Twirp: a sweet new RPC framework for Go

Jan 16 2018 - By Spencer Nelson

Today Twitch is releasing an RPC framework we use for communication between backend servers written in Go. It's called **Twirp**, and it's **available now** under an Apache 2 open source license.

Twirp has been tremendously successful at Twitch — it has grown in usage exponentially, roughly tripling every three months as more and more internal teams continue to adopt it due to its advantages over "REST" APIs or gRPC, its two closest competitors.

Structured RPCs are *much* easier to design and maintain than URL-oriented REST APIs, as they let you focus on business logic instead of routing schemes. Changing APIs to add new fields or methods is much easier and the peculiarities of serialization (like, say, JSON's

lack of 64-bit numbers) can be hidden from view.

gRPC implements structured RPCs but we found that its complexity and sprawl brought an unacceptable number of bugs — and we had trouble justifying its hard http/2 requirement. Twirp is a structured RPC framework, but with an emphasis on **simplicity**. It works on HTTP 1.1, chooses stability and modularity over an expansive feature set, and then gets out of the way.

Twirp is *so* simple, in fact, that you can make valid requests on the command line with cURL without much thought. It's as simple as a Content-Type header, a proper payload, and the right URL — all of which are very well-standardized. So, for example:

```
# This is a valid Twirp request:
curl \
 -header 'Content-Type:application/json' \
 -data '{"user": "spencer", "email": "spencer@twitch.tv"}' \
 [http://localhost:9090/twirp/twitch.example.EmailBoss
/UpdateEmail](http://localhost:9090/twirp/twitch.example.EmailBoss
/UpdateEmail)
```

We think you'll want to use Twirp if you're working in a service-oriented architecture and program mostly in Go. We also think you'll be able to drop it into your system one service at a time, even if you're using several non-Go languages. The core design of Twirp is language-agnostic and we're planning to expand into new languages, but our Go implementation is already stable and capable of serving heavy production loads.

Get started now by visiting **github.com/twitchtv/twirp**.

But let's back up a bit. What do we mean by an "RPC framework,"

and why would you want one? Read on for more technical details. We'll cover:

- Why you need structured RPCs
- An example using Twirp to build an API
- Low-level details of Twirp's protocol
- Why not gRPC?
- Twirp in production
- How you can get involved

We'll start by describing a typical way engineers write backend service APIs: hand-written, with a collection of URLs serving JSON responses.

Why you need structured RPCs

Service oriented architectures are nothing new, but many organizations (including Twitch) arrive at them when they outgrow an early monolithic codebase. In 2015, we made a big push to migrate away from a single giant app and pull logic out into separate services.

When you do this, you almost immediately have to start designing APIs for these new services. How are other components of Twitch going to contact your new subcomponent? There are a *lot* of decisions to make here — let's stroll through just some of them.

We'll imagine a very, very simple mini-service which just updates a user's email address. How are we going to accept requests? Any of these HTTP APIs seem plausible:

- `POST /users/:id/email`, body is the new email address to use

- `PUT /users/:id/email`, body is the new email address to use
- `POST (or PUT) /users/:username`, body is JSON-encoded, like `{“email”: }`
- `PATCH /users/:username/info,` body is some funky JSON structure describing edits (for the real REST purists out there)

And what should the endpoint return? And how should it communicate errors? And are there particular headers that should be set? What about versioning? The list of concerns can go on a long time.

Many services will have more than one “action” they support; these design questions need to be re-decided for *each method* in the API. Now multiply all that decision-making many times *again*, because you’re going to need to do it again for every new service.

And that’s just at the moment of service creation! You will continue to pay debt for a long time if you hand-write APIs in that way:

- You’ll have different URL schemes for different services, which makes logs harder to understand and metrics very difficult to organize clearly.
- Every client will need to be hand-written, and you’ll need hand-written documentation of every API — which will need to be kept up-to-date just to help others write those clients.
- When it comes time to update the service to support new products — which will be sooner than you think — you’ll need to make changes, which will be difficult to propagate out to clients, assuming you even have a complete list of your clients.

These problems get harder and harder as you have more services and more teams in a larger organization. If you walk down this road

for too long, you'll realize you traded a monolith filled with spaghetti code for a whole lot of small services, all strung together with spaghetti.

What we need here is some way of standardizing our URL schemes, our request and response payload shapes, and our error messages. We could try to standardize this purely by fiat, but that rarely scales well, if it gets off the ground at all. We're software engineers — we can solve this with code.

