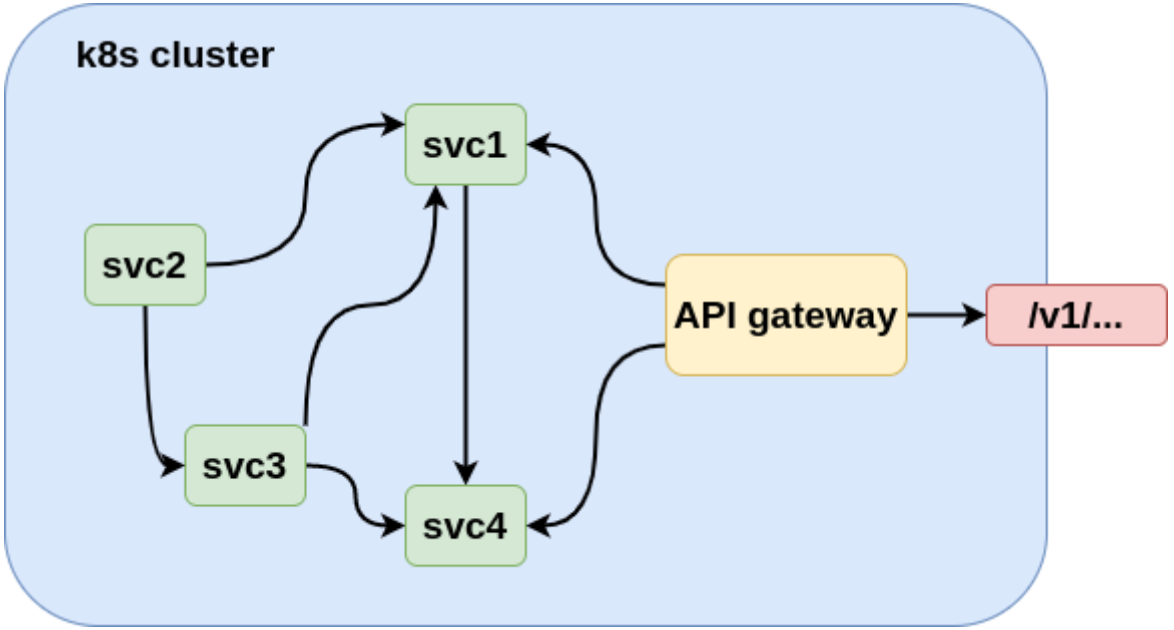
Protobuf-First APIs: gRPC & Co.

October 27, 2020: Boston Golang meetup

Evgeny Khabarov
PowerFly.Consulting
Ottawa, ON, Canada

What is this talk about?

- gRPC & Protobuf.
- Usage cases
- REST API and how to get it for free... if you really need it

