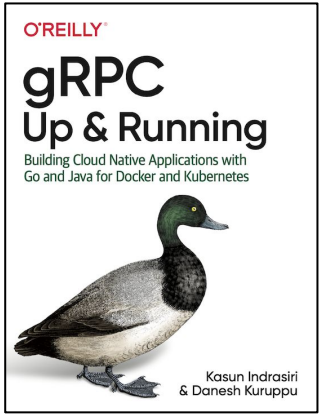
# Cloud Native Communication Patterns with gRPC

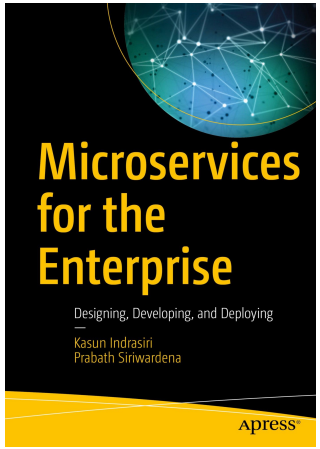
Kasun Indrasiri
Author “gRPC Up and Running” and “Microservices for Enterprise”

GOTOpia
EUROPE 2020
goto; conferences

# About Me

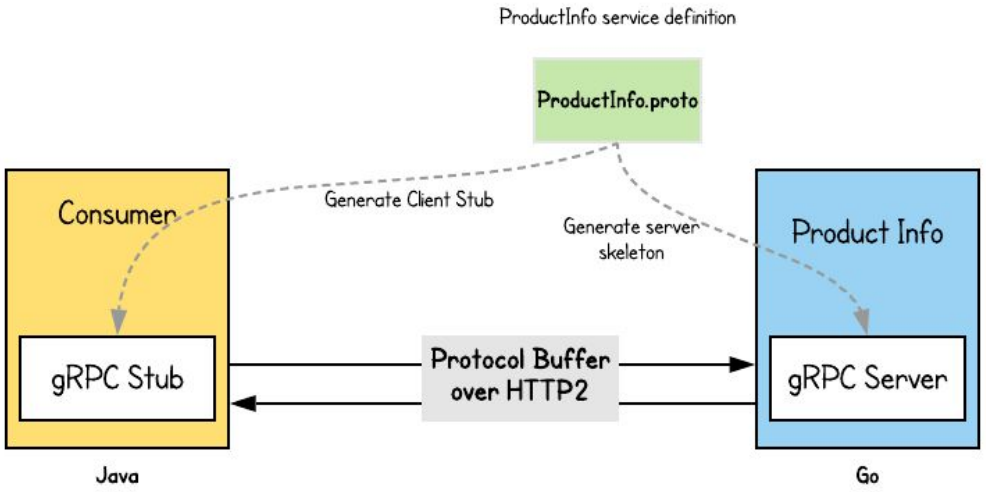
- Author “gRPC Up & Running”, “Microservices for Enterprise”
- Product Manager/Senior Director at WSO2.
- Committer and PMC member at Apache Software Foundation.
- Founder “Bay area Microservices, APIs and Integration” meetup group.



# What is gRPC?

- Modern Inter-process communication technology.
- Invoking remote functions as easy as making a local function invocation.
- Contract-first.
- Binary messaging on the wire on top of HTTP2
- Polyglot.



# Fundamentals of gRPC - Service Definition

- Defines the business capabilities of your service.
- Protocol Buffers used as the IDL for define services.
- Protocol Buffers :
  - A language-agnostic, platform-neutral, extensible mechanism to serializing structured data.
- Defines service, remote methods, and data types.

```
syntax = "proto3";
package ecommerce;
service ProductInfo {
   rpc addProduct(Product) returns (ProductID);
   rpc getProduct(ProductID) returns (Product);
}
message Product {
   string id = 1;
   string name = 2;
   string description = 3;
   float price = 4;
}

message ProductID {
   string value = 1;
}
```

ProductInfo.proto

# Fundamentals of gRPC - gRPC Service

- gRPC service implements the business logic.
- Generate server side skeleton from service definition.
- Implements business logic on top of the generated code.
- Run server application.

```go
// AddProduct implements ecommerce.AddProduct
func (s *server) AddProduct(ctx context.Context,
                            in *pb.Product) (*pb.ProductID,
error) {
   // Business logic
}
// GetProduct implements ecommerce.GetProduct
func (s *server) GetProduct(ctx context.Context, in
*pb.ProductID) (*pb.Product, error) {
   // Business logic
}
// gRPC server
func main() {
   lis, err := net.Listen( "tcp", port)
   ...
   s := grpc.NewServer()
   pb.RegisterProductInfoServer(s, &server{})
   if err := s.Serve(lis); err != nil {
       log.Fatalf( "failed to serve: %v" , err)
   }
```

ProductInfo.go

# Fundamentals of gRPC - gRPC Client

- Connect to a gRPC service and invokes a remote method.
- Generate client-side code(client stub) from service definition and invoke the remote method.

```
ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
ProductInfoGrpc.ProductInfoBlockingStub stub =
            ProductInfoGrpc.newBlockingStub(channel);
        ProductInfoOuterClass.Product product =
                        stub.getProduct(productID);
        logger.info("Product: " + product.toString());
```

