gRPC, REST, and Go...

How to write web services in Go without hating your life

Jeff Martin
Software Engineer @ Weave

A bit about me

- # of kids has multiplied substantially
- Still recovering from being a Java engineer for 15 years
- Conversion to Go began in 2015… but took a few years
- Love working at Weave

Writing Web Services Should not be Painful

For example...

DELETE /products/{id}

- How do I know if the call “worked”?
  - Response status code under happy path? (204?, 202?, 200?)
  - Response status code if the resource has already been deleted? … (404? 204? 200?)
- If there is an error, what is the format of the error message returned?
- Will the response body be empty if the call was successful?
- Naming things is hard… should the resource be “product” or “products”?
- Have I truly reached “The Glory of REST”?

Why gRPC?

“Because that’s what the engineers before me decided to do”

Better reasons:

- Well-defined contracts
- Code generation for client/server, directly from the contract
- Performance
- Supports most sane backend languages (even some it shouldn’t):
  - C#, C++, Dart, Go, Java, Kotlin, Node, Objective-C, PHP, Python, Ruby

gRPC, REST Comparison

Feature	gRPC	HTTP APIs with JSON
Contract	Required (*.proto*)	Optional (OpenAPI)
Protocol	HTTP/2	HTTP
Payload	Protobuf (small, binary)	JSON (large, human readable)
Prescriptiveness	Strict specification	Loose. Any HTTP is valid.
Streaming	Client, server, bi-directional	Client, server
Browser support	No (requires grpc-web)	Yes
Security	Transport (TLS)	Transport (TLS)
Client code-generation	Yes	OpenAPI + third-party tooling

Source: https://docs.microsoft.com/en-us/aspnet/core/grpc/comparison?view=aspnetcore-5.0

gRPC Basics: Define your Contract (proto)

```
syntax = "proto3";

option go_package = "github.com/jmartin127/dashboard/proto/gen/go/trafficv1";

package traffic.v1;

import "google/protobuf/duration.proto";

service Traffic {
  rpc GetTravelTime (GetTravelTimeRequest) returns (GetTravelTimeReply) {}
}

message GetTravelTimeRequest {
  string originAddress = 1;
  string destinationAddress = 2;
}

message GetTravelTimeReply {
  google.protobuf.Duration travelTime = 1;
}
```

- Service defines which functions are available for this service
- Request/Reply messages clearly document input/output
- Strongly typed (JSON only supports string, number, object, array, boolean, null)
- In this example we return a Duration! See well-known types of gRPC.

gRPC Basics: Code Generation (protoc)

```
protoc -I . \
  --go_out=./gen/go/ --go_opt=paths=source_relative \
  --go-grpc_out=./gen/go/ --go-grpc_opt=paths=source_relative \
  traffic/v1/traffic.proto
```

- Protocol Buffer Compiler (aka protoc)
- If your proto file has dependencies, protoc must be able to find them
- Configure where generated code should reside within your project
- Ideally, bake protoc into a **Docker** image for consistently generated code

gRPC Basics: Code Generation Output

Traffic_grpc.pb.go (Client/Server generation)

Client code generated:

```
type TrafficClient interface {
    GetTravelTime(ctx context.Context, in *GetTravelTimeRequest, opts ...grpc.CallOption) (*GetTravelTimeReply, error)
}
```

Server code generated:

```
func RegisterTrafficServer(s grpc.ServiceRegistrar, srv TrafficServer) {
    s.RegisterService(&Traffic_ServiceDesc, srv)
}
```

traffic.pb.go (struct generation)

```
type GetTravelTimeRequest struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    OriginAddress      string `protobuf:"bytes,1,opt,name=originAddress,proto3" json:"originAddress,omitempty"`          // By default maps to URL query parameter `origin`.
    DestinationAddress string `protobuf:"bytes,2,opt,name=destinationAddress,proto3" json:"destinationAddress,omitempty"` // By default maps to URL query parameter `destinatio
}
```

gRPC Basics: Implement Server Functions

Writing the server-side code is as simple as implementing the generated interface:

```
func (s *server) GetTravelTime(ctx context.Context, in *pb.GetTravelTimeRequest) (*pb.GetTravelTimeReply, error) {
    return &pb.GetTravelTimeReply{TravelTime: durationpb.New(11 * time.Minute)}, nil
}
```

