Overview

This series will cover multiple aspects of API design, with a specific focus on the components that make up a multi-service system.

Then we will put those designs into practice.

By the end of this article you should have a good understanding of different needs of Interservice vs Public APIs, and be generating half of your code from protobufs.

By the end of this series you should be telling everybody around you how to write microservices at every opportunity until they ask you to stop but you won't.

Technologies Used in This Series

- go *
- kubernetes
- docker *
- grpc *
- prometheus
- protobuf *
- zipkin
- opentracing-go
- urfave/cli
- wercker/blueprint *

Concepts Covered in This Series

- General API Design
- Interservice APIs *
- Public APIs *
- Protocol Buffers *
- Authentication and Authorization
- Public Gateways

- Internal RPC
- Service Metrics
- RPC Tracing
- Command-line Flags
- Deployment
- Service Templates and Auditing

Interservice APIs

In addition to the General API Design in Part 1, there are a few good decisions to make when designing your APIs for communication between your own services.

When communicating between services you know, you are often optimizing for integrity, performance, and development velocity.

Use RPC

Back in The Day(tm), when REST was new and the internet was made of milk and honey, we decided that REST was the answer to all questions.

REST still has a lot going for it, but for Service-to-Service calls within your own infrastructure the tooling around RPC has improved so much there is no longer any reason not to use RPC-style calls instead.

Some of the benefits of RPC over REST:

- Can map directly to your actual method calls.
- Easier to test using fakes, mocks, or loopback interfaces (no HTTP clients and servers to emulate).
- Due to direct mapping and defined protocols, lends itself well to tracing and metrics.

Overall, REST creates a translation layer between your requests and your actual method calls that need not be there for your internal services.

Specifically, Use gRPC

There are plenty of good reasons to choose gRPC as your protocol for your interservice APIs. You'll find that it:

- Almost certainly supports the languages that you are using to write your services. There are 11 tutorials available at https://grpc.io/docs/
- Is well documented but you'll probably not even need docs.
- Has a robust ecosystem integrating many of the tools you likely should be using: https://github.com/grpc-ecosystem/awesome-grpc
- Works very well with all the other concepts we'll be going over in this series of articles.
- Gives you useful features like auto-reconnect with backoff strategies.
- Operates using HTTP/2, which allows faster and more efficient requests.

Auto-Generate Your Clients from Schemas

Auto-generation is not only much easier than writing your own clients, it is also much more reliable.

When generating your API clients from schema, you:

- Radically reduce the footprint of code that you are maintaining.
- Encourage consistency through a much smaller, declarative definition.
- Move closer to a single source of truth to test against.
- Probably can generate a lot more than just your clients!

More on the specifics of how to do this later in the protobuf sections.

Kubernetes Can Help

If you are running your services on Kubernetes, as you should be if you take the advice of this humble writer, you can leverage a few more useful design patterns in your APIs:

- Services! Kind of a no-brainer, this concept in Kubernetes is explicitly designed to abstract away the need for discovery, load balancing, and routing down to simple DNS names.
- Namespaces! Both for simple reasons like grouping resources specific to a service as well as bigger ones like access control.
- Labels! You can attach metadata to your instances and use it to route requests for versioning, canaries, and staging.

Public APIs

Much of the purpose of a Public API is to present the easiest to use tool for the widest range of users. As such, your public interfaces tend towards a lower common denominator.

Use RESTish

Unlike RPC, which is great when you are living in code and importing libraries (and especially great when you own both sides of the interface), REST is particularly valuable when you want your interface to be accessible from basically anything with a digital pulse (and probably some things without).

There is no limit to the number of HTTP clients in the world and by providing a public RESTish API many of your users will be able to solve their API use case with 1 or 2 lines of a shell script.

Don't go overboard on REST theory, just try to make consistent decisions about your POST, PUT, and PATCH calls.

And, of course, you should:

Auto-Generate Your RESTish APIs from Schema

Just like with RPC, there are tons of reasons *not* to write your REST interfaces out by hand.

About the only thing that you should have to define differently from you interservice schema is attaching the URL routes to the calls that will be public.

Generate Swagger

The same schema used to generate your RPC and REST interfaces can be used to generate a Swagger schema for your API as well.

