Backend code, even more choices

Just like with the frontend, the backend gives you multiple choices of languages and feature sets for working with Bazel. C/C++, Go and Rust are the three languages that I tested generating WASM code with so it may be interesting to see how well they work with Bazel and how they abstract WASM generation.

We can keep all the code for our project in a single monorepo and let Bazel sort out what tools to use to compile wich part.

I use C and C++ as the baseline for testing Bazel with backend languages. It also works slightly different from other languages as it is built into Bazel and doesn't require the rules to be installed manually.

As far as backend code goes, Go is my favorite language. However, the Bazel toolchain's way of compiling Go code to WASM is not intuitive.

I'm also exploring Rust toolchains for Bazel because it provides better WebAssembly tooling than Go, but it's still a challenge to get it right every time.

Backend code: Go

I've always been interested in Go as a backend language or something that will allow me to create WASM code to run on the web.

The first step, as usual, is to load the rules in the `WORKSPACE`.

```
http_archive(
  name = "io_bazel_rules_go",
  sha256 = "6f111c57fd50baf5b8ee9d63024874dd2a014b069426156c55adbf6d3d22cb
  urls = [
    "https://mirror.bazel.build/github.com/bazelbuild/rules_go/releases/do
    "https://github.com/bazelbuild/rules_go/releases/download/v0.25.0/rule
  ],
)
```

```
load("@io_bazel_rules_go//go:deps.bzl", "go_register_toolchains", "go_rule

go_rules_dependencies()

go_register_toolchains(version = "1.15.5")
```

This must be the first set of rules to appear in the global WORKSPACE or be in its own WORKSPACE, otherwise it will cause an error.

The ruleset provides the following core rules

- Core rules
  - go_binary
  - go_library
  - go_test
  - go_source
  - go_path

For more information see the `rules_go` README.

The BUILDFILE is simple. We load the `go_binary` rule and use it to compile the file we want. All the source files must be in the `main` package.

```
load("@io_bazel_rules_go//go:def.bzl", "go_binary")

go_binary(
  name = "hello_go",
  srcs = ["hello.go"],
)
```

The code for `hello.go` is the basic 'hello world` example that builds a single binary executable.

```
package main

import "fmt"
```

```
func main() {
  fmt.Println("hello world")
  fmt.Println("I'm learning Blaze and Go")
}
```

To create packages we use `go_library` instead of `go_binary`. All the source files must be part of the same package for this to work.

```
go_library(
  name = "go_default_library",
  srcs = [
    "foo.go",
    "bar.go",
  ],
  deps = [
 "foo.go"//tools:go_default_library", "@org_golang_x_utils//stuff:go_defau
  ]"@org_golang_x_utils//stuff:go_default_library"o",
  visibility = ["//visibility:public"],
)
",
  visibility = ["//visibility:public"],//visibility:public"],
)
```

These libraries will not generate an executable and are meant to be consumed by binaries or other libraries.

Cross-compiling to WASM

While not as intuitive as Rust, Go also allows you to compile code into WASM.

`rules_go` can cross-compile Go projects to any platform the Go toolchain supports. The simplest way to do this is by setting the `--platforms` flag on the command line.

```
bazel build --platforms=@io_bazel_rules_go//go/toolchain:linux_amd64 //my
```

You can replace `linux_amd64` in the example above with any valid `GOOS` /

GOARCH pair. To list all platforms, run this command:

```
bazel query 'kind(platform, @io_bazel_rules_go//go/toolchain:all)'
```

By default, cross-compilation will cause Go targets to be built in “pure mode”, which disables cgo; cgo files will not be compiled, and C/C++ dependencies will not be compiled or linked.

To compile a Go target into WASM, run the following command

```
bazel build --platforms=@io_bazel_rules_go//go/toolchain:js_wasm //go:hel
```

Note that this will change all future builds of that particular Go package to WASM. To change it back you’ll have to build again with a toolchain matching the GOOS / GOARCH you want to build for.

```
bazel build \
--platforms=@io_bazel_rules_go//go/toolchain:darwin_amd64 \
//go:hello_go
```

Rust

Bazel also provides a set of rules to work with Rust and it’s ecosystem. The idea, as with Go and C/C++, is to be able to code backend software in the same repository as the frontend or be able to create WASM modules to address Javascript hot path bottlenecks in the frontend.

