**Building gRPC services with bazel and rules\_protobuf**

**Posted on Thursday, October 13, 2016 by Paul Cody Johnston**

[gRPC](https://grpc.io/) makes it easier to build high-performance microservices by providing generated service entrypoints in a variety of different languages. [Bazel](https://bazel.io) complements these efforts with a capable and fast polyglot build environment.

[rules\_protobuf](https://github.com/pubref/rules_protobuf) extends bazel and makes it easier develop gRPC services.

It does this by:

1. Building protoc (the protocol buffer compiler) and all the necessary protoc-gen-\* plugins.
2. Building the protobuf and gRPC libraries required for gRPC-related code to compile.
3. Abstracting away protoc plugin invocation (you don’t have to necessarily learn or remember how to call protoc).
4. Regenerating and recompiling outputs when protobuf source files change.

In this post I’ll provide background about how bazel works ([Part 1](https://grpc.io/blog/bazel_rules_protobuf/#about-bazel)) and how to get started building gRPC services with rules\_protobuf ([Part 2](https://grpc.io/blog/bazel_rules_protobuf/#building-a-grpc-service-with-rulesprotobuf)). If you’re already a bazel aficionado, you can skip directly to Part 2.

To best follow along, [install bazel](https://www.bazel.io/versions/master/docs/install.html) and clone the rules\_protobuf repository:

~$ git clone https://github.com/pubref/rules\_protobuf

~$ cd rules\_protobuf

~/rules\_protobuf$

Great. Let’s get started!

**1: About Bazel**

[Bazel](https://www.bazel.io/) is Google’s open-source version of their internal build tool called “Blaze”. Blaze originated from the challenges of managing a large monorepo with code written in a variety of languages. Blaze was the inspiration for other capable and fast build tools including [Pants](https://www.pantsbuild.org/) and [Buck](https://buckbuild.com/). Bazel is conceptually simple but there are some core concepts & terminology to understand:

1. **Bazel command**: a function that does some type of work when called from the command line. Common ones include bazel build (compile a libary), bazel run (run a binary executable), bazel test (execute tests), and bazel query (tell me something about the build dependency graph). See all with bazel help.
2. **Build phases**: the three stages (loading, analysis, and execution) that bazel goes through when calling a bazel command.
3. **The WORKSPACE file**: a required file that defines the project root. It is primarily used to declare external dependencies (external workspaces).
4. **BUILD files**: the presence of a BUILD file in a directory defines it as a *package*. BUILD files contain *rules* that define *targets* which can be selected using the *target pattern syntax*. Rules are written in a python-like language called [*skylark*](https://bazel.io/versions/master/docs/skylark/index.html). Syklark has stronger deterministic guarantees than python but is intentionally minimal, excluding language features such as recursion, classes, and lambdas.

**1.1: Package Structure**

To illustrate these concepts, let’s look at the package structure of the [rules\_protobuf examples subdirectory](https://github.com/pubref/rules_protobuf/tree/master/examples). Let’s look at the file tree, showing only those folder having a BUILD file:

$ tree -P 'BUILD|WORKSPACE' -I 'third\_party|bzl' examples/

.

├── BUILD

├── WORKSPACE

└── examples

   ├── helloworld

   │   ├── cpp

   │   │   └── BUILD

   │   ├── go

   │   │   ├── client

   │   │   │   └── BUILD

   │   │   ├── greeter\_test

   │   │   │   └── BUILD

   │   │   └── server

   │   │   └── BUILD

   │   ├── grpc\_gateway

   │   │   └── BUILD

   │   ├── java

   │   │   └── org

   │   │   └── pubref

   │   │   └── rules\_protobuf

   │   │   └── examples

   │   │   └── helloworld

   │   │   ├── client

   │   │   │   └── BUILD

   │   │   └── server

   │   │   └── BUILD

   │   └── proto

   │      └── BUILD

   └── proto

   └── BUILD

**1.2: Targets**

To get a list of targets within the examples/ folder, use a query. This says *“Ok bazel, show me all the callable targets in all packages within the examples folder, and say what kind of thing it is in addition to its path label”*:

~/rules\_protobuf$ bazel query //examples/... --output label\_kind | sort | column -t

cc\_binary rule //examples/helloworld/cpp:client

cc\_binary rule //examples/helloworld/cpp:server

cc\_library rule //examples/helloworld/cpp:clientlib

cc\_library rule //examples/helloworld/proto:cpp

cc\_library rule //examples/proto:cpp

cc\_proto\_compile rule //examples/helloworld/proto:cpp.pb

cc\_proto\_compile rule //examples/proto:cpp.pb

cc\_test rule //examples/helloworld/cpp:test

filegroup rule //examples/helloworld/proto:protos

filegroup rule //examples/proto:protos

go\_binary rule //examples/helloworld/go/client:client

go\_binary rule //examples/helloworld/go/server:server

go\_library rule //examples/helloworld/go/server:greeter

go\_library rule //examples/helloworld/grpc\_gateway:gateway

go\_library rule //examples/helloworld/proto:go

go\_library rule //examples/proto:go\_default\_library

go\_proto\_compile rule //examples/helloworld/proto:go.pb

go\_proto\_compile rule //examples/proto:go\_default\_library.pb

go\_test rule //examples/helloworld/go/greeter\_test:greeter\_test

go\_test rule //examples/helloworld/grpc\_gateway:greeter\_test

grpc\_gateway\_proto\_compile rule //examples/helloworld/grpc\_gateway:gateway.pb

java\_binary rule //examples/helloworld/java/org/pubref/rules\_protobuf/examples/helloworld/client:netty