Swagger is a reasonably common schema format for API definition, and, while it is not very pleasant (or recommended) to write by hand, it makes it very easy for a large set of clients in many languages to be generated on the fly to interact with your API.

Kubernetes Can Help

Kubernetes offers a couple very pleasant features that make your public interfaces much easier to manage.

The first is the idea of an Ingress controller. This concept lets you define routes to your services from your "ingress" point, usually a load balancer pointing to an nginx router. It is very configurable and supports a wide variety of features.

Read more about Ingress https://kubernetes.io/docs/concepts/services-networking/ingress/

Ingress also makes use of a Kubernetes feature called Annotations. These are extra configuration options that can be added to routes to enable things like authentication, caching, and URL rewriting.

Protocol Buffers

Protocol Buffers have a weird name (buffer?) for things that are effectively defining the message and RPC schemas for your life.

Protocol Buffers (protobufs from here on), came out of Google, and there is a similar system from Facebook named Thrift, but we're only covering protobufs here as there is already a bunch to cover.

Some pre-reading:

1. A great starting point: https://developers.google.com/protocol-buffers/
2. GRPC is the next thing to look at: https://grpc.io/
3. Why to choose protobufs over JSON: https://codeclimate.com/blog/choose-protocol-buffers/

Messages and gRPC

The base component of your protobufs are the message schemas:

```
enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

message PhoneNumber {
  string number = 1;
  PhoneType type = 2;
}

message Person {
  int32 id = 1;  // Unique ID number for this person.
  string name = 2;
  string email = 3;
  repeated PhoneNumber phones = 4;
}
```

If you're familiar with schemas these shouldn't look too unusual. They give you a reasonable amount of flexibility while still requiring a strict enough format that the resulting interfaces are predictable.

The documents linked above can explain what all those fields mean in depth and which others are available. I'm just going to add in a few useful tips about them before we move on:

- Whenever possible, use the same models and fields as you are using in your datastore: you will almost certainly be writing methods to convert your db objects to protobufs at some point, so make it easy on yourself.
- Decide on some reasonable ordering scheme for your fields and try to use it everywhere. This isn't super huge but it reduces the cognitive load of anybody (read: you in two weeks) who has to read the code. I usually do:
  - id / primary key
  - name / title
  - additional meta (email, type, content, created_at)
  - references to other models (user, group, org, project)
  - any repeated fields
- Remember that you'll be able to change your schemas a bunch as long as others haven't started a) storing them or b) somehow aren't using generated clients; don't let people do either if you can help it!

For most usages, however, the message definitions are only half the battle. Enter gRPC where you will define your service API:

```
// The MaitreD talks to people waiting for tables at a restaurant
service MaitreD {
  // Append somebody to the queue.
  rpc Append(AppendRequest) returns (AppendResponse) {
    option (google.api.http) = {
      post: "/v1/queue"
      body: "*"
    };
  }
  // Get the next person from the queue.
  rpc Next(NextRequest) returns (NextResponse) {
    option (google.api.http) = {
      get: "/v1/queue/next"
    };
  }
}

message AppendRequest {
  Person person = 1;
}

message AppendResponse {
  // Timestamp of when they joined the queue
  Timestamp timestamp = 1;
}

message NextRequest {}

message NextResponse {
  // Timestamp of when they joined the queue
  Timestamp timestamp = 1;
  Person person = 2;
}
```

Most of that probably looks how you'd expect it to except for those option clauses. The options are extra annotations imported from another schema to define what URLs your RESTish gateway will serve once you get around to generating that.

More on that: https://github.com/grpc-ecosystem/grpc-gateway

Some gRPC tips:

- If you name everything {rpc name}Request and {rpc name}Response you'll never have to remember the names.
- You're going to have plenty of Request and Response objects. If you make the Responses look like a wrapper around one of your common messages (models), instead of duplicating the content, you'll save yourself a lot of work in typing out field names.
- Because you'll be generating so much of your code, you can spend less time getting everything perfect to begin with. Just make sure to allot some time every week or so to making sure your interfaces are uniform and predictable.

You'll be following a bunch of tutorials when you get started, so we won't go into all the details of the syntax. The existing (linked) documentation is all very good.