To get started add the following block to the WORKSPACE file:

```
http_archive(
  name = "io_bazel_rules_rust",
  sha256 = "0e2e633bf0f7f25392ffb477d677c88eb34fe70ffae"0e2e633bf0f7f2539
  urls = [
    # Master branch as of 2020-12-05
    "https://github.com/bazelbuild/rules_rust/archive/bc0578798f50d018ca4
```

```
  ],
)

load("@io_bazel_rules_rust//rust:repositories.bzl", "rust_repositories")

rust_repositories()
"https://github.com/bazelbuild/rules_rust/archive/bc0578798f50d018ca4278a
```

The file we're using to test the build is simple, it's a hello world example in Rust.

```
// This is the main function
fn main() {
    // Print text to the console
    println!("Hello World!");
    println!("I'm a Rustacean in training!")
}
"Hello World!"
```

The `BUILD` command is simple.

```
load("@io_bazel_rules_rust//rust:rust.bzl", "rust_binary")

rust_binary(
    name = "hello_rust",
    srcs = ["src/main.rs"],
)
```

However, there is a problem with the build. I generated the repository with Cargo using `cargo init`. But the documentation for `rust_rules` proposes a different system, one that doesn't use the language's built-in tools, at least not directly.

This means that the project either works with the standard Rust tools or with Bazel, not with both.

If you look at the example on Github you will see that it has no `Cargo.toml` or `Cargo.lock` but it still build the binaries and libraries when using Bazel.

The project also depends on a library crate also located as a top level child on the workspace. Moving them to a directory under the workspace (meaning they are not at the top level) will result in an error because Bazel cannot find the BUILDFILE for either crate.

To run the code in `rust2/hello_rust` from the root of the project, use the following command:

```
bazel build //rust2/hello_rust:all
```

Bazel has no concept of default build tasks so the `:all` component is important. Using a command like this means you can skip a `BUILD` file at the root of every package :)

Using Rust External Dependencies

Currently the most common approach to managing external dependencies is using cargo-raze to generate BUILD files for Cargo crates.

This sounds like the best way to bring an existing Rust project into Bazel.

WebAssembly

To build a rust_binary for `wasm32-unknown-unknown` target add the `--platforms=@io_bazel_rules_rust//rust/platform:wasm` flag to your `bazel build` command.

```
bazel build @examples//hello_world_wasm /
--platforms=@io_bazel_rules_rust//rust/platform:wasm
```

To build a rust_binary for wasm32-wasi target add the --platforms=@io_bazel_rules_rust//rust/platform:wasi flag.

```
bazel build @examples//hello_world_wasm /
--platforms=@io_bazel_rules_rust//rust/platform:wasi
```

`rust_wasm_bindgen` will automatically transition to the wasm platform and can

be used when building WebAssembly code for the host target.

C++

If you will work with C++ code on Bazel it's slightly different than working with Go or Rust.

Loading rules for C++ is not necessary. The rules are built into Bazel so we can safely skip loading them in the WORKSPACE we want to work on so we can concentrate on the code and the `BUILD` files.

The first example builds a binary from a single file without dependencies.

This is the `BUILD` file we use to build the package:

```
load("@rules_cc//cc:defs.bzl", "cc_binary")

cc_binary(
  name = "hello-world",
  srcs = ["hello-world.cc"],
)
```

Note that we load the `cc_binary` rule from the `rules_cc` rule set, even though we didn't load it into the workspace, it just works.

The second example build a library and integrates it with a binary.

The buildfile defines two targets, a library and a binary.

The `hello_greet` library target uses a C++ file and a header.

The `hello-world` binary target uses a single C++ file and depends on the library we created.

```
load("@rules_cc//cc:defs.bzl", "cc_binary", "cc_library")

cc_library(
    name = "hello-greet",
```

```
    srcs = ["hello-greet.cc"],
    hdrs = ["hello-greet.h"],
)

cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
    deps = [
        ":hello-greet",
    ],
)
```

See the cpp_tutorial in the Bazel examples repository for the code for these examples.