java\_binary rule //examples/helloworld/java/org/pubref/rules\_protobuf/examples/helloworld/server:netty

java\_library rule //examples/helloworld/java/org/pubref/rules\_protobuf/examples/helloworld/client:client

java\_library rule //examples/helloworld/java/org/pubref/rules\_protobuf/examples/helloworld/server:server

java\_library rule //examples/helloworld/proto:java

java\_library rule //examples/proto:java

java\_proto\_compile rule //examples/helloworld/proto:java.pb

java\_proto\_compile rule //examples/proto:java.pb

js\_proto\_compile rule //examples/helloworld/proto:js

js\_proto\_compile rule //examples/proto:js

py\_proto\_compile rule //examples/helloworld/proto:py.pb

ruby\_proto\_compile rule //examples/proto:rb.pb

We’re not limited to targets in our own workspace. As it turns out, the [Google Protobuf repo](https://github.com/google/protobuf) is named as an external repository (more on this later) and we can also address targets in that workspace in the same way. Here’s a partial list:

~/rules\_protobuf$ bazel query @com\_github\_google\_protobuf//... --output label\_kind | sort | column -t

cc\_binary rule @com\_github\_google\_protobuf//:protoc

cc\_library rule @com\_github\_google\_protobuf//:protobuf

cc\_library rule @com\_github\_google\_protobuf//:protobuf\_lite

cc\_library rule @com\_github\_google\_protobuf//:protoc\_lib

cc\_library rule @com\_github\_google\_protobuf//util/python:python\_headers

filegroup rule @com\_github\_google\_protobuf//:well\_known\_protos

java\_library rule @com\_github\_google\_protobuf//:protobuf\_java

objc\_library rule @com\_github\_google\_protobuf//:protobuf\_objc

py\_library rule @com\_github\_google\_protobuf//:protobuf\_python

...

This is possible because the protobuf team provides a [BUILD file](https://github.com/google/protobuf/blob/master/BUILD) at the root of their repository. Thanks Protobuf team! Later we’ll learn how to “inject” our own BUILD files into repositories that don’t already have one.

Inspecting the list above, we see a cc\_binary rule named protoc. If we bazel run that target, bazel will clone the protobuf repo, build all the dependent libraries, build a pristine executable binary from source, and call it (pass command line arguments to binary rules after the double-dash):

~/rules\_protobuf$ bazel run @com\_github\_google\_protobuf//:protoc -- --help

Usage: /private/var/tmp/\_bazel\_pcj/63330772b4917b139280caef8bb81867/execroot/rules\_protobuf/bazel-out/local-fastbuild/bin/external/com\_github\_google\_protobuf/protoc [OPTION] PROTO\_FILES

Parse PROTO\_FILES and generate output based on the options given:

-IPATH, --proto\_path=PATH Specify the directory in which to search for

imports. May be specified multiple times;

directories will be searched in order. If not

given, the current working directory is used.

--version Show version info and exit.

-h, --help Show this text and exit.

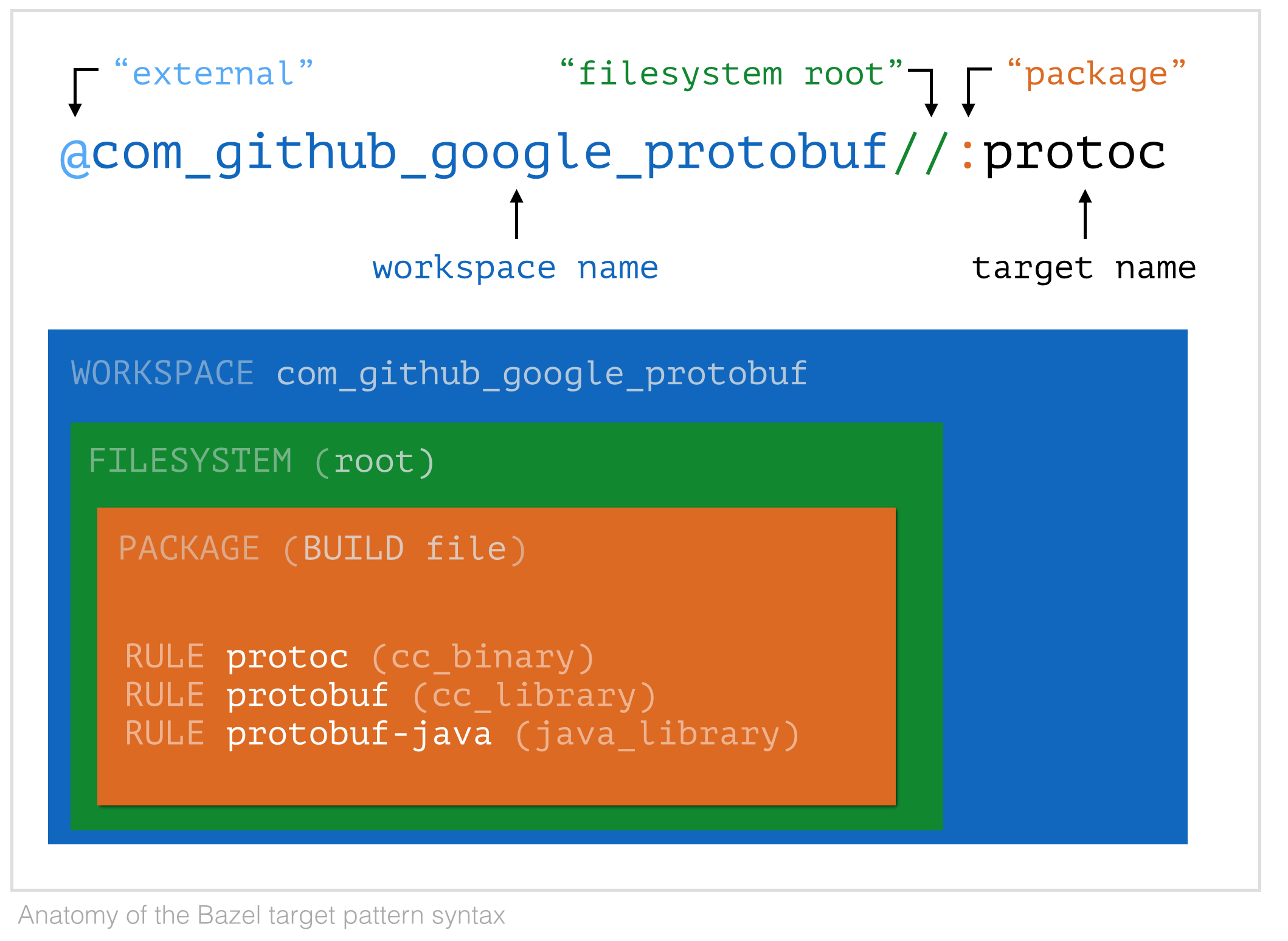
...