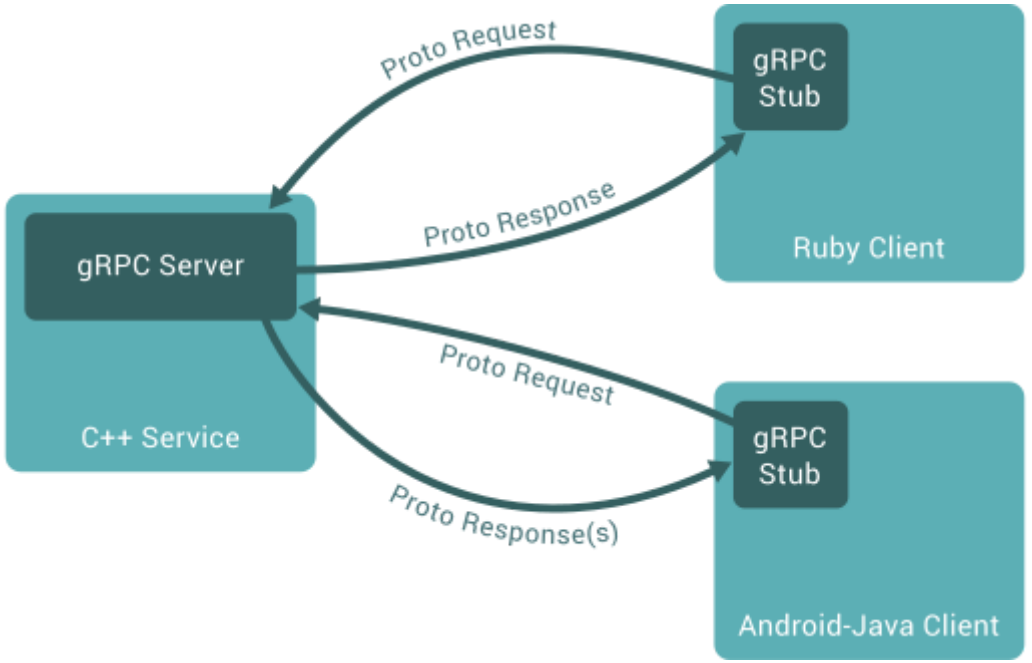


$ whoami

- Golang developer
- work, work, work.... boom
- Co-founder at PowerFly.Consulting
- Work mostly with e-commerce companies
- Time-to-time automate a routine by writing generators
- github.com/ekhabarov (https://github.com/ekhabarov)

What is gRPC?

- A high performance, open-source universal RPC framework. (https://grpc.io)
- Platform and language independent.
- **Uses Protobuf for service definition.**
- Uses HTTP/2 and binary serialization.
- Supports server-, client-, and bidirectional streaming



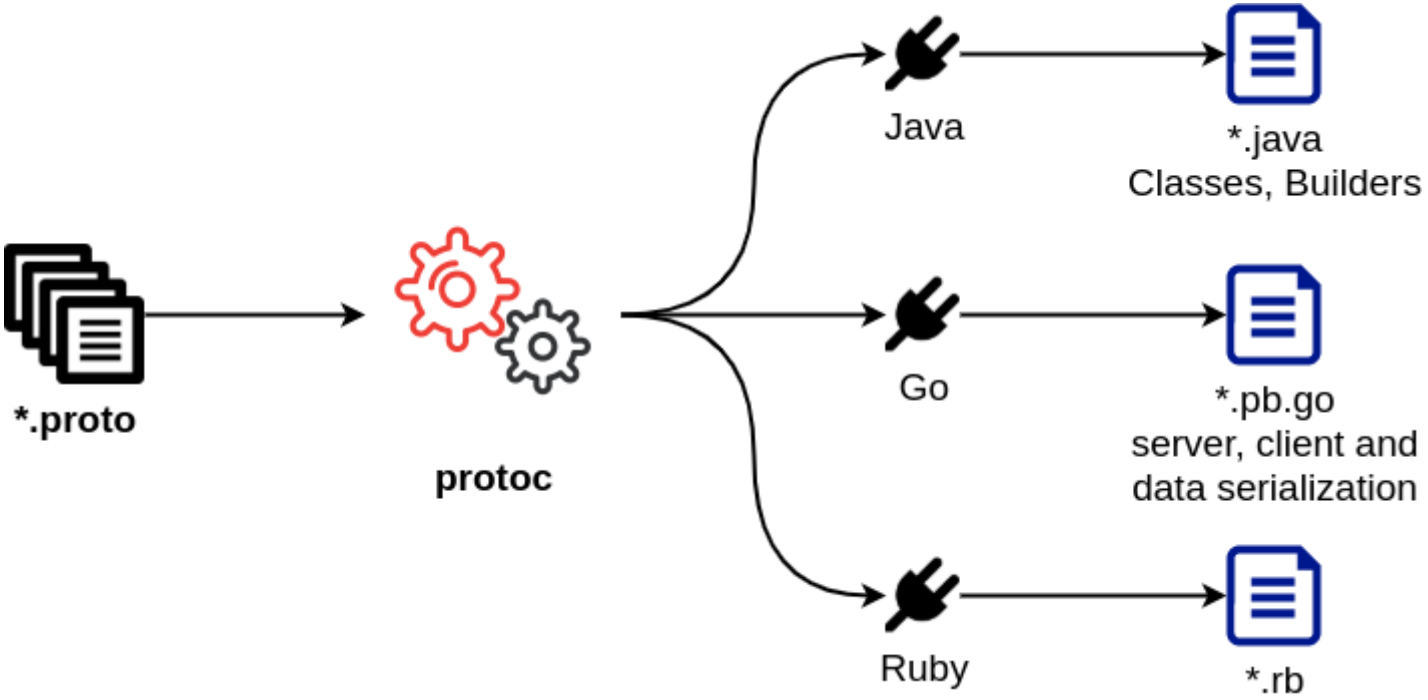
What are Protocol buffers (protobuf)?

- "Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data." (c)
- Interface Definition Language (IDL)
- Clearly defines an API interface in **.proto** files
- Shareable aka generate your own clients.

developers.google.com/protocol-buffers/docs/proto3 (https://developers.google.com/protocol-buffers/docs/proto3)

Protobuf compiler (protoc)

- Parses **.proto** files and passes parsed data to plugins.
- Output depends on plugin.