Code generation to the rescue

Twirp solves the backend communication problem through code generation. You write a small protobuf document describing your API. You pass it through the Twirp code generator, and it emits a Go file containing code for an HTTP router and a matching HTTP client that encapsulate all the grubby bits here — routing, serialization, HTTP methods, and so on.

To make this a little clearer, let's look at an example of one of these protobuf service specifications:

```
syntax = "proto3";
```

```
package twitch.users.email;
```

```
service EmailBoss {
  rpc UpdateEmail(UpdateEmailRequest) returns
(UpdateEmailResponse);
```

```
}
```

```
message UpdateEmailRequest {
  int64 user_id = 1;
  string new_email = 2;
}
```

```
message UpdateEmailResponse {
  // EmailBoss may clean up the email it receives to trim
whitespace
  // or remove superfluous characters. The cleaned_email will be
the
  // email actually stored after this cleaning.
  string cleaned_email = 1;
}
```

This defines a service called `EmailBoss`. It has one method (or "`rpc`"), `UpdateEmail`. We define the shapes of the request and response message types, including the types of their fields.

(Immediately, we see one advantage of defining a service like this: there's a clear place to put documentation on behavior. There's a big comment in the `UpdateEmailResponse` message type's `cleaned_email` field which explains its intent and its meaning.)

When Twirp generates Go code from this protobuf file, it will generate an interface that describes the EmailBoss service:

```
```

```
type EmailBoss interface {
  UpdateEmail(context.Context, *UpdateEmailRequest)
(*UpdateEmailResponse, error)
}
```

This should make good sense if you're used to Go: the interface maps cleanly onto the service's definition. It has one method, `UpdateEmail`. The `UpdateEmailRequest` and `UpdateEmailResponse` types are generated by the Go protobuf generator, and are used in that method.

Next, Twirp will generate a function that will expose any implementation of the `EmailBoss` interface as an HTTP Handler. It looks something like this, slightly edited to keep this blog post clear:

```
func NewEmailBossServer(svc EmailBoss) http.Handler
```

And it also provides a constructor to make a client that talks to a Twirp server providing the EmailBoss functionality:

```
func NewEmailBossProtobufClient(hostport string, client
*http.Client) EmailBoss
```

This sets you off to the races very quickly, both as a service owner or a user of a Twirp service. As the service owner, you can dive in immediately to making an implementation of the EmailBoss interface — in other words, you can focus just on the business logic. You don't need to do any thinking about URLs or HTTP methods or headers or whatever.

On the client side, all you need to know is the host and port of the service, and you're good to go — you have a client that you can directly use to call UpdateEmail.

Under the hood: Twirp's protocol

So how do our generated servers and clients work?

It's all plain HTTP 1.1. Every request is a POST, and the URLs are static strings identifying the right method in the right service. The body is either JSON- or binary-encoded protobuf, and so is the response. A header identifies the encoding of the body. And that's about it.

For example, for our EmailBoss example:

- The URL is `/twirp/twitch.users.email.EmailBoss/UpdateEmail`.
- The HTTP Method is `POST`, because it's *always* POST.
- The body of the message should be a protobuf `UpdateEmailRequest`.
- The body should be encoded either in **binary** or in **JSON**.
- The request should have a Content-Type header set to either `application/protobuf` or `application/json`, matching the encoding of the body.
- The response body will be a protobuf `UpdateEmailResponse`, encoded in the same way as the request.
- If there is an error, it will be JSON encoded, including a message and a standardized error code.

This is simple enough that the Twirp generator "inlines" it: a generated Twirp file has very few runtime dependencies, and contains all of its routing and serialization code directly.

Why not gRPC?

This code generation approach is not a novel idea at all. Google provides a framework, **gRPC**, which does a very similar thing, and gRPC has grown to be pretty prominent. In fact, we started out at Twitch using gRPC. It didn't gain a lot of traction, though — we had some problems with it, and these problems spurred us to create Twirp. There were four core problems in gRPC for us which Twirp solved:

1. Lack of HTTP 1.1 support: gRPC only supports http/2, because its protocol relies upon HTTP Trailers and full-duplex streams. Twirp supports HTTP 1.1 and http/2. This is important because many load balancers (both hardware and software) only support HTTP 1.1 — including AWS's ELBs, which sit in front of EC2 instances. The downside is that Twirp doesn't support streaming RPCs, but those are rare at Twitch — we've found that pretty much all of our services have request-response workloads.