As we’ll see in a moment, *we name the protobuf external dependency with a specific commit ID so there’s no ambiguity about which protoc version we’re using*. In this way you can vendor in tools with your project with reliable, repeable, secure precision without bloating your repository by checking in binaries, resorting to git submodules, or similar hacks. Very clean!

Note: the gRPC repository also has a BUILD file: $ bazel query @com\_github\_grpc\_grpc//... --output label\_kind

**1.3: Target Pattern Syntax**

With those examples under our belt, let’s examine the target syntax a bit more. When I first started working with bazel I found the target-pattern syntax somewhat intimidating. It’s actually not too bad. Here’s a closer look:



* The @ (at-sign) selects an external workspace. These are established by [workspace rules](https://bazel.io/docs/be/workspace.html#workspace-rules) that bind a name to something fetched over the network (or your filesystem).
* The // (double-slash) selects the workspace root.
* The : (colon) selects a target (rule or file) within a *package*. Recall that a package is established by the presence of a BUILD file in a subfolder of the workspace.
* The / (single-slash) selects a folder within a workspace or package.

A common source of confusion is that the mere presence of a BUILD file defines that filesystem subtree as a package and therefore one must always account for that. For example, if there exists a file qux.js in foo/bar/baz/ and there exists a BUILD file in baz/ also, the file is selected with foo/bar/baz:qux.js and not foo/bar/baz/quz.js

*Common shortcut*: if there exists a rule having the same name as the package, this is the implied target and can be omitted. For example, there is a :jar target in the //jar package in the external workspace com\_google\_guava\_guava, so the following are eqivalent:

deps = ["@com\_google\_guava\_guava//jar:jar"]

deps = ["@com\_google\_guava\_guava//jar"]

**1.4: External Dependencies: Workspace Rules**

Many large organizations check-in in all the required tools, compilers, linkers, etc to guarantee correct, repeatable builds. With external workspaces, one can effectively accomplish the same thing without bloating your repository.

Note: the bazel convention is to use a fully-namespaced identifier for external dependency names (replacing special chars with underscore). For example, the remote repository URL is <https://github.com/google/protobuf.git>. This is simplified to the workspace identifier com\_github\_google\_protobuf. Similarly, by convention the jar artifact io.grpc:grpc-netty:jar:1.0.0-pre1 becomes io\_grpc\_grpc\_netty.

**1.4.1: Workspace Rules that require a pre-existing WORKSPACE**

These rules assume that the remote resource or URL contains a WORKSPACE file at the top of the file tree and BUILD files that define rule targets. These are referred to as *bazel repositories*.

* [git\_repository](https://bazel.io/docs/be/workspace.html#git_repository): external bazel dependency from a git repository. The rule requires commit (or tag).
* [http\_archive](https://bazel.io/docs/be/workspace.html#http_archive): an external zip or tar.gz dependency from a URL. It is highly recommended to name a sha265 for security.

Note: although you don’t interact directly with the bazel execution\_root, you can peek at what these external dependencies look like when unpacked at $(bazel info execution\_root)/external/WORKSPACE\_NAME.

**1.4.2: Workspace Rules that autogenerate a WORKSPACE file for you**

The implementation of these repository rules contain logic to autogenerate a WORKSPACE file and BUILD file(s) to make resources available. As always, it is recommended to provide a known sha265 for security to prevent a malicious agent from slipping in tainted code via a compromised network.

* [http\_jar](https://bazel.io/docs/be/workspace.html#http_jar): external jar from a URL. The jar file is available as a java\_library dependency as @WORKSPACE\_NAME//jar.
* [maven\_jar](https://bazel.io/docs/be/workspace.html#maven_jar): external jar from a URL. The jar file is available as a java\_library dependency as @WORKSPACE\_NAME//jar.
* [http\_file](https://bazel.io/docs/be/workspace.html#http_file): external file from a URL. The resource is available as a filegroup via @WORKSPACE\_NAME//file.