Protobuf source: example.proto

```
syntax = "proto3";
package greeter;

option go_package = "pb";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string name = 1;
}

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

```
protoc \
  --go_out=plugins=grpc:. \
  ./example.proto
```

Generated code: example.pb.go

```
package pb
```

```
type GreeterServer interface {
    SayHello(context.Context, *HelloRequest) (*HelloResponse, error)
}
```

```
type GreeterClient interface {
    SayHello(ctx context.Context, in *HelloRequest, opts ...grpc.CallOption) (*HelloResponse, error)
}
```

```
type HelloRequest struct {
    Name string `protobuf:"bytes,1,opt,name=name,proto3" json:"name,omitempty"`
}
```

```
type HelloResponse struct {
    Message      string     `protobuf:"bytes,1,opt,name=message,proto3" json:"message,omitempty"`
    ResponseTime *time.Time `protobuf:"bytes,2,opt,name=response_time,json=responseTime,proto3,stdtime"
}
```

- It also contains a bunch of code for serialization.

Implementation

```
type server struct{}

func (s *server) SayHello(ctx context.Context, req *pb.HelloRequest) (*pb.HelloResponse, error) {
    return &pb.HelloResponse{Message: "Hello " + req.Name}
}

func main() {
    lis, err := net.Listen("tcp", ":5001")
    if err != nil {
        log.Fatal(err)
    }

    s := grpc.NewServer()
    pb.RegisterGreeterServer(s, &server{})
    reflection.Register(s)
    s.Server(lis)
}
```

Call it from Go code

```
package client
```

```
func main() {
    cc, err := grpc.Dial("127.0.0.1:5001", grpc.WithInsecure())
    if err != nil {
        log.Fatal(err)
    }
    defer cc.Close()

