Using Bazel for building and testing C++/Python projects

Stéphane Caron

February 20, 2024

Inria

Outline

- Features
- My experience
- Limitations
- Takeaways

Bazel

Bazel is a build and test system:

```
$ bazel build //some/path:target
$ bazel run //some/path:target
$ bazel test //some/...
```

Features:

- **Fast:** local and distributed caching, dependency analysis
- **Multi-language:** C++, Python, Java, Go, Android, iOS, ...
- **Multi-platform:** Linux, macOS, Windows
- **Extensible:** Starlark configuration language, a subset of Python

Workspace

Bazel reads a `WORKSPACE` file[1] at the root of the repository:

```
workspace(name = "project_name")

http_archive(
    name = "palimpsest",
    sha256 = "244ffe888888bc12d6d5270020993a79e56ddb38f2beafa7647f17cf0192d4c9",
    strip_prefix = "palimpsest-2.0.0",
    url = "https://github.com/upkie/palimpsest/archive/refs/tags/v2.0.0.tar.gz",
)

git_repository(
    name = "upkie_description",
    remote = "https://github.com/upkie/upkie_description",
    commit = "bb886d0f453c2d6822d431cfd42385bf06052b42",
    shallow_since = "1687961108 +0200"
)
```

[1]This presentation is for Bazel < 7.0, with workspaces rather than modules.

Anatomy of a Bazel rule

Bazel reads rules from `BUILD` files in each directory, like so:

```
cc_binary(
    name = "bullet_spine",
    srcs = ["bullet_spine.cpp"],
    data = ["@upkie_description"],
    deps = [
        "//upkie/config:layout",
        "//upkie/observers",
        "//upkie/utils:datetime_now_string",
        "//upkie:version",
        "@vulp//vulp/actuation:bullet_interface",
        "@vulp//vulp/observation",
        "@vulp//vulp/observation/sources",
        "@vulp//vulp/spine",
    ],
)
```

Anatomy of a Bazel rule

Python targets work the same:

```
py_library(
    name = "upkie_base_env",
    srcs = ["upkie_base_env.py"],
    deps = [
        "//upkie/config",
        "//upkie/observers/base_pitch",
        "//upkie/utils:exceptions",
        "//upkie/utils:nested_update",
        "//upkie/utils:robot_state",
        "@vulp//:python",
    ],
)
```

My experience with Bazel

- Had to use it anyway ;-)
- Strict hermeticity feels right
- Only dependency and target definitions in a project: feels *very* right[2]
- Cross-compilation toolchain was painless to use
- More complex custom use cases: if not already done somewhere, brace!
- Main drawback: Python dependencies from PyPI, coming up now...

[2] Looking at you, CMake...

PyPI rules

In the `WORKSPACE` file:

```
load("@rules_python//python:pip.bzl", "pip_parse")

pip_parse(
    name = "pip_vulp",
    requirements_lock = Label("//tools/workspace/pip_vulp:requirements_lock.txt"),
)

load("@pip_vulp//:requirements.bzl", "install_deps")
install_deps()
```

Install PyPI deps *hermetically*. Con #1: breaks some package distributions!

Example with msgpack on macOS

Con #2: a sneakier failure mode we ran into:

```
$ ./tools/bazel run //pink_balancer -- -c bullet
INFO: Analyzed target //pink_balancer:pink_balancer (56 packages loaded, 1507 targets
INFO: Found 1 target...
Target //pink_balancer:pink_balancer up-to-date:
  bazel-bin/pink_balancer/pink_balancer
INFO: Elapsed time: 14.460s, Critical Path: 0.14s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Running command line: bazel-bin/pink_balancer/pink_balancer -c bullet
Traceback (most recent call last):
  File "bazel-out/darwin-opt/bin/pink_balancer/pink_balancer.runfiles/pink_balancer/pi
    spine = SpineInterface()
  File "bazel-out/darwin-opt/bin/pink_balancer/pink_balancer.runfiles/vulp/vulp/spine/
    self.__perf_checks()
  File "bazel-out/darwin-opt/bin/pink_balancer/pink_balancer.runfiles/vulp/vulp/spine/
    raise PerformanceIssue("msgpack is running in pure Python")
vulp.spine.exceptions.PerformanceIssue: msgpack is running in pure Python
```

Takeaways

- Multi-language, multi-platform: ✓
- C++ build and test: ✓
- Cross-compilation: ✓
- Python build and test: ✓
- Python dependencies from PyPI[3]: —

[3] Now considering Conda...

Thank you for your attention!