For example, we can peek at the generated BUILD file for the maven\_jar guava dependency via:

~/rules\_protobuf$ cat $(bazel info execution\_root)/external/com\_google\_guava\_guava/jar/BUILD

# DO NOT EDIT: automatically generated BUILD file for maven\_jar rule com\_google\_guava\_guava

java\_import(

name = 'jar',

jars = ['guava-19.0.jar'],

visibility = ['//visibility:public']

)

filegroup(

name = 'file',

srcs = ['guava-19.0.jar'],

visibility = ['//visibility:public']

)

Note: the external workspace directory won’t exist until you actually need it, so you’ll have to have built a target that requires it, such as bazel build examples/helloworld/java/org/pubref/rules\_protobuf/examples/helloworld/client

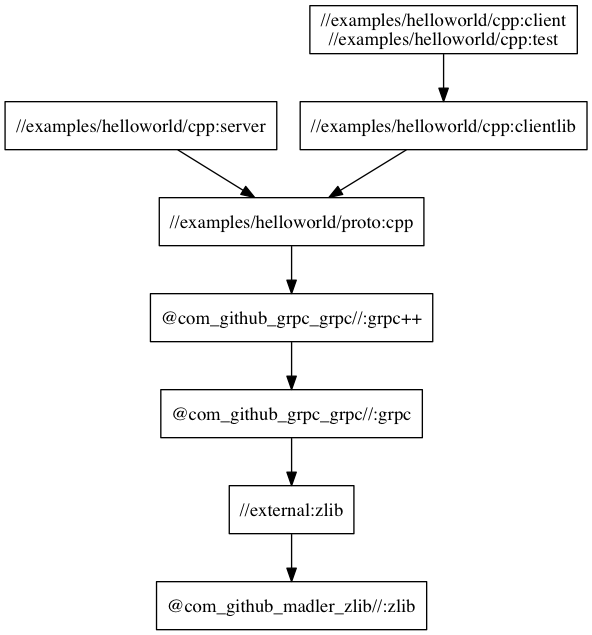
**1.4.3: Workspace Rules that accept a BUILD file as an argument**

If a repository has no BUILD file(s), you can put one into its filesystem root to adapt the external resource into bazel’s worldview and make those resources available to your project.

For example, consider [Mark Adler’s zlib library](https://github.com/madler/zlib). To start, let’s learn what depends on this code. This query says “*Ok bazel, for all targets in examples, find all dependencies (a transitive closure set), then tell me which ones depend on the zlib target in the root package of the external workspace com\_github\_madler\_zlib.*” Bazel reports this reverse dependency set. We request the output in graphviz format and pipe this to dot to generate the figure:

~/rules\_protobuf$ bazel query "rdeps(deps(//examples/...), @com\_github\_madler\_zlib//:zlib)" \

--output graph | dot -Tpng -O



So we can see that all grpc-related C code ultimately depends on this library. But, there is no BUILD file in Mark’s repo… where did it come from?

By using the variant workspace rule new\_git\_repository, we can provide our [own BUILD file](https://github.com/pubref/rules_protobuf/blob/master/protobuf/build_file/com_github_madler_zlib.BUILD) (which defines the cc\_library target) as follows:

new\_git\_repository(

name = "com\_github\_madler\_zlib",

remote = "https://github.com/madler/zlib",

tag: "v1.2.8",

build\_file: "//bzl:build\_file/com\_github\_madler\_zlib.BUILD",

)

This new\_\* family of workspace rules keeps your repository lean and allows you to vendor in pretty much any type of network-available resource. Awesome!

* [new\_git\_repository](https://bazel.io/docs/be/workspace.html#new_git_repository)
* [new\_local\_repository](https://bazel.io/docs/be/workspace.html#new_local_repository)
* [new\_http\_archive](https://bazel.io/docs/be/workspace.html#new_http_archive)

You can also [write your own repository rules](https://bazel.io/docs/skylark/repository_rules.html) that have custom logic to pull resources from the net and bind it into bazel’s view of the universe.

**1.5: Bazel Summary**

When presented with a command and a target-pattern, bazel goes through the following three phases:

1. Loading: Read the WORKSPACE and required BUILD files. Generate a dependency graph.
2. Analysis: for all nodes in the graph, which nodes are actually required for this build? Do we have all the necessary resources available?
3. Execution: execute each required node in the dependency graph and generate outputs.

Hopefully you now have enough conceptual knowledge of bazel to be productive.

**1.6: rules\_protobuf**

[rules\_protobuf](https://github.com/pubref/rules_protobuf) is an extension to bazel that takes care of:

1. Building the protocol buffer compiler protoc,
2. Downloading and/or building all the necessary protoc-gen plugins.
3. Downloading and/or building all the necessary gRPC-related support libraries.
4. Invoking protoc for you (on demand), smoothing out the idiosyncracies of different protoc plugins.

It works by passing one or more proto\_language specifications to the proto\_compile rule. A proto\_language rule contains the metadata about how to invoke the plugin and the predicted file outputs, while the proto\_compile rule interprets a proto\_language spec and builds the appropriate command-line arguments to protoc. For example, here’s how we can generate outputs for multiple languages simultaneously:

proto\_compile(

name = "pluriproto",

protos = [":protos"],

langs = [

"//cpp",

"//csharp",

"//closure",

"//ruby",

"//java",

"//java:nano",

"//python",

"//objc",

"//node",

],

verbose = 1,

with\_grpc = True,

)

$ bazel build :pluriproto

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

cd $(bazel info execution\_root) && bazel-out/host/bin/external/com\_github\_google\_protobuf/protoc \

--plugin=protoc-gen-grpc-java=bazel-out/host/genfiles/third\_party/protoc\_gen\_grpc\_java/protoc\_gen\_grpc\_java \

--plugin=protoc-gen-grpc=bazel-out/host/bin/external/com\_github\_grpc\_grpc/grpc\_cpp\_plugin \

