Go, Protocol Buffers and Microservices

About me

- Murilo
- +10 years in tech
- background: php, ruby, nodejs, go, linux
- these days: go, rust

Overview

- go
- schema oriented design
- IDLs
- protocol buffers
- grpc
- code generation
- rest

3

Go

- simple syntax/easy to learn
- standard formatting (gofmt)
- deploy friendly (fast compilation time + static binaries)
- strongly typed (compile-time error checking)
- concurrency primitives
- very fast

https://golang.org/doc/effective_go.html (https://golang.org/doc/effective_go.html)

```
package main

import (
    "log"
    "net/http"
)

func main() {
    json := `{ "message": "hi picpay" }`
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte(json))
    })
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}


$ go run main.go


$ curl -s localhost:8000 | jq
{
  "message": "hi picpay"
}
```

5

```
package main

import "github.com/gin-gonic/gin"

type response struct {
    Message string `json:"message"`
}

func main() {
    router := gin.Default()
    router.GET("/", func(ctx *gin.Context) {
        json := response{
            Message: "hi picpay",
        }
        ctx.JSON(200, json)
    })
    router.Run("localhost:8000")
}
```

```
$ curl -s localhost:8000 | jq
{
  "message": "hi picpay"
}
```

Schema oriented design

- interface definition as part of the process
- specs of a systems behavior
- entrypoint for documentation and discussions
- data structures are central to programming [1] (https://users.ece.utexas.edu/~adnan/pike.html)

```
type Request struct {
  name string
}

type Response struct {
  message string
}

type Service inteface {
  GetMessage(req Request) Response
}
```

IDLs

An interface description language or interface definition language (IDL), is a specification language used to describe a software component's application programming interface (API).

IDLs describe an interface in a language-independent way, enabling communication between software components that do not share one language, for example, between those written in C++ and those written in Java.

Examples:

- OpenAPI
- Protocol Buffers
- Cap'n'Proto
- Avro

Protocol Buffers

Protobuf, or Protocol Buffers, are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler.

https://developers.google.com/protocol-buffers (https://developers.google.com/protocol-buffers) 9

```
// proto/message.proto

syntax="proto3";

package Message.v1;

message MessageRequest {}

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

```
$ protoc \
    -I ./proto \
    --go_out=./generated \
    proto/message.proto
```

10

```
// generated/message.pb.go

type MessageRequest struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields
}

type MessageResponse struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    Message string `protobuf:"bytes,1,opt,name=message,proto3" json:"message,omitempty"`
}
```

```
package main

import (
    "github.com/gin-gonic/gin"
    pb "github.com/mvrilo/presentations/go-protobuf-microservices/generated"
)

func main() {
    router := gin.Default()
    router.GET("/", func(ctx *gin.Context) {
        json := &pb.MessageResponse{
            Message: "hi picpay",
        }
        ctx.JSON(200, json)
    })
    router.Run("localhost:8000")
}
```

```
$ curl -s localhost:8000 | jq
{
  "message": "hi picpay"
}
```

grpc

gRPC is a modern open source high performance RPC framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication. It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

https://grpc.io (https://grpc.io)

- C# / .NET, C++, Dart
- Rust, Go, Java, Kotlin/JVM
- Node.js, Objective-C
- PHP, Python, Ruby

13

```
// proto/message.proto

syntax="proto3";

package Message.v1;

message MessageRequest {}

message MessageResponse {
  string message = 1;
}

service MessageService {
  rpc GetMessage (MessageRequest) returns (MessageResponse) {};
}
```

14

```
$ protoc \
    -I ./proto \
    --go_out=./generated \
    --go-grpc_out=./generated \
    proto/message.proto
```

15

```
// generated/message_grpc.pb.go

type MessageServiceServer interface {
    GetMessage(context.Context, *MessageRequest) (*MessageResponse, error)
}

type MessageServiceClient interface {
    GetMessage(ctx context.Context, in *MessageRequest, opts ...grpc.CallOption) (*MessageResponse, error)
}

func NewMessageServiceClient(cc grpc.ClientConnInterface) MessageServiceClient {
    return &messageServiceClient{cc}
}

func (c *messageServiceClient) GetMessage(ctx context.Context, in *MessageRequest, opts ...grpc.CallOption) (*Me
    out := new(MessageResponse)
    err := c.cc.Invoke(ctx, "/Message.v1.MessageService/GetMessage", in, out, opts...)
    if err != nil {
        return nil, err
    }
    return out, nil
}
```

16