Exercising the client code is straightforward as well, just a function call:

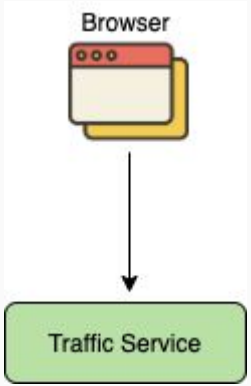
```
getTravelTimeRequest := &trafficpb.GetTravelTimeRequest{
    OriginAddress:      "Ogden, Utah",
    DestinationAddress: "Lehi, Utah",
}
travelTime, err := s.trafficClient.GetTravelTime(ctx, getTravelTimeRequest)
if err != nil {
    return nil, status.Error(codes.Internal, err.Error())
}
```

REST is still convenient, but so is gRPC

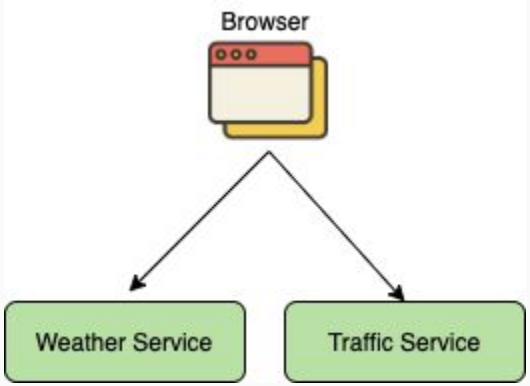
Front-end Engineer:
“I want a single payload that represents exactly what I want to display on the front-end… Oh, and it must be JSON.”

Back-end Engineer:
“I want my backend services to have sub-millisecond performance, easy to augment, and millions of microservices… Isn’t JavaScript a toy language?”

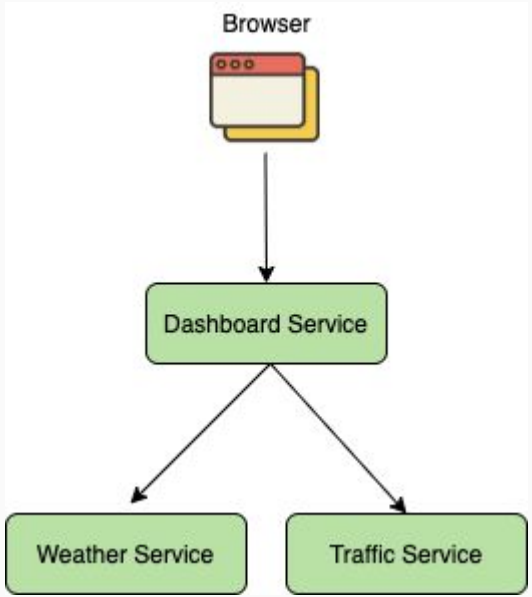
Evolving Architecture



Evolving Architecture

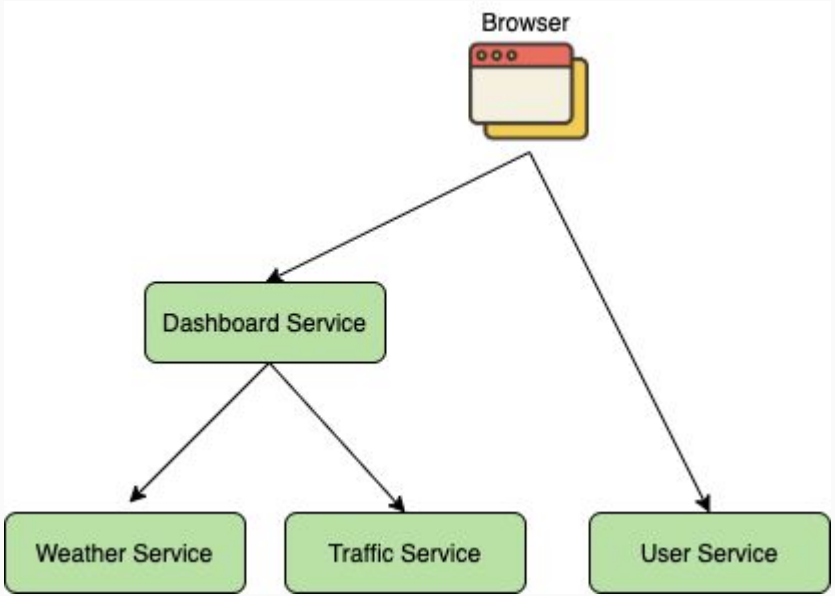


Evolving Architecture



Front-end:
“I Need a BFF!”