--plugin=protoc-gen-grpc-nano=bazel-out/host/genfiles/third\_party/protoc\_gen\_grpc\_java/protoc\_gen\_grpc\_java \

--plugin=protoc-gen-grpc-csharp=bazel-out/host/genfiles/external/nuget\_grpc\_tools/protoc-gen-grpc-csharp \

--plugin=protoc-gen-go=bazel-out/host/bin/external/com\_github\_golang\_protobuf/protoc\_gen\_go \

--descriptor\_set\_out=bazel-genfiles/examples/proto/pluriproto.descriptor\_set \

--ruby\_out=bazel-genfiles \

--python\_out=bazel-genfiles \

--cpp\_out=bazel-genfiles \

--grpc\_out=bazel-genfiles \

--objc\_out=bazel-genfiles \

--csharp\_out=bazel-genfiles/examples/proto \

--java\_out=bazel-genfiles/examples/proto/pluriproto\_java.jar \

--javanano\_out=ignore\_services=true:bazel-genfiles/examples/proto/pluriproto\_nano.jar \

--js\_out=import\_style=closure,error\_on\_name\_conflict,binary,library=examples/proto/pluriproto:bazel-genfiles \

--js\_out=import\_style=commonjs,error\_on\_name\_conflict,binary:bazel-genfiles \

--go\_out=plugins=grpc,Mexamples/proto/common.proto=github.com/pubref/rules\_protobuf/examples/proto/pluriproto:bazel-genfiles \

--grpc-java\_out=bazel-genfiles/examples/proto/pluriproto\_java.jar \

--grpc-nano\_out=ignore\_services=true:bazel-genfiles/examples/proto/pluriproto\_nano.jar \

--grpc-csharp\_out=bazel-genfiles/examples/proto \

--proto\_path=. \

examples/proto/common.proto

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

examples/proto/common\_pb.rb

examples/proto/pluriproto\_java.jar

examples/proto/pluriproto\_nano.jar

examples/proto/common\_pb2.py

examples/proto/common.pb.h

examples/proto/common.pb.cc

examples/proto/common.grpc.pb.h

examples/proto/common.grpc.pb.cc

examples/proto/Common.pbobjc.h

examples/proto/Common.pbobjc.m

examples/proto/pluriproto.js

examples/proto/Common.cs

examples/proto/CommonGrpc.cs

examples/proto/common.pb.go

examples/proto/common\_pb.js

examples/proto/pluriproto.descriptor\_set

The various \*\_proto\_library rules (that we’ll be using below) internally invoke this proto\_compile rule, then consume the generated outputs and compile them with the requisite libraries into .class, .so, .a (or whatever) objects.

So let’s *make something* already! We’ll use bazel and rules\_protobuf to build a gRPC application.

**2: Building a gRPC service with rules\_protobuf**

The application will involve communication between two different gRPC services:

**2.1: Services**

1. **The Greeter service**: This is the familiar “Hello World” starter example that accepts a request with a user argument and replies back with the string Hello {user}.
2. **The GreeterTimer service**: This gRPC service will repeatedly call a Greeter service in batches and report back aggregate batch times (in milliseconds). In this way we can compare some average rpc times for the different Greeter service implementations.

This is an informal benchmark intended only for demonstration of building gRPC applications. For more formal performance testing, consult the [gRPC performance dashboard](https://performance-dot-grpc-testing.appspot.com/explore?dashboard=5760820306771968).

**2.2: Compiled Programs**

For the demo, we’ll use 6 different compiled programs written in 4 languages:

* A GreeterTimer client (go). This command-line interface requires the greetertimer.proto service definition defined locally in the //proto:greetertimer.proto file.
* A GreeterTimer server (java). This netty-based server requires both the //proto/greetertimer.proto file and the proto definition defined externally in @org\_pubref\_rules\_protobuf//examples/helloworld/proto:helloworld.proto.
* Four Greeter server implementations (C++, java, go, and C#). rules\_protobuf already provides these example implementations, so we’ll just use them directly.

**2.3: Protobuf Definitions**

GreeterTimer accepts a unary TimerRequest and streams back a sequence of BatchReponse until all messages have been processed, at which point the remote procedure call is complete.

service GreeterTimer {

// Unary request followed by multiple streamed responses.

// Response granularity will be set by the request batch size.

rpc timeHello(TimerRequest) returns (stream BatchResponse);

}

TimerRequest includes metadata about where to contact the Greeter service, how many total RPC calls to make, and how frequent to stream back a BatchResponse (configured via the batch size).

message TimerRequest {

// the host where the grpc server is running

string host = 1;

// The port of the grpc server

int32 port = 2;

// The total number of hellos

int32 total = 3;

// The number of hellos before sending a BatchResponse.

int32 batchSize = 4;

}

BatchResponse reports the number of calls made in the batch, how long the batch run took, and the number of remaining calls.

message BatchResponse {

// The number of checks that are remaining, calculated relative to

// totalChecks in the request.

int32 remaining = 1;

// The number of checks actually performed in this batch.

int32 batchCount = 2;

// The number of checks that failed.

int32 errCount = 3;

// The total time spent, expressed as a number of milliseconds per

// request batch size (total time spent performing batchSize number

// of health checks).

int64 batchTimeMillis = 4;

}

The non-streaming Greeter service takes a unary HelloRequest and responds with a single HelloReply:

service Greeter {

rpc SayHello (HelloRequest) returns (HelloReply) {}

}

message HelloRequest {

string name = 1;

common.Config config = 2;

}

message HelloReply {

string message = 1;

}

The common.Config message type is not particularly functional here but serves to demonstrate the use of imports. rules\_protobuf can help with more complex setups having multiple proto → proto dependencies.