    client := pb.NewGreeterClient(cc)
    resp, err := client.SayHello(context.Background(), &pb.HelloRequest{Name: "Boston"})
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(resp.Message)
}
```

Call it from command line

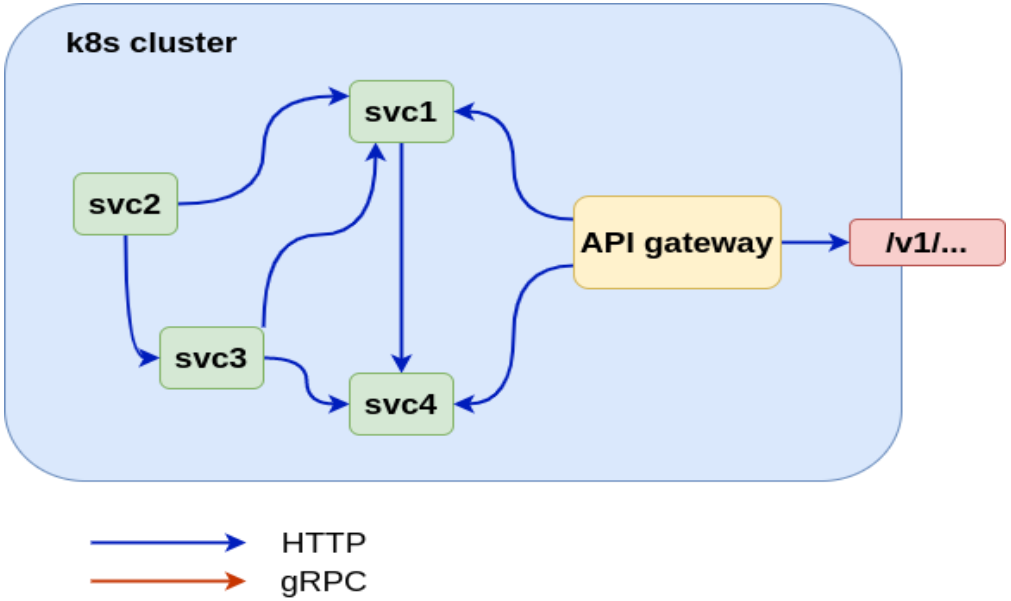
```
grpcurl -plaintext \
  -d '{"name": "Boston"}' \
  127.0.0.1:5001 greeter.Hello.SayHello
```

github.com/fullstorydev/grpcurl (https://github.com/fullstorydev/grpcurl)

Several cases from my experience

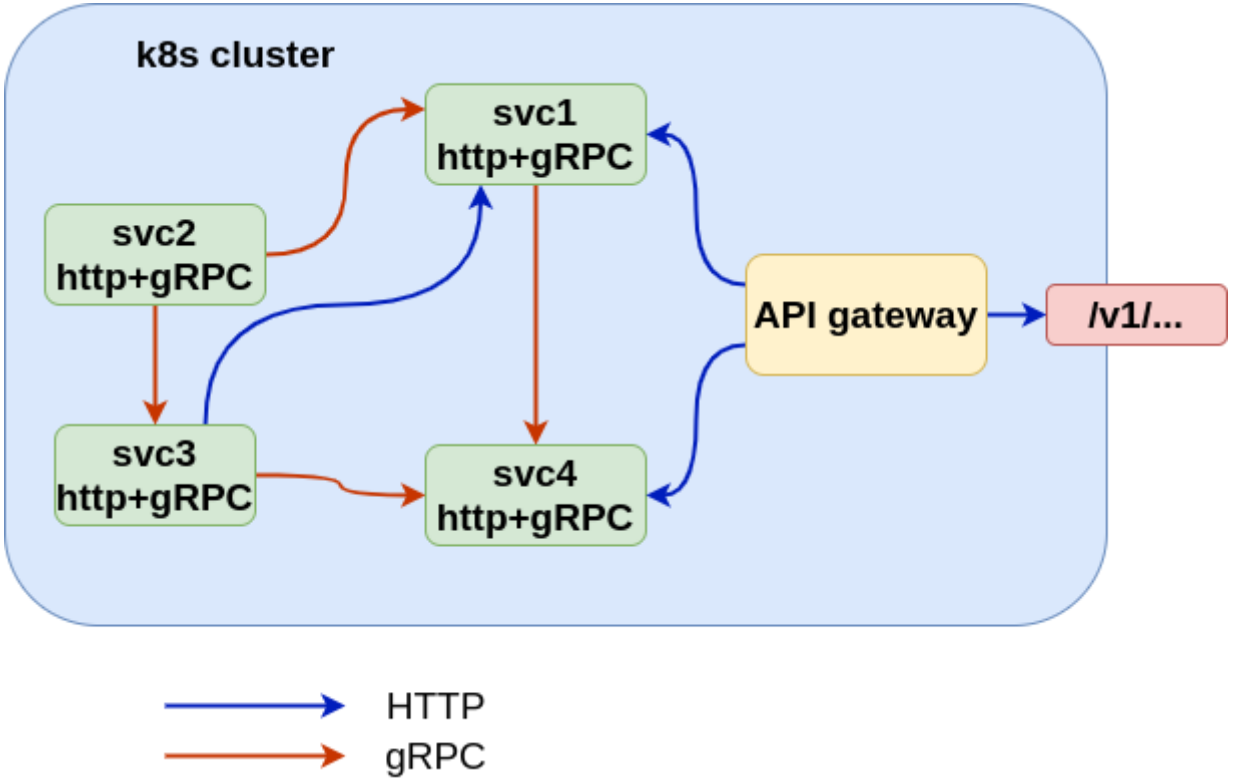
1st iteration (REST + gRPC)

- We've build REST API first
- Used it internally & externaly