Evolving Architecture



Evolving Architecture

gRPC ⟷ REST Translation == Boilerplate Code

```go
func (s *server) myHandler(w http.ResponseWriter, r *http.Request) {
    // make gRPC request
    weather, err := s.weatherClient.GetCurrentWeather(r.Context(), &weatherpb.GetCurrentWeatherRequest{Address: "..."})
    if err != nil {
        log.Printf("Body read error, %v", err)
        w.WriteHeader(500) // Return 500 Internal Server Error.
        return
    }

    // convert gRPC reply to the REST response body
    data := CurrentWeather{
        TempFahrenheit:   weather.TempFahrenheit,
        PrecipitationPct: weather.PrecipitationPct,
        HumidityPct:      weather.HumidityPct,
        WindMPH:          weather.WindMPH,
    }

    // write the REST response
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    json.NewEncoder(w).Encode(data)
}
```

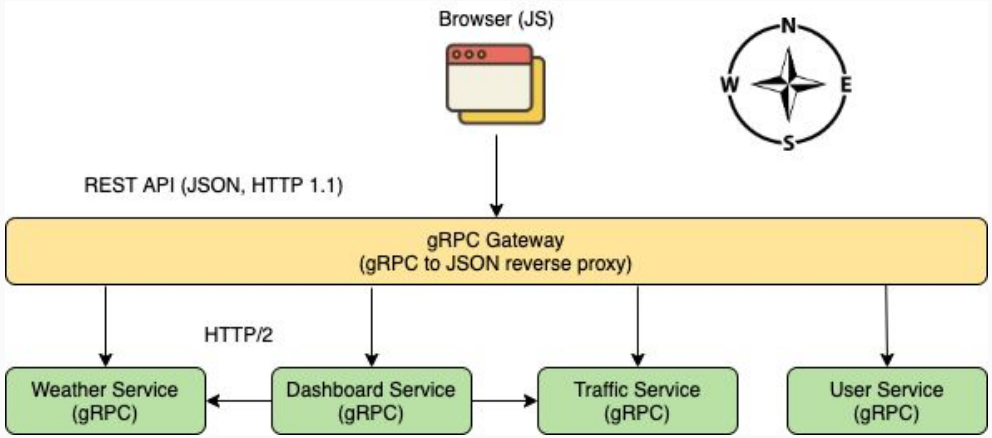
One Option…

Write transformation code yourself…

Did I enjoy writing that transformation code? Am I a better person because of it? Would I ever want to do that again? Do I value my own self-worth?

The marriage of gRPC and REST

gRPC Gateway to the Rescue!



- North/South Traffic is still REST, JSON, but proxied to gRPC
- Edge APIs exposed to the Browser are REST
- Service-to-service interaction is gRPC with well-defined contracts
- Both gRPC and REST contract are defined in the proto

Adding REST API Annotations to be used by the Gateway Proxy

```
service Traffic {
  rpc GetTravelTime (GetTravelTimeRequest) returns (GetTravelTimeReply) {
    option (google.api.http) = {
      get: "/traffic/travelTime"
    };
  }
}
```

- Readable REST API annotations in your proto… ***THE*** contract
- gRPC Request parameters are automatically converted to REST Query Parameters
- gRPC status codes are automatically converted to REST status codes
  - NOT_FOUND (gRPC) → 404 Not Found (REST API)

REST API Annotations: More Advanced

```
rpc Create(ABitOfEverything) returns (ABitOfEverything) {
        option (google.api.http) = {
                post: "/v1/example/a_bit_of_everything/{float_value}/{double_value}/{int64_value}/separator/{uint64_value}/{int32_value}/{fixed64_value}
        };
}
rpc CreateBody(ABitOfEverything) returns (ABitOfEverything) {
        option (google.api.http) = {
                post: "/v1/example/a_bit_of_everything"
                body: "*"
        };
}
```

