Twirp

A really simple RPC framework

17 January 2018

Spencer Nelson

HTTP API Design

StreamTracker

StreamTracker is a service that tracks the state of live streams on Twitch.

Streams can start, and can then end.

A stream is associated with a **broadcaster**.

Good ol' HTTP

Let's give it a shot:

Start a stream:

```
PUT /streams <broadcaster ID>
```

End a stream:

```
DELETE /streams/:id
```

But wait

Start a stream:

```
PUT /streams <broadcaster ID>
```

- Should it be PUT?
- Should it be POST?
- Should it be /stream?
- Should it be /stream/:id?

The right answer

The right answer

Stop thinking in HTTP.

Start thinking in **functions**

Structured RPCs

- Write a spec describing your API.
- Generate a client and server from the specification.
- Just call functions.

Meet Twirp!

Twirp is an RPC framework for Twitch with a goal of simplicity:

- Clarity in development over flexibility
- Simplicity over performance
- No weird dependency tangles

Used in production today by dozens of backend services on different teams (and growing fast).

What it does

You write a protobuf spec file describing your API:

```
// rpc/service.proto:
service StreamTracker {
  rpc StartStream(Stream) returns (StreamSession);
}

message Stream {
  string channel_id = 1;
  // formats is the list of available quality levels for the stream.
  repeated string formats = 2;
}

message StreamSession {
  string id = 1;
  Stream stream = 2;
}
```

Services can have multiple methods

```
// rpc/service.proto:
service StreamTracker {
  rpc StartStream(Stream) returns (StreamSession);
  rpc EndStream(StreamSession) returns (EndStreamResponse);
}

message Stream {
  string channel_id = 1;
  // formats is the list of available quality levels for the stream.
  repeated string formats = 2;
}

message StreamSession {
  string id = 1;
  Stream stream = 2;
}

message EndStreamResponse {}
```

Generating code from a proto file

You generate code using `protoc-gen-twirp`, which is a protoc plugin:

```
go get github.com/twitchtv/twirp/protoc-gen-twirp
protoc --go_out=. --twirp_out=. rpc/service.proto
```

This produces a file, `rpc/service.twirp.go`, which has everything you need:

- a **interface** describing the service
- a function to make a **server**
- a function to make a **client**

Generated code: Interface

The interface lines up with the proto definition cleanly:

```
// rpc/service.proto
service StreamTracker {
  rpc StartStream(Stream) returns (StreamSession);
  rpc EndStream(StreamSession) returns (EndStreamResponse);
}
```

```
// rpc/service.proto
type StreamTracker interface {
    StartStream(context.Context, *Stream) (*StreamSession, error)
    EndStream(context.Context, *StreamSession) (*EndStreamResponse, error)
}
```

Generated code: Servers

If you implement that interface, you can serve it:

```
func main() {
    s := twirptalk.NewStreamTrackerServer(&streamLogger{}, nil)
    log.Println("launching on port 12345")
    if err := http.ListenAndServe(":12345", s); err != nil {
        log.Fatalf("unable to run: %s", err)
    }
}
```

Generated code: Clients

If you know where a server is running, you can connect to it:

```
func main() {
    c := twirptalk.NewStreamTrackerProtobufClient("http://127.0.0.1:12345", http.DefaultClient)

    everySecond(func(idx int) {
        resp, err := c.StartStream(context.Background(), &twirptalk.Stream{
            ChannelId: fmt.Sprintf("%d", idx),
            Formats:   []string{"high", "medium", "low", "mobile", "source"},
        })
        if err != nil {
            log.Fatalf("err: %s", err)
        }
        log.Printf("stream %d started, session id=%s", idx, resp.Id)
    })
}
```

What just happened?

We wrote a protobuf specification file.

We generated Go code from that spec, using `protoc-gen-twirp`.

The generated code handled serialization and routing for us.

There's no chance to get parameters or types wrong: users of your API just import the client, which "just works."

Compare this to REST

In a **REST** API:

- What HTTP method do I use?
- What's the URL scheme?
- Which parameters go in URL path segments, and which go in the body of the request?
- What error codes do I use?
- What headers do I use?
- How do I serialize requests?

In **Twirp**:

- What methods do I serve?
- What are the types for requests and responses?

Let's talk about gRPC

Our problems with gRPC

- http/2 can be a dealbreaker
- Big runtime requires lockstep upgrades across the org
- Big runtime has bugs
- Inconvenience of protobuf-only

gRPC: http/2 only

- Load balancers might only work on HTTP 1.1 (like AWS ELBs)
- Varnish only supports http/2 experimentally in bleeding-edge release
- Only upside is full-duplex streaming RPCs, which are rare

This blocks adoption if a team already has infrastructure they know how to use.

Twirp: http/2 optional

Twirp **can** run on http/2, or HTTP 1.1. It just uses `net/http`, so you don't really think about it.

gRPC: big runtime makes upgrades hard

Generated code is thin and links to a big runtime, which must be API compatible.

What if I generate code today, then the runtime changes, then you generate tomorrow and use my generated client? (Answer: 💥).

Everyone in the dependency graph **must** use the same version the code generator **and** the same version of the runtime.

In practice, we never upgraded.

Twirp: minimal runtime

Twirp puts nearly everything into the generated code.

Generated servers have legacy logic included to support old generations of clients.

In practice, upgrading is simple: just regenerate.

gRPC: big runtime has bugs

gRPC has a full http/2 implementation, with its own flow control, separate from stdlib.

That turns out to be really hard to do right.

Some **fixed** bugs:

- Race conditions during connection setup.
- If a client cancels a request, its later requests can be throttled down to about 15 bytes/sec.
- Servers can panic if a client disconnects at the wrong moment.

Twirp: so simple that bugs have been very rare

Twirp just uses the standard library's net/http package, which is very, very, very sturdy.

We haven't had any Twirp bugs that have caused production issues so far.

gRPC: Protobuf only

Difficult to hand-craft a protobuf request on the command line when debugging quickly.

Not feasible to write clients in other languages.

Twirp: Allows JSON

Makes it possible to write a valid request in cURL:

```
curl --header 'Content-Type:application/json' \
     --data '{"channel_id": 98210, "formats": ["high", "medium", "low"]}' \
     localhost:12345/twirp/examples.twirptalk.StreamTracker/StartStream

# Output:
{"id":"498081","stream":{"channel_id":"98210","formats":["high","medium","low"]}}
```

Cross-language support has been easy, which is very important for adoption.

Simplicity is a feature

Thank you

Spencer Nelson
spencer@twitch.tv (mailto:spencer@twitch.tv)