- Faced with performance issues
- Added gRPC for internal communication
- Got rid of performance issue, received sync-related issue

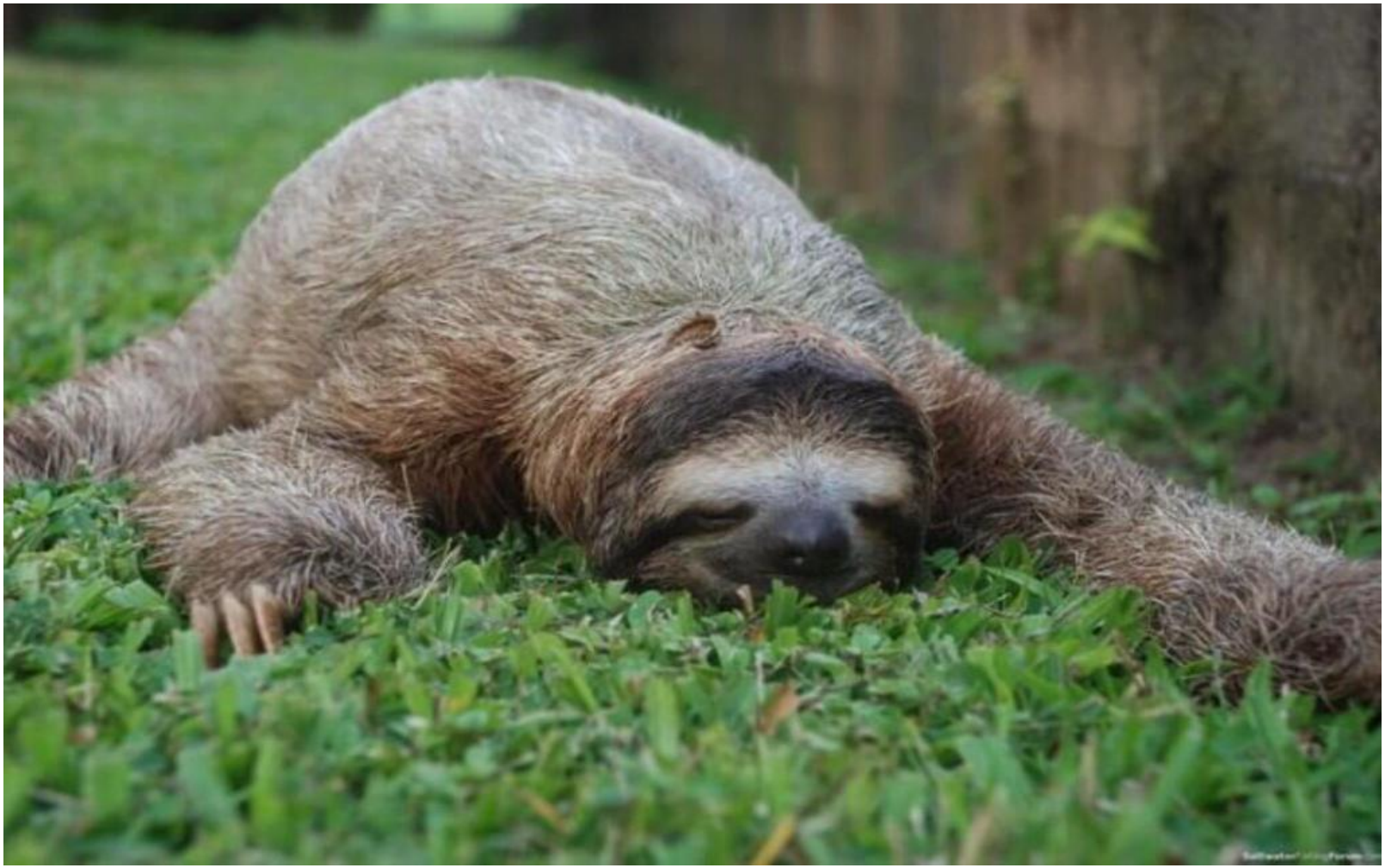


2nd iteration (gRPC + REST)

- We've build gRPC API first, but still duplicate logic for REST part
- No performance issues, but still had some issues with code sync

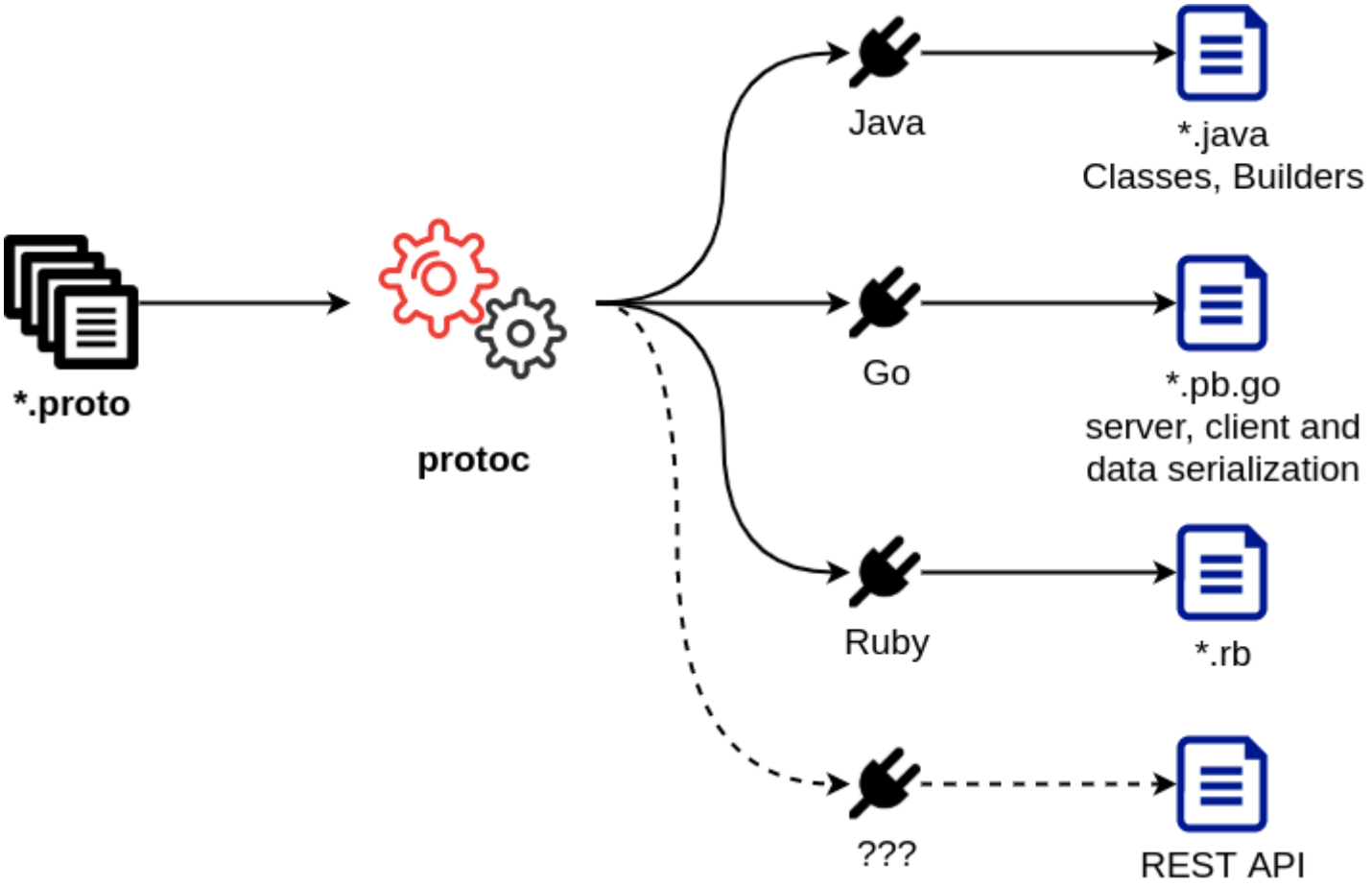


I'm too lazy to support duplications in my code



How we can get rid of it?

Remember this image?



- Can we use any plugin to generate REST?

3rd iteration: yes, we can!

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