“A Bit of Everything” example from the gRPC gateway project:
https://github.com/grpc-ecosystem/grpc-gateway/blob/master/examples/internal/proto/examplepb/a_bit_of_everything.proto

Registering a gRPC Service with the Gateway

```
if err := trafficgw.RegisterTrafficHandlerFromEndpoint(ctx, gwmux, *grpcServerEndpointTraffic, opts); err != nil {
    return err
}
```

- The Gateway code is generated via **protoc-gen-grpc-gateway** plugin
- The generated code makes it easy to register the gateway with your favorite mux (HTTP multiplexer)
- Ok, so now I can make REST API requests via the Gateway, now what?

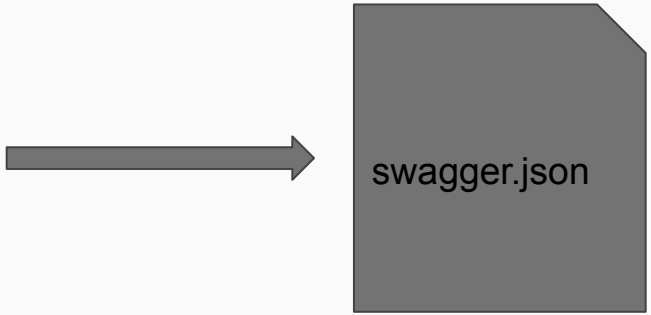

TOP 2 REASONS
YOU DON'T HAVE
NICE THINGS ANYMORE.

OpenAPI (Swagger) Docs

Go Developers CAN have "Nice things"

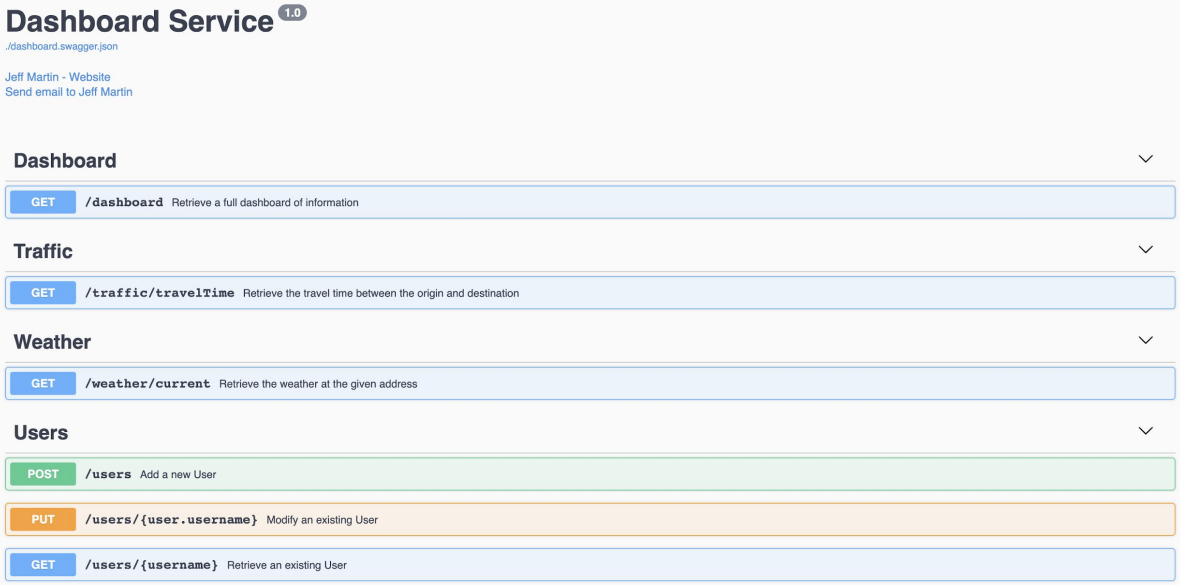
- What if the same REST API annotations could be used to generate OpenAPI Documentation?
- **protoc-gen-openapiv2 to the rescue!**

```
service Traffic {
  rpc GetTravelTime (GetTravelTimeRequest) returns (GetTravelTimeReply) {
    option (google.api.http) = {
      get: "/traffic/travelTime"
    };
    option (grpc.gateway.protoc_gen_openapiv2.options.openapiv2_operation) = {
      summary: "Retrieve the travel time between the origin and destination";
      description: "Use this endpoint to determine the travel time between an orig
    };
  }
}
```