2. Large runtime with breaking changes: gRPC is very complex, so the Go generated code is relatively thin and calls into a large runtime called grpc-go. The generated code and the runtime library are tightly linked and need to match closely. Unfortunately, that runtime has seen breaking changes, sometimes **without warning or explanation**. This would be merely annoying, but becomes a *real* pain when you have a large network of services communicating amongst themselves, importing each others clients.

Breaking changes in the runtime mean that old client code no longer compiles; this means that clients need to use the same gRPC runtime version as that of the services they depend on. The same is true for *those* services dependencies, and quickly this means that *all* of us at Twitch need to use the same version of gRPC in lockstep. Go dependency management is famously rough, so in practice this means we could never upgrade without everyone stopping work and

coordinating an upgrade together, even in the face of bugs.

To make matters worse, the grpc-go runtime requires a particular version of the Go protobuf library, **github.com/golang/protobuf**, also enforced at the compilation level — so we had the same problem ever upgrading protobuf. In practice, we almost never really upgraded, even in the face of severe bugs.

In contrast, Twirp sticks almost everything into the generated code, so it's fine for different services to upgrade at their own pace. We've taken compatibility-breaking changes *extremely* seriously and helped legacy systems continue to work.

3. Bugs due to the complex runtime: grpc-go includes a _*complete* http/2 implementation_, independent of the standard library, and introduces its own flow-control mechanisms. This stuff is very difficult to understand quickly and can lead to **confusing**, **counterintuitive**, and even **totally-broken behavior** (that last bug caused several outages at Twitch). **Leaks are** not unheard of because of the internal complexity of the project, which is a complete deal-breaker for long-running, high-availability services.

Twirp, by contrast, can use plain old HTTP 1.1, which might not be blazingly efficient but at least it's simple and we know how to work with it, and the standard library's HTTP 1.1 implementation is rock-solid. And if you *do* need the extra boosts of http/2, Twirp can use it too — it doesn't ditch the efficiency, just the complexity.

This isn't meant as a bash on the grpc-go team. They do terrific work. But the reality is that the standard library's HTTP implementation is *always* going to be better-tested and more broadly used than the custom one in grpc-go.

4. Difficulty working with binary protobuf: gRPC only supports binary protobuf payloads. Twirp supports the binary encoding, but also supports protobuf's JSON encoding for payloads, using the

official JSON mapping spec.

Allowing JSON has two advantages: for one, makes it easier to write cross-language and third-party clients of Twirp servers — getting the protobuf right by hand is really hard, but getting JSON right by hand is doable.

Second, it makes it easier to write quick command-line cURL requests to debug a running server on the fly. This is the sort of small quality-of-life thing that really can make a difference in the long run, and can especially help when first setting a service up — gRPC felt completely opaque, while it's clear how to quickly check your new Twirp service.

That said, gRPC does have some benefits. It might be worth the costs for you. In particular, gRPC supports bidirectional streaming RPCs, sending flows of uninterrupted data back and forth between client and service. Twirp has nothing like this — just plain old request-and-response. We haven't really missed this at Twitch, but it might be important for your problems, and if so, gRPC is pretty much the only game in town.

And Twirp takes a minimalist approach (the server is just a http.Handler, the client is nothing special), grpc-go is practically overflowing with ambitious add-ons and extras, from a load balancing library to a name resolution framework to let you plug in your own DNS alternative, if that's your thing. We've preferred Twirp's modular approach, letting dedicated load balancing software handle load balancing, but grpc-go's all-in-one system might interest you.

Twirp in production

So, to sum up: If you're writing backend services, you shouldn't be generating and parsing HTTP requests by hand. You should be using

a structured RPC system. If you work in Go, we think you could benefit from using Twirp. It's performed very, very well for us at Twitch, powering dozens of backend systems.

The simplicity of Twirp is something we're especially proud of, since it makes it easy for engineers on the job to understand their own systems. But hopefully, you won't need to. You can get back to the business of writing code that supports your product, rather than trolling through docs to track down whether this request should be a PUT or a POST or a DELETE request.

That simplicity also leads to stability. We have not seen a single outage related to Twirp in the last year — which isn't surprising, as there isn't much that *could* break in Twirp.

Get involved!

We're publishing Twirp because we think it's really effective and we think everyone could benefit from it. But we're also hoping that the community will get engaged and help us write more implementations across languages. We've provided a tool to help with this called 'clientcompat' which runs a battery of tests against a client written in any language. It's a little rough around the edges, but we want it to be better.

Remember, if you have feedback or contributions, I want to hear from you. You can reach me in the **gophers slack** as @spenczar, or email as **spencer@twitch.tv**.