Building (Python) with Bazel

Benjamin Peterson @ PyCon IL 2016
Blaze to Bazel
Simple example
Hello world example

$ ls
BUILD
greeting.cc
greeting.hh
hello.cc
Makefile
greeting.cc

#include <iostream>

void greet(std::string name)
{
    std::cout << "Hello, " << name << ".
";
}
#pragma once

#include <string>

void greet(std::string);
#include "greeting.hh"

int main() {
    greet("Benjamin");
}

Makefile

libgreeting.a: greeting.cc
    g++ -c -o greeting.o $<
    ar rcs $@ greeting.o

hello: libgreeting.a greeting.hh hello.cc
    g++ -static hello.cc -L. -lгreeting -o $@
$ make hello

g++ -c -o greeting.o greeting.cc
ar rcs libgreeting.a greeting.o

g++ -static hello.cc -L. -lgreeting -o hello

$ ./hello

Hello, Benjamin.
cc_library(
    name = "greeting_lib",
    srcs = ["greeting.cc"],
    hdrs = ["greeting.hh"],
)
cc_binary(
    name = "hello",
    srcs = ["hello.cc"],
    deps = [":greeting_lib"],
)
Trying it out with Bazel

```bash
$ bazel build hello
INFO: Found 1 target...
Target //hello up-to-date:
   bazel-bin/hello
INFO: Elapsed time: 0.365s, Critical Path: 0.09s
$ bazel-bin/hello
Hello, Benjamin.
```
libgreeting.a: greeting.cc

    g++ -c -o greeting.o $<
    ar rcs $@ greeting.o

hello: libgreeting.a greeting.hh hello.cc

    g++ -static hello.cc -L. -lgreeting -o $@
Trying it out

$ make hello
make: `hello' is up to date.

$ ./hello
Hello, Benjamin.
BUILD

cc_binary(
    name = "hello",
    srcs = ["hello.cc"],
    deps = [":greeting_lib"],
)

Bazel with missing dep

$ bazel build hello

INFO: Found 1 target...

ERROR: BUILD:1:1: undeclared inclusion(s) in rule '//hello':

this rule is missing dependency declarations for the following files included by 'hello.cc':

'greeting.hh'.

Target //hello failed to build
make(1) problems

- Based on modified times
- No verification of dependencies
- Not scalable for large projects
- Unqueryable
The ideal build step according to Bazel
C++ building

A C++ file
Headers
Toolchain
Flags
Runtime

gcc

*.o file
Why hermeticity?

- Perfect incrementality
- “Change pruning”
- Distributed building and caching
- Reproducible builds across dev/test/production
- Discover dependency bugs
Python example
Hello world example for Python

$ ls
BUILD

greeting.py

hello.py
greeting.py

def greet(name):
    print("Hello, {}.").format(name))

hello.py

from greeting import greet
greet("Benjamin")
BUILD

py_library(
    name = 'greeting',
    srcs = ['greeting.py'],
)

...
BUILD (continued)

def py_binary(
    name = 'hello',
    main = 'hello.py',
    srcs = ['hello.py'],
    deps = [':greeting'],
)

Trying it out with Bazel

$ bazel build hello

$ bazel-bin/hello

Hello, Benjamin.
Structure of the Bazel package

$ ls bazel-bin

hello

hello.runfiles

$ ls bazel-bin/hello.runfiles

greeting.py

hello.py

hello
bazel query
$ bazel query 'deps(hello)'
greeting
greeting.py
hello.py

$ bazel query 'kind("source file", deps(hello))'
greeting.py
hello.py
Fancy bazel queries

kind("cc_library", deps(kind(".*_test", foo/...)) except deps(foo_bin))

somepath(foo/..., //bar/baz:all)
Bazel limitations

- Existing build processes are rarely hermetic
- Most useful when everything uses it
- Doesn’t completely work on Windows (yet)
- Have to build dependency checking for Python
- Python developers don’t like “building”
Questions?

benjamin@python.org