Extracted from:

Distributed Services with Go

Your Guide to Reliable, Scalable, and Maintainable Systems

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Distributed Services with Go

Your Guide to Reliable, Scalable, and Maintainable Systems

Travis Jeffery

edited by Dawn Schanafelt and Katharine Dvorak

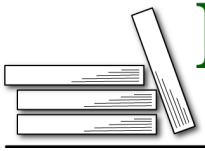
Distributed Services with Go

Your Guide to Reliable, Scalable, and Maintainable Systems

Travis Jeffery

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Dawn Schanafelt and Katharine Dvorak
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-760-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2021

CHAPTER 2

Structure Data with Protocol Buffers

When building distributed services, you're communicating between the services over a network. To send data (such as your structs) over a network, you need to encode the data in a format to transmit, and lots of programmers choose JSON. When you're building public APIs or you're creating a project where you don't control the clients, JSON makes sense because it's accessible—both for humans to read and computers to parse. But when you're building private APIs or building projects where you *do* control the clients, you can make use of a mechanism for structuring and transmitting data that—compared to JSON—makes you more productive and helps you create services that are faster, have more features, and have fewer bugs.

So what is this mechanism? Protocol buffers (also known as *protobuf*), which is Google's language and platform-neutral extensible mechanism for structuring and serializing data. The advantages of using protobuf are that it:

- Guarantees type-safety;
- Prevents schema-violations;
- Enables fast serialization; and
- Offers backward compatibility.

Protobuf lets you define how you want your data structured, compile your protobuf into code in potentially many languages, and then read and write your structured data to and from different data streams. Protocol buffers are good for communicating between two systems (such as microservices), which is why Google used protobuf when building gRPC to develop a high-performance remote procedure call (RPC) framework.

If you haven't worked with protobuf before, you may have some of the same concerns I had—that protobuf seems like a lot of extra work. I promise you that, after working with it in this chapter and the rest of the book, you'll see

that it's really not so bad. It offers many advantages over JSON, and it'll end up saving you a lot of work.

Here's a quick example that shows what protocol buffers look like and how they work. Imagine you work at Twitter and one of the object types you work with are Tweets. Tweets, at the very least, comprise the author's message. If you defined this in protobuf, it would look like this:

StructureDataWithProtobuf/example.proto

```
syntax = "proto3";

package twitter;

message Tweet {
  string message = 1;
}
```

You'd then compile this protobuf into code in the language of your choice. For example, the protobuf compiler would take this protobuf and generate the following Go code:

StructureDataWithProtobuf/example.pb.go

```
// Code generated by protoc-gen-go. DO NOT EDIT.
// source: example.proto

package twitter

type Tweet struct {
        Message string `protobuf:"bytes,1,opt,name=message,proto3"
json:"message,omitempty"`
        // Note: Protobuf generates internal fields and methods
        // I haven't included for brevity.
}
```

But why not just write that Go code yourself? Why use protobuf instead? I'm glad you asked.

Why Use Protocol Buffers?

Protobuf offers all kinds of useful features:

Consistent schemas

With protobuf, you encode your semantics once and use them across your services to ensure a consistent data model throughout your whole system. My colleagues and I built the infrastructures at my last two companies on microservices, and we had a repo called "structs" that housed our protobuf and their compiled code, which all our services depended on. By doing this, we ensured that we didn't send multiple, inconsistent schemas to prod. Thanks to Go's type checking, we could update our structs dependency, run the tests that touched our data models, and the

compiler and tests would tell us whether our code was consistent with our schema.

Versioning for free

One of Google's motivations for creating protobuf was to eliminate the need for version checks and prevent ugly code like this:

StructureDataWithProtobuf/example.go

```
if (version == 3) {
...
} else if (version > 4) {
    if (version == 5) {
        ...
    }
    ...
}
```

Think of a protobuf message like a Go struct because when you compile a message it turns into a struct. With protobuf, you number your fields on your messages to ensure you maintain backward compatibility as you roll out new features and changes to your protobuf. So it's easy to add new fields, and intermediate servers that need not use the data can simply parse it and pass through it without needing to know about all the fields. Likewise with removing fields: you can ensure that deprecated fields are no longer used by marking them as reserved; the compiler will then complain if anyone tries to use to the deprecated fields.

Less boilerplate

The protobuf libraries handle encoding and decoding for you, which means you don't have to handwrite that code yourself.

Extensibility

The protobuf compiler supports extensions that can compile your protobuf into code using your own compilation logic. For example, you might want several structs to have a common method. With protobuf, you can write a plugin to generate that method automatically.

Language agnosticism

Protobuf is implemented in many languages: since Protobuf version 3.0, there's support for Go, C++, Java, JavaScript, Python, Ruby, C#, Objective C, and PHP, and third-party support for other languages. And you don't have to do any extra work to communicate between services written in different languages. This is great for companies with various teams that want to use different languages, or when your team wants to migrate to another language.

Performance

Protobuf is highly performant, and has smaller payloads and serializes up to six times faster than JSON.[1]

gRPC uses protocol buffers to define APIs and serialize messages; we'll use gRPC to build our client and server.

Hopefully I've done a decent job of convincing you that protobuf is cool. But the theory alone is boring! Let's get you set up to create your own protobuf and use it to build stuff.

Install the Protocol Buffer Compiler

The first thing we need to do to get you compiling protobuf is—you guessed it—install the compiler. Go to the Protobuf release page on GitHub[2] and download the relevant release for your computer. If you're on a Mac, for instance, you'd download protoc-3.9.0-osx-x86_64.zip. You can download and install in your terminal like so:

```
$ wget https://github.com/protocolbuffers/protobuf/\
releases/download/v3.9.0/protoc-3.9.0-osx-x86_64.zip
$ unzip protoc-3.9.0-osx-x86_64.zip -d /usr/local/protobuf
```

Here's what the layout and files in the extracted protobuf directory look like:

```
› tree /usr/local/protobuf
/usr/local/protobuf
├── bin
│   └── protoc
├── include
│   └── google
│       └── protobuf
│           ├── any.proto
│           ├── api.proto
│           ├── compiler
│           │   └── plugin.proto
│           ├── descriptor.proto
│           ├── duration.proto
│           ├── empty.proto
│           ├── field_mask.proto
│           ├── source_context.proto
│           ├── struct.proto
│           ├── timestamp.proto
│           ├── type.proto
│           └── wrappers.proto
└── readme.txt
```

1. https://auth0.com/blog/beating-json-performance-with-protobuf
2. https://github.com/protocolbuffers/protobuf/releases

As you can see, a protobuf installation consists of two directories. The bin directory contains the compiler binary named protoc, and the include directories contains a bunch of protobuf files that are like protobuf's standard library. A mistake I've seen many people make when setting up their systems to work with protobuf is that they install the compiler binary without the include protobuf files. But without those files you can't compile successfully, so just extract the whole release using the commands I just showed you and you'll be just dandy.

Now that you've got the compiler binary installed, make sure your shell can find and run it. Add the binary to your PATH env var using your shell's configuration file. If you're using ZSH for instance, run something like the following to update your configuration:

```
$ echo 'export PATH="$PATH:/usr/local/protobuf/bin"' >> ~/.zshenv
```

At this point the protobuf compiler is installed on your machine. To test the installation, run protoc --version. If you don't see any errors, you're ready to handle the rest of this chapter. If you do see errors, don't worry: few installation problems are unique. Google will show you the way.

With the compiler installed, you're ready to write and compile some protobuf. Let's get to it!