```
package main

import (
    "context"
    "net"

    pb "github.com/mvrilo/presentations/go-protobuf-microservices/generated"
    "google.golang.org/grpc"
)

type messageService struct {
    pb.UnimplementedMessageServiceServer
}

func (m *messageService) GetMessage(ctx context.Context, in *pb.MessageRequest) (*pb.MessageResponse, error) {
    json := &pb.MessageResponse{Message: "hi picpay"}
    return json, nil
}
```

17

```
func main() {
    lis, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        panic(err)
    }

    srv := grpc.NewServer()
    msg := &messageService{}
    pb.RegisterMessageServiceServer(srv, msg)
    panic(srv.Serve(lis))
}
```

```
$ evans --proto proto/message.proto --host localhost --port 8000 repl
Message.v1.MessageService@localhost:8000> call GetMessage
{
  "message": "hi picpay"
}
```

code generation

- fully generated client code
- server interfacing
- multi language support
- parse all the things

and the rest?

Introducing http as a proxy: grpc-gateway

https://github.com/grpc-ecosystem/grpc-gateway (https://github.com/grpc-ecosystem/grpc-gateway)

> The gRPC-Gateway is a plugin of the Google protocol buffers compiler protoc. It reads protobuf service definitions and generates a reverse-proxy server which translates a RESTful HTTP API into gRPC. This server is generated according to the google.api.http annotations in your service definitions.

21

```
syntax="proto3";

option go_package=".;pb";

import "google/api/annotations.proto";

package Message.v1;

message MessageRequest {}

message MessageResponse {
  string message = 1;
}

service MessageService {
  rpc GetMessage (MessageRequest) returns (MessageResponse) {
    option (google.api.http) = {
      get: "/v1/message"
    };
  };
}
```

22

```
package main

import (
    "context"
    "net"
    "net/http"

    "github.com/grpc-ecosystem/grpc-gateway/v2/runtime"
    pb "github.com/mvrilo/presentations/go-protobuf-microservices/generated"
    "google.golang.org/grpc"
)

type messageService struct {
    pb.UnimplementedMessageServiceServer
}

func (m *messageService) GetMessage(ctx context.Context, in *pb.MessageRequest) (*pb.MessageResponse, error) {
    json := &pb.MessageResponse{Message: "hi picpay"}
    return json, nil
}
```

```
func main() {
    serverAddress := "localhost:8000"
    ctx := context.Background()
    srv := grpc.NewServer()
    router := runtime.NewServeMux()
    opts := []grpc.DialOption{grpc.WithInsecure()}

    msg := &messageService{}
    pb.RegisterMessageServiceServer(srv, msg)

    lis, err := net.Listen("tcp", serverAddress)
    if err != nil {
        panic(err)
    }

    go func(srv *grpc.Server, listener net.Listener) {
        panic(srv.Serve(lis))
    }(srv, lis)

    if err = pb.RegisterMessageServiceHandlerFromEndpoint(ctx, router, serverAddress, opts); err != nil {
        panic(err)
    }

    panic(http.ListenAndServe("localhost:8001", router))
}
```

grpc:

```
$ evans --path proto/ --path third_party/googleapis/ --proto message.proto --host localhost --port 8000 repl
Message.v1.MessageService@localhost:8000> call GetMessage
{
  "message": "hi picpay"
}
```

http:

```
$ curl -s http://localhost:8001/v1/message | jq
{
  "message": "hi picpay"
}
```

25

conclusion

- protobuf with go is very friendly
- supporting both grpc and http is easy
- protobuf as source of truth
- helps keeping the code clean

Thank you

Murilo Santana (@mvrilo)(mailto:Murilo%20Santana%20%28@mvrilo%29)

PicPay | Store

Dec 2020