**2.4: Build the grpc\_greetertimer example application.**

This demo application can be cloned at <https://github.com/pubref/grpc_greetertimer>.

**2.4.1: Create the Project Layout**

Here’s the directory layout and relevant BUILD files we’ll be using:

~$ mkdir grpc\_greetertimer && cd grpc\_greetertimer

~/grpc\_greetertimer$ mkdir -p proto/ go/ java/org/pubref/grpc/greetertimer/

~/grpc\_greetertimer$ touch WORKSPACE

~/grpc\_greetertimer$ touch proto/BUILD

~/grpc\_greetertimer$ touch proto/greetertimer.proto

~/grpc\_greetertimer$ touch go/BUILD

~/grpc\_greetertimer$ touch go/main.go

~/grpc\_greetertimer$ touch java/org/pubref/grpc/greetertimer/BUILD

~/grpc\_greetertimer$ touch java/org/pubref/grpc/greetertimer/GreeterTimerServer.java

**2.4.2: The WORKSPACE**

We’ll begin by creating the [WORKSPACE](https://github.com/pubref/grpc_greetertimer) file with a reference to the rules\_protobuf repository. We load the main entrypoint skylark file [rules.bzl](https://github.com/pubref/rules_protobuf/blob/master/protobuf/rules.bzl) in the //bzl package and call its protobuf\_repositories function with the languages to we want to use (in this case java and go). We also load [rules\_go](https://github.com/bazelbuild/rules_go) for go compile support (not shown).

# File //:WORKSPACE

workspace(name = "org\_pubref\_grpc\_greetertimer")

git\_repository(

name = "org\_pubref\_rules\_protobuf",

remote = "https://github.com/pubref/rules\_protobuf.git",

tag = "v0.6.0",

)

# Load language-specific dependencies

load("@org\_pubref\_rules\_protobuf//java:rules.bzl", "java\_proto\_repositories")

java\_proto\_repositories()

load("@org\_pubref\_rules\_protobuf//go:rules.bzl", "go\_proto\_repositories")

go\_proto\_repositories()

Refer to the [repositories.bzl file](https://github.com/pubref/rules_protobuf/protobuf/internal/repositories.bzl), if you are interested in inspecting the dependencies.

Bazel won’t actually *fetch* something unless we actually need it by some other rule later, so let’s go ahead and write some code. We’ll store our protocol buffer sources in //proto, our java sources in //java, and go source in //go.

Note: go development within a bazel workspace is a little different than vanilla go. In particular, one does not have to adhere to a typical GOCODE layout having a src/, pkg/, bin/ subdirectories.

**2.4.3: The GreeterTimer Server**

The [java server’s](https://grpc.io/blog/bazel_rules_protobuf/java/org/pubref/grpc/greetertimer/GreeterTimerServer.java) main job is to accept requests and then connect to the requested Greeter service as a client. The implementation counts down the number of remaining messages and does a blocking sayHello(request) for each one. If the batchSize limit is met, the observer.onNext(response) message is invoked, streaming back a response to the client.

/\* File //java/org/pubref/grpc/greetertimer:GreeterTimerServer.java \*/

while (remaining-- > 0) {

if (batchCount++ == batchSize) {

BatchResponse response = BatchResponse.newBuilder()

.setRemaining(remaining)

.setBatchCount(batchCount)

.setBatchTimeMillis(batchTime)

.setErrCount(errCount)

.build();

observer.onNext(response);

}

blockingStub.sayHello(HelloRequest.newBuilder()

.setName("#" + remaining)

.build());

}

}

**2.4.4: The GreeterTimer Client**

The [go client](https://grpc.io/blog/bazel_rules_protobuf/go/main.go) prepares a TimerRequest and gets back a stream interface from the client.TimeHello method. We call its Recv() method until EOF, at which point the call is complete. A summary of each BatchResponse is simply printed out to the terminal.

// File: //go:main.go

func submit(client greeterTimer.GreeterTimerClient, request \*greeterTimer.TimerRequest) error {

stream, err := client.TimeHello(context.Background(), request)

if err != nil {

log.Fatalf("could not submit request: %v", err)

}

for {

batchResponse, err := stream.Recv()

if err == io.EOF {

return nil

}

if err != nil {

log.Fatalf("error during batch recv: %v", err)

return err

}

reportBatchResult(batchResponse)

}

}

**2.4.5: Generate the go protobuf+gRPC code**

In our //proto:BUILD file, we have a go\_proto\_library rule loaded from the rules\_protobuf repository. Internally, the rule declares to bazel that it is responsible for creating greetertimer.pb.go output file. This rule won’t actually *do* anything unless we depend on it somewhere else.