```
syntax = "proto3";
package greeter;
option go_package = ".;pb";

import "google/api/annotations.proto";

service Hello {
  rpc SayHello(SayHelloRequest) returns (SayHelloResponse) {
    option (google.api.http) = {
      get: "/hello";
    };
  }
}

message SayHelloRequest { string name = 1; }
message SayHelloResponse { string msg = 1; }
```

```
protoc \
  --go_out=plugins=grpc:. \
  --grpc-gateway_out=:. \
  ./example.proto
```

Generated code: example.pb.gw.go

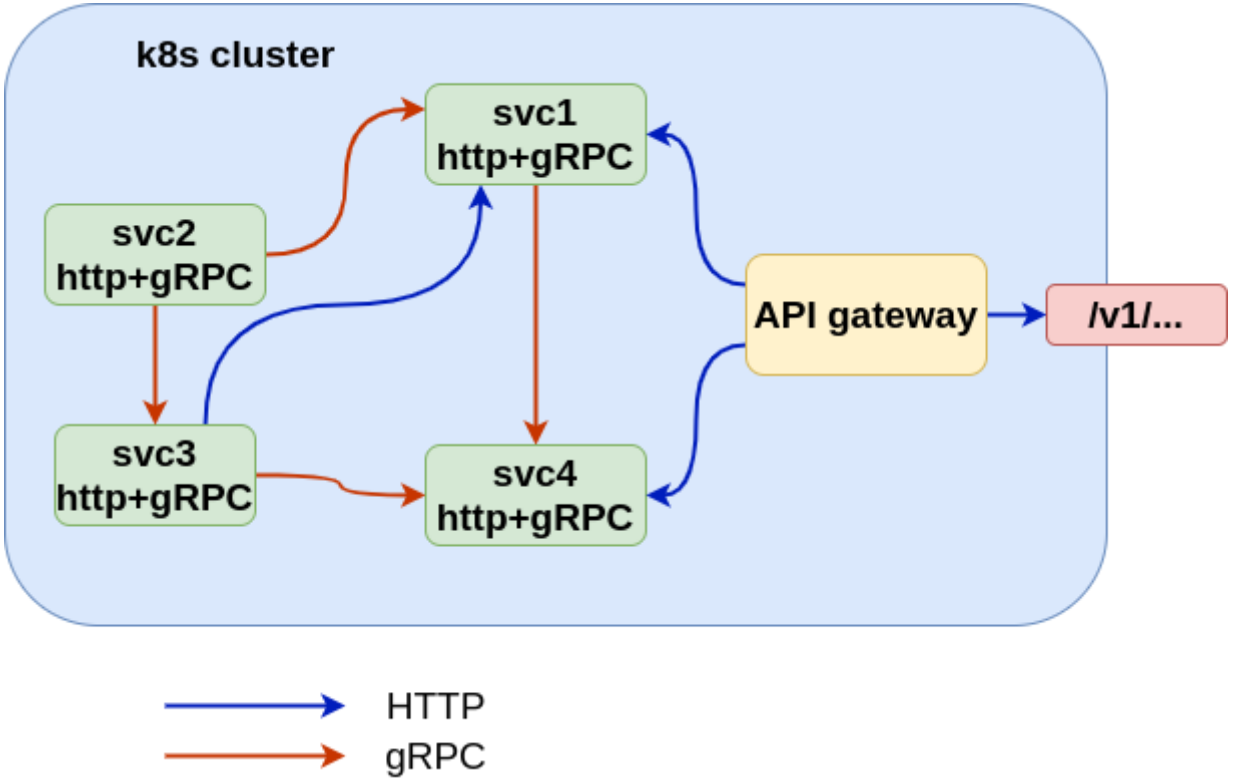
- Contains HTTP reverse proxy.
- Calls gRPC service directly underthehood.

Change main.go a bit