OpenAPI (Swagger) Docs

You can host your swagger.json via your gRPC gateway mux:



Input Validation

No really... Go Developers can have "Nice things"

- We really ***should*** validate input to our APIs and fail fast if input is bogus
- **protoc-gen-validate (PGV) to the rescue!**
- Originally part of the Lyft project, later moved to Envoy
- Supported languages (Go, C++, Java, Python)
- Remember Java Preconditions anyone? I don't miss Guava.

Input Validation Example

```
message User {
  string username = 1 [(validate.rules).string = {
    pattern: "^[a-z0-9_-]{3,16}$",
  }];
  string email = 2 [(validate.rules).string.email = true]; // RFC 1034
  string firstName = 3 [(validate.rules).string.min_len = 1];
  string lastName = 4 [(validate.rules).string.min_len = 1];
}
```

- Lots of handy built-in validation templates for:
  - numerics, bools, strings, bytes, enums, messages, repeated values, maps, well-known types

Input Validation: Generated Code

```go
// Validate checks the field values on User with the rules defined in the proto
// definition for this message. If any rules are violated, an error is returned.
func (m *User) Validate() error {
    if m == nil {
        return nil
    }

    if !_User_Username_Pattern.MatchString(m.GetUsername()) {
        return UserValidationError{
            field:  "Username",
            reason: "value does not match regex pattern \"^[a-z0-9_-]{3,16}$\"",
        }
    }

    if err := m._validateEmail(m.GetEmail()); err != nil {
        return UserValidationError{
            field:  "Email",
            reason: "value must be a valid email address",
            cause:  err,
        }
    }
      You, 6 hours ago • Added user service
    if utf8.RuneCountInString(m.GetFirstName()) < 1 {
        return UserValidationError{
            field:  "FirstName",
            reason: "value length must be at least 1 runes",
        }
    }
```

Generated code is readable...

Likely better than the code many of us would have written!

To use the Validation code, you just call this Validate() function on incoming request objects.

Disclaimer!

protoc-gen-validate (PGV)

*This project is currently in **alpha**. The API should be considered unstable and likely to change*

… Don’t blame me… I’ll just say ***I told you so***

Putting it all together… Code Generation

```
gen:
    cd ../../proto && \
    protoc -I . \
     --go_out=./gen/go/ --go_opt=paths=source_relative \
     --go-grpc_out=./gen/go/ --go-grpc_opt=paths=source_relative \
     --grpc-gateway_out=./gen/go/ --grpc-gateway_opt paths=source_relative \
     --grpc-gateway_opt logtostderr=true \
     --validate_out=lang=go,paths=source_relative:./gen/go \
     jmartin127/traffic/v1/traffic.proto
```

This generates gRPC client/server/messages, gateway, and validation code

Tip: When generating OpenAPI (Swagger) Docs you can use the “allow_merge=true” option to combine them.

Putting it all together… Demo

Key Takeaways

- You ***CAN*** have nice things!
- **gRPC** makes service-to-service client/server interactions as simple as calling another function
- **gRPC Gateway** allows you to expose REST without writing ***any*** REST handlers
- You can **unify multiple gRPC micro/nano/right-sized services** under a single REST Gateway
- **OpenAPI (Swagger) Generation** gives you nice Documentation and a useable UI for ad-hoc testing... and makes front-end engineers not hate you
- **Envoy (Lyft) Validation** automagically generates boilerplate validation code

How do I get started?

GitHub Repo: https://github.com/jmartin127/dashboard

- README includes links to getting started resources

Start now:

- Convince your organization to start with a single service
- Demonstrate how this could streamline your current process and remove boilerplate code
- Come join us at Weave

Other projects to look at:

- grpc-web: A JavaScript implementation of gRPC for browser clients
- Envoy gRPC-JSON transcoder
- Google Cloud Endpoints

Thank You!