# File: //proto:BUILD

load("@org\_pubref\_rules\_protobuf//go:rules.bzl", "go\_proto\_library")

go\_proto\_library(

name = "go\_default\_library",

protos = [

"greetertimer.proto",

],

with\_grpc = True,

)

The go client implementation depends on the go\_proto\_library as source file provider to the go\_binary rule. We also pass in some compile-time dependencies named in the GRPC\_COMPILE\_DEPS list.

load("@io\_bazel\_rules\_go//go:def.bzl", "go\_binary")

load("@org\_pubref\_rules\_protobuf//go:rules.bzl", "GRPC\_COMPILE\_DEPS")

go\_binary(

name = "hello\_client",

srcs = [

"main.go",

],

deps = [

"//proto:go\_default\_library",

] + GRPC\_COMPILE\_DEPS,

)

~/grpc\_greetertimer$ bazel build //go:client

Here’s what happens when we invoke bazel to actually build the client binary:

1. Bazel checks to see if the inputs (files) that the binary depends on have changed (by content hash and filestamps). Bazel recognizes that the output files for the //proto:go\_default\_library have not been built.
2. Bazel checks to see if all the necessary inputs (including tools) for the go\_proto\_library are available. If not, download and build all the necessary tools. Then, invoke the rule.
   1. Fetch the google/protobuf repository and build protoc from source (via a cc\_binary rule).
   2. Build the protoc-gen-go plugin from source (via a go\_binary rule).
   3. Invoke protoc with the protoc-gen-go plugin with the appropriate options and arguments.
   4. Confirm that all the declared outputs of the go\_proto\_library where actually built (should be in bazel-bin/proto/greetertimer.pb.go).
3. Compile the generated greetertimer.pb.go with the client main.go file, creating the bazel-bin/go/client executable.

**2.4.6: Generate the java protobuf libraries**

The java\_proto\_library rule is functionally identical to the go\_proto\_library rule. However, instead of providing a \*.pb.go file, it bundles up all the generated outputs into a \*.srcjar file (which is then used as an input to the java\_library rule). This an implementation detail of the java rule. Here is how we build the final java binary:

java\_binary(

name = "server",

main\_class = "org.pubref.grpc.greetertimer.GreeterTimerServer",

srcs = [

"GreeterTimerServer.java",

],

deps = [

":timer\_protos",

"@org\_pubref\_rules\_protobuf//examples/helloworld/proto:java",

"@org\_pubref\_rules\_protobuf//java:grpc\_compiletime\_deps",

],

runtime\_deps = [

"@org\_pubref\_rules\_protobuf//java:netty\_runtime\_deps",

],

)

1. The :timer\_protos is a locally defined java\_proto\_library rule.
2. The @org\_pubref\_rules\_protobuf//examples/helloworld/proto:java is an external java\_proto\_library rule that generates the greeter service client stub in our own workspace.
3. Finally, we name the compile-time and run-time dependencies for the executable jar. If these jar files have not yet been downloaded from maven central, they will be fetch as soon as we need them:

~/grpc\_greetertimer$ bazel build java/org/pubref/grpc/greetertimer:server

~/grpc\_greetertimer$ bazel build java/org/pubref/grpc/greetertimer:server\_deploy.jar

This last form (having the extra \_deploy.jar) is called an *implicit target* of the :server rule. When invoked this way, bazel will pack up all the required classes and generate a standalone executable jar that can be run independently in a jvm.

**2.4.7: Run it!**

First, we’ll start a greeter server (one at a time):

~/grpc\_greetertimer$ cd ~/rules\_protobuf

~/rules\_protobuf$ bazel run examples/helloworld/go/server

~/rules\_protobuf$ bazel run examples/helloworld/cpp/server

~/rules\_protobuf$ bazel run examples/helloworld/java/org/pubref/rules\_protobuf/examples/helloworld/server:netty

~/rules\_protobuf$ bazel run examples/helloworld/csharp/GreeterServer

INFO: Server started, listening on 50051

In a separate terminal, start the greetertimer server:

~/grpc\_greetertimer$ bazel build //java/org/pubref/grpc/greetertimer:server\_deploy.jar

~/grpc\_greetertimer$ java -jar bazel-bin/java/org/pubref/grpc/greetertimer/server\_deploy.jar

Finally, in a third terminal, invoke the greetertimer client:

# Timings for the java server

~/rules\_protobuf$ bazel run examples/helloworld/java/org/pubref/rules\_protobuf/examples/helloworld/server:netty

~/grpc\_greeterclient$ bazel run //go:client -- -total\_size 10000 -batch\_size 1000