```
type server struct{}

func (s *server) SayHello(ctx context.Context, req *pb.HelloRequest) (*pb.HelloResponse, error) {
    return &pb.HelloResponse{Message: "Hello " + req.Name}
}

func main() {
    lis, err := net.Listen("tcp", ":5001")
    if err != nil {
        log.Fatal(err)
    }

    srv := grpc.NewServer()
    simple.RegisterHelloServer(srv, &server{})
    reflection.Register(srv)
    go srv.Serve(lis)

    mux := runtime.NewServeMux()
    opts := []grpc.DialOption{grpc.WithInsecure()}
    simple.RegisterHelloHandlerFromEndpoint(context.Background(), mux, lis.Addr().String(), opts)
    http.ListenAndServe(":8080", mux)
}
```

But...

- while you have HTTP endpoints inside the cluster, someone will use it again and again



Can we do even better?

4th iteration: Envoy proxy

- That's where we just "turn on" REST API
- No code needed! Config only
- Envoy uses the same annotations as grpc-gateway

```
syntax = "proto3";
package greeter;
option go_package = ".;pb";

import "google/api/annotations.proto";

service Hello {
  rpc SayHello(SayHelloRequest) returns (SayHelloResponse) {
    option (google.api.http) = {
      get: "/hello";
    };
  }
}

message SayHelloRequest { string name = 1; }
message SayHelloResponse { string msg = 1; }
```

main.go

```
type server struct{}

func (s *server) SayHello(ctx context.Context, req *pb.HelloRequest) (*pb.HelloResponse, error) {
    return &pb.HelloResponse{Message: "Hello " + req.Name}
}

func main() {
    lis, err := net.Listen("tcp", ":5001")
    if err != nil {
        log.Fatal(err)
    }

    s := grpc.NewServer()
    pb.RegisterGreeterServer(s, &server{})
    reflection.Register(s)
    s.Server(lis)
}
```

Run protoc