Generate All the Things!

Protobufs are "compiled", but for most situations what that actually means is that they are fed into a series of tools to generate code.

Common things you'll be generating are:

- API Models (core protobuf)
- API Docs (protoc-gen-doc)
- RPC servers (grpc)
- RPC clients (in multiple languages, go-openapi)
- REST Gateway (grpc-gateway)
- API definition docs (e.g. Swagger)

Less-common-but-totally-awesome things you can be generating are:

- RPC Tracing code (grpc-opentracing)
- Database CRUD stubs!
- Test stubs!
- GraphQL!
- JS Flow Types!

For a big list, see: https://github.com/grpc-ecosystem/awesome-grpc

Your Protobuf Ecosystem

One of the initial hurdles for teams starting with protobufs is deciding on how to share and reference both the protos and the generated files. We found the following things to be helpful:

Directory Layout

Protobufs have the concept of a namespace, which generally is just a directory path. So let's say you have a project named "identity" in Go.

```
$GOPATH/src/github.com/wercker/identity
```

If you want to be able to import github.com/wercker/identity in your protobufs you'll want your .proto file to live in the root of that.

```
../identity/identity.proto
```

However, you'll be generating a lot of code, and it can be a pain to have that all in your root dir, so we like to name our go-package (in the .proto file) ${dir}pb, so that we get a code structure after generation of:

```
.../identity/identitypb/identity.pb.go
.../identity/identitypb/identity.pb.gw.go
.../identity/identitypb/identity.swagger.json
```

Which makes it pretty easy to import the protobuf file itself in another project, as well have a clear distinction in your application code about which code is generated vs hand-written.

Consolidate Your Swagger

A couple of your API definitions might need to be accessible from other languages where it doesn't make sense to import your codebase; things like swagger and graphql. For these, you'll want your CI/CD system to upload the files to a public server after each release (assuming publicly used; use a private server if they are only used internally.)

Stick to the Same Version

One pitfall early on is that people may have slightly different versions of the protobuf libraries installed, which results in slightly different (or possibly broken) generated code, which leads to ugly merges and confused developers.

We recommend keeping a Docker image with all the correct protobuf libraries installed and having that be the golden truth for generating the code. An example of how to do that, assuming our identity service from above:

```
#!/bin/bash
set -e

LOCAL=$(dirname $PWD)

# Check for docker and use it, with common paths included
# NOTE: that's a public docker image, feel free to try it out
if [ -e /var/run/docker.sock ]; then
  ROOT=${LOCAL//$GOPATH/\/go}
  protoc="docker run --rm \
    -u $(id -u $USER):$(id -g $USER) \
    -w $ROOT \
    -v $LOCAL:$ROOT \
    quay.io/wercker/protoc"
# Otherwise run it using local protoc, useful when testing CI
else
  ROOT=$LOCAL
  protoc="protoc \
    -I/usr/local/include
    -I.
    -I$GOPATH/src \
    -I./vendor \
    -I./vendor/github.com/grpc-ecosystem/grpc-gateway/third_party/googleapis"
```

```
fi

cd $LOCAL

echo "Generating gRPC server, gateway, swagger, flow"
$protoc --go_out=plugins=grpc:$ROOT/identitypb \
        --grpc-gateway_out=logtostderr=true,request_context=true:$ROOT/identitypb \
        --swagger_out=logtostderr=true:$ROOT/identitypb \
        --flow_out=$ROOT/identitypb \
        identity.proto
```

That code will output with the directory structure we described above.

A newer tool that was recently released to the public is `prototool` that tries to give you a reliable build environment for your protobufs. Check it out here: https://github.com/uber/prototool

Use go generate

We found it convenient to reference the script above from a file named proto.go inside the identitypb directory:

```
package identitypb

//go:generate ./generate-protobuf.sh
```

Commit the Generated Code

This one is probably more controversial, but we found it most helpful to commit the generated code rather than regenerating it every time.

Once the project is somewhat mature it doesn't need to be generated as often, and since it touches a lot of code you usually want a human involved to check the work after.

More on Microservices

- Introduction to Microservices
- Microservice API design

License