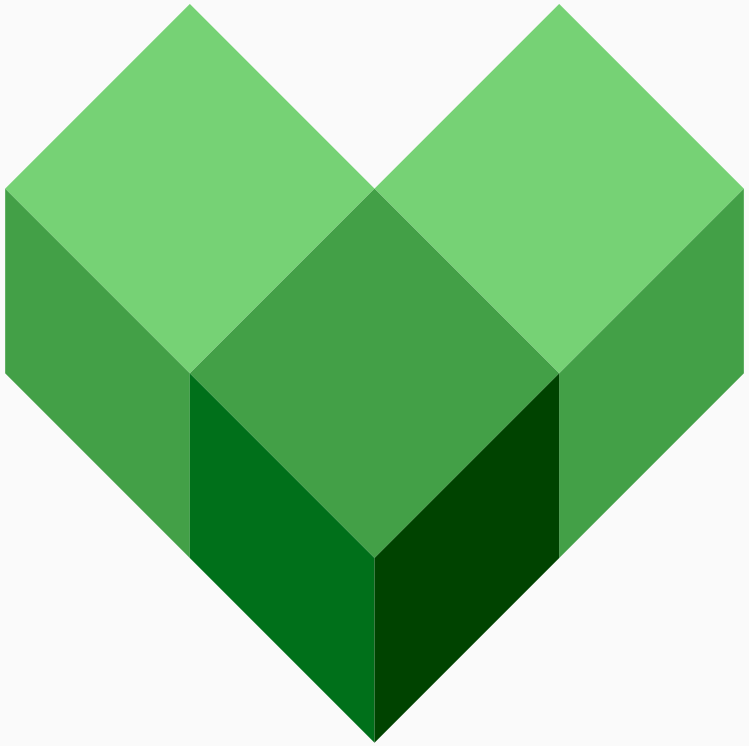
# Makefile generators

Quite a few ”solutions” are simply makefile generators

- autoconf
- non-recursive make
- CMake
- ninja
- ...

We can do better than that!

# Introducing Bazel



# Bazel design principles

- Config files are easy to read
- Bazel is fast and reliable
- Bazel scales
- Bazel is extensible

# Bazel concepts

## Workspace

The folder that contains your project.

May or may not reference external projects.

## Packages

Any directory in your workspace that has a `BUILD` file in it.

Contains related files and dependence specifications.

## Targets

Files and rules the package provides.

Every target has a label.

# Internal dependencies

Bazel sandboxes its builds by default.

You therefore have complete control over what is external and what is internal for your package.

Also means you need to be in control over package dependencies.

Let us look at Example 01

# External dependencies

External as in external to the `WORKSPACE`.

For example `local_repository` and `new_http_archive`.

Interfaces well with other bazel repos, well enough with non-bazel ones.

Let us look at Examples 02, 03, & 04

# The toolchain

So, I wanted to compile an MPI project...

# The toolchain

So, I wanted to compile an MPI project...

*This is a fabulously difficult project that causes hardened engineers to stare blankly at screens in defeat.*

# The toolchain

So, I wanted to compile an MPI project...

*This is a fabulously difficult project that causes hardened engineers to stare blankly at screens in defeat.*

Basically to change any one aspect of the compilation process of bazel you have to replace the entire thing. Let's see an example...

# To sum it up

- Modular (Yes)
- Easy to use (Yes)
- Easy to maintain (Yes)
- Easy to configure (No)
- Portable (sort of)
- Extendible to new programming languages (Not easily)
- Handles external dependencies cleanly (Yes)
- Lends itself to CI/CD (Yes)

# To sum it up

I am not necessarily sold, but as an article by
Windmill Engineering puts it:

Bazel is the worst build system, except for all the others