```
protoc \
  -I . \
  -I $GOPATH/pkg/mod/github.com/grpc-ecosystem/grpc-gateway@v1.15.2/third_party/googleapis \
  --go_out=plugins=grpc:. \
  --include_imports --include_source_info \
  --descriptor_set_out=example.pb \
  ./example.proto
```

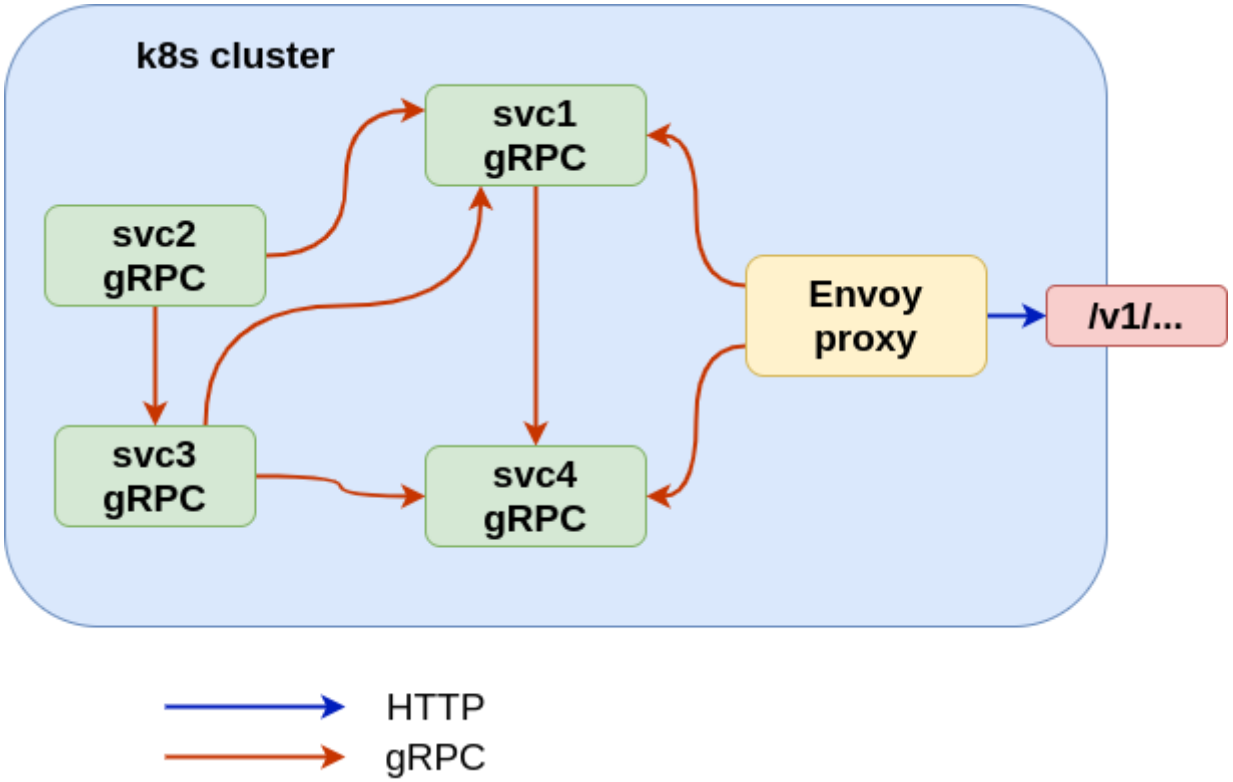
envoy.yaml

```
filter_chains:
  - filters:
      - name: envoy.filters.network.http_connection_manager
        config:
          stat_prefix: grpc_json
          codec_type: auto
          route_config:
            name: local_route
            virtual_hosts:
              - name: local_service
                domains: ["*"]
                routes:
                  - match: { prefix: "/greeter.Hello/", grpc: {}}
                    route: { cluster: hello }
```

envoy.yaml (cont.)

```
http_filters:
  - name: envoy.filters.http.grpc_json_transcoder
    config:
      proto_descriptor: "/etc/example.pb"
      services: ["greeter.Hello"]
      convert_grpc_status: true
      print_options:
        add_whitespace: true
        always_print_primitive_fields: true
        always_print_enums_as_ints: false
        preserve_proto_field_names: true

  - name: envoy.filters.http.router
```

Finally


k8s cluster
svc1
gRPC
svc2
gRPC
Envoy
proxy
/v1/...
svc3
gRPC
svc4
gRPC
HTTP
gRPC

Pros & Cons

- Good enough solution

Pros:

- Automation
- Declarative development.
- Well-defined API interface
- Fast implementation, when you have appropriate toolset.
- Less code.
- Extendable with plugins.

Pros & Cons (cont.)

Cons:

- It's not simple from the beginning
- gRPC is hard to debug in compare with REST.
- Generated code is hard to fix.
- Toolset: different versions. (can be fixed with Bazel)

Tools & libs

- Bazel (https://bazel.build/)
- grpc-web (https://github.com/grpc/grpc-web)
- Custom plugins like github.com/bold-commerce/protoc-gen-struct-transformer (https://github.com/bold-commerce/protoc-gen-struct-transformer)

Thank you

Evgeny Khabarov
PowerFly.Consulting
Ottawa, ON, Canada

twitter: @eekhabarov (#ZgotmplZ)
https://dev.ms/talks (https://dev.ms/talks)
https://powerfly.ca (https://powerfly.ca)