17:31:04 1001 hellos (0 errs, 8999 remaining): 1.7 hellos/ms or ~590µs per hello

# ... plus a few runs to warm up the jvm...

17:31:13 1001 hellos (0 errs, 8999 remaining): 6.7 hellos/ms or ~149µs per hello

17:31:13 1001 hellos (0 errs, 7998 remaining): 9.0 hellos/ms or ~111µs per hello

17:31:13 1001 hellos (0 errs, 6997 remaining): 8.9 hellos/ms or ~112µs per hello

17:31:13 1001 hellos (0 errs, 5996 remaining): 9.2 hellos/ms or ~109µs per hello

17:31:13 1001 hellos (0 errs, 4995 remaining): 9.4 hellos/ms or ~106µs per hello

17:31:13 1001 hellos (0 errs, 3994 remaining): 9.0 hellos/ms or ~111µs per hello

17:31:13 1001 hellos (0 errs, 2993 remaining): 9.4 hellos/ms or ~107µs per hello

17:31:13 1001 hellos (0 errs, 1992 remaining): 9.4 hellos/ms or ~107µs per hello

17:31:13 1001 hellos (0 errs, 991 remaining): 9.1 hellos/ms or ~110µs per hello

17:31:14 991 hellos (0 errs, -1 remaining): 9.0 hellos/ms or ~111µs per hello```

```sh

# Timings for the go server

~/rules\_protobuf$ bazel run examples/helloworld/go/server

~/grpc\_greeterclient$ bazel run //go:client -- -total\_size 10000 -batch\_size 1000

17:32:33 1001 hellos (0 errs, 8999 remaining): 7.5 hellos/ms or ~134µs per hello

17:32:33 1001 hellos (0 errs, 7998 remaining): 7.9 hellos/ms or ~127µs per hello

17:32:34 1001 hellos (0 errs, 6997 remaining): 7.8 hellos/ms or ~128µs per hello

17:32:34 1001 hellos (0 errs, 5996 remaining): 7.7 hellos/ms or ~130µs per hello

17:32:34 1001 hellos (0 errs, 4995 remaining): 7.9 hellos/ms or ~126µs per hello

17:32:34 1001 hellos (0 errs, 3994 remaining): 8.0 hellos/ms or ~125µs per hello

17:32:34 1001 hellos (0 errs, 2993 remaining): 7.6 hellos/ms or ~132µs per hello

17:32:34 1001 hellos (0 errs, 1992 remaining): 7.9 hellos/ms or ~126µs per hello

17:32:34 1001 hellos (0 errs, 991 remaining): 7.9 hellos/ms or ~127µs per hello

17:32:34 991 hellos (0 errs, -1 remaining): 7.8 hellos/ms or ~128µs per hello

# Timings for the C++ server

~/rules\_protobuf$ bazel run examples/helloworld/cpp:server

~/grpc\_greeterclient$ bazel run //go:client -- -total\_size 10000 -batch\_size 1000

17:33:10 1001 hellos (0 errs, 8999 remaining): 9.1 hellos/ms or ~110µs per hello

17:33:10 1001 hellos (0 errs, 7998 remaining): 9.0 hellos/ms or ~111µs per hello

17:33:10 1001 hellos (0 errs, 6997 remaining): 9.1 hellos/ms or ~110µs per hello

17:33:10 1001 hellos (0 errs, 5996 remaining): 8.6 hellos/ms or ~116µs per hello

17:33:10 1001 hellos (0 errs, 4995 remaining): 9.0 hellos/ms or ~111µs per hello

17:33:10 1001 hellos (0 errs, 3994 remaining): 9.0 hellos/ms or ~111µs per hello

17:33:10 1001 hellos (0 errs, 2993 remaining): 9.1 hellos/ms or ~110µs per hello

17:33:10 1001 hellos (0 errs, 1992 remaining): 9.0 hellos/ms or ~111µs per hello

17:33:10 1001 hellos (0 errs, 991 remaining): 9.0 hellos/ms or ~111µs per hello

17:33:11 991 hellos (0 errs, -1 remaining): 9.0 hellos/ms or ~111µs per hello

# Timings for the C# server

~/rules\_protobuf$ bazel run examples/helloworld/csharp/GreeterServer

~/grpc\_greeterclient$ bazel run //go:client -- -total\_size 10000 -batch\_size 1000

17:34:37 1001 hellos (0 errs, 8999 remaining): 6.0 hellos/ms or ~166µs per hello

17:34:37 1001 hellos (0 errs, 7998 remaining): 6.7 hellos/ms or ~150µs per hello

17:34:37 1001 hellos (0 errs, 6997 remaining): 6.8 hellos/ms or ~148µs per hello

17:34:37 1001 hellos (0 errs, 5996 remaining): 6.8 hellos/ms or ~147µs per hello

17:34:37 1001 hellos (0 errs, 4995 remaining): 6.7 hellos/ms or ~150µs per hello

17:34:38 1001 hellos (0 errs, 3994 remaining): 6.7 hellos/ms or ~150µs per hello

17:34:38 1001 hellos (0 errs, 2993 remaining): 6.7 hellos/ms or ~149µs per hello

17:34:38 1001 hellos (0 errs, 1992 remaining): 6.7 hellos/ms or ~149µs per hello

17:34:38 1001 hellos (0 errs, 991 remaining): 6.8 hellos/ms or ~148µs per hello

17:34:38 991 hellos (0 errs, -1 remaining): 6.8 hellos/ms or ~147µs per hello

The informal analysis demonstrated comparable timings for c++, go, and java greeter service implementations. The c++ server had the overall fastest and most consistent performance. The go implementation was also very consistent, but slightly slower than C++. Java demonstrated some initial relative slowness likely due to the JVM warming up but soon converged on timings similar to the C++ implementation. C# has consistent performance but marginally slower.

**2.5: Summary**

Bazel assists in the construction of gRPC applications by providing a capable build environment for services built in a multitude of languages. [rules\_protobuf](https://github.com/pubref/rules_protobuf/) complements bazel by packaging up all the dependencies needed and abstracting away the need to call protoc directly.

In this workflow one does not need to check in the generated source code (it is always generated on-demand within your workspace). For projects that *do* require this, one can use the output\_to\_workspace option to place the generated files alongside the protobuf definitions.

Finally, rules\_protobuf has full support for the [grpc-gateway](https://github.com/grpc-ecosystem/grpc-gateway) project via the [grpc\_gateway\_proto\_library](https://github.com/pubref/rules_protobuf/tree/master/grpc_gateway#grpc_gateway_proto_library) and [grpc\_gateway\_binary](https://github.com/pubref/rules_protobuf/tree/master/grpc_gateway#grpc_gateway_binary) rules, so you can easily bridge your gRPC apps with HTTP/1.1 gateways.

Refer to the [complete list of supported languages and gRPC versions](https://github.com/pubref/rules_protobuf/#rules) for more information.

And… that’s a wrap. Happy procedure calling!

Paul Johnston is the principal at [PubRef](https://pubref.org) ([@pub\_ref](https://twitter.com/pub_ref)), a solutions provider for scientific communications workflows. If you have an organizational need for assistance with Bazel, gRPC, or related technologies, please contact pcj@pubref.org. Thanks!

https://grpc.io/blog/bazel\_rules\_protobuf/

https://github.com/pubref/grpc\_greetertimer