ProductInfoClient.java

# Why gRPC?

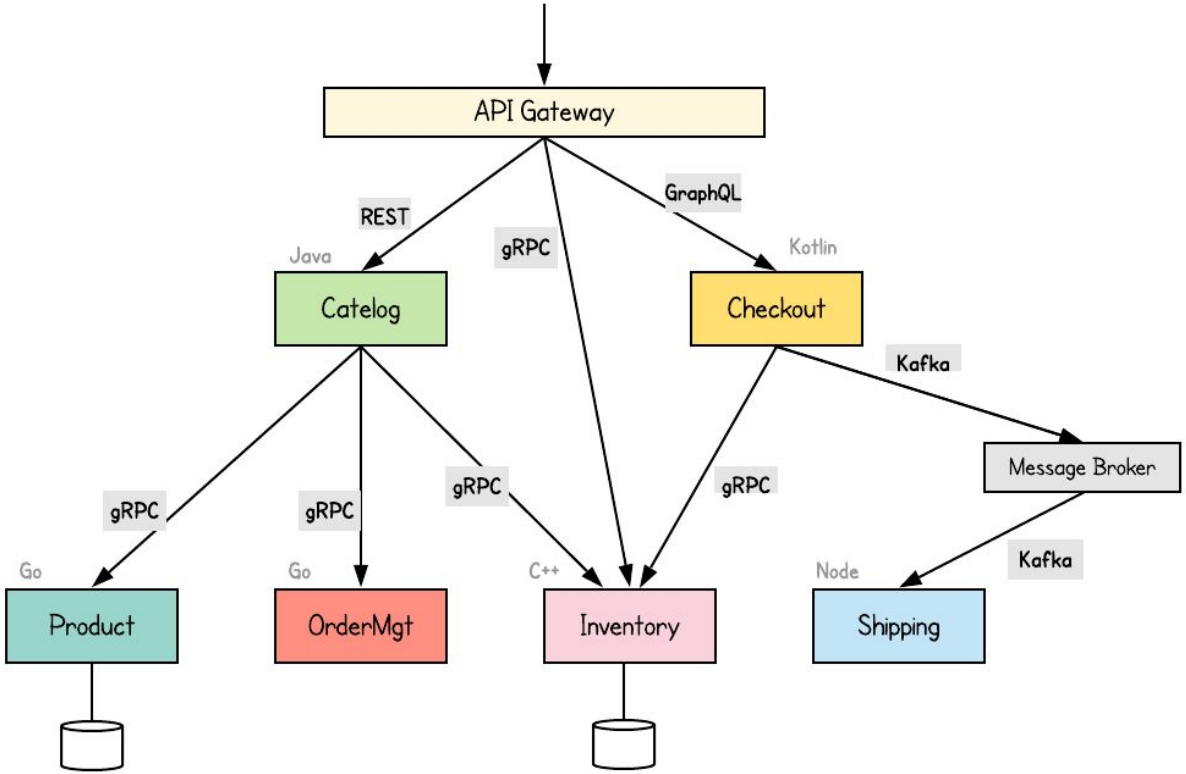
- Efficient.
- Strict specification and well-defined service contracts.
- Strongly Typed.
- Polyglot.
- Duplex Streaming.
- Native integration with cloud native ecosystem.

# gRPC vs OpenAPI/REST vs GraphQL

| | REST/OAS | gRPC | GraphQL |
|---|---|---|---|
| Protocol | HTTP 1.x, HTTP2 | HTTP2 | HTTP 1.x, HTTP2 |
| Payload | Text - JSON, XML<br>(large, human readable) | Binary - Protocol Buffers<br>(small, binary) | Text - JSON |
| Service Definition | OAS (optional) | gRPC IDL/Protocol Buffer(required) | GraphQL (required) |
| Code generation | Optional, via third party libs | Required and natively supported. | Optional |
| Prescriptiveness | Loose, (need to follow best practices) | Strict | Strict |
| Streaming | No first class support | First class support | No first class support |
| Browser support | Yes | No (require grpc-web) | Yes |

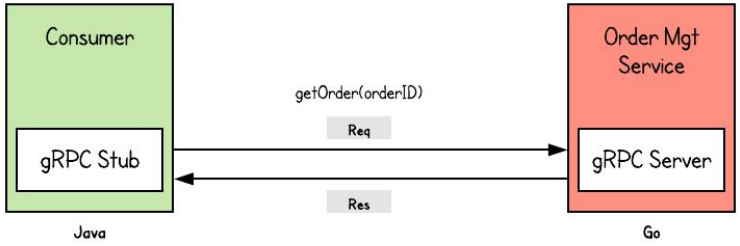
# gRPC in the Microservices Landscape

- Coexistence of  gRPC with other communication protocols.



# Unary/Simple RPC

- Client sends a single request to the server and gets a single response.




order_mgt.proto

```
service OrderManagement {
   rpc getOrder(google.protobuf.StringValue) returns (Order);
}
message Order { ... }
```

Service Impl (Go)

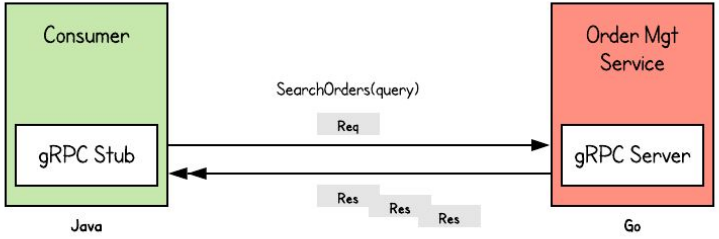
```
func (s *server) GetOrder(ctx context.Context, orderId
*wrapper.StringValue) (*pb.Order, error) {
   ord := orderMap[orderId.Value]
   return &ord, nil
}
```

gRPC Client (Go)

```
   c := pb.NewOrderManagementClient(conn)
   retrievedOrder , err := c.GetOrder(ctx,
                              &wrapper.StringValue{Value: "106"})
```

# Server Streaming RPC

- Server sends back a sequence of responses(stream) after getting the client's request message.
- After sending all the responses server marks the end of stream.




order_mgt.proto

```
service OrderManagement {
   rpc searchOrders(google.protobuf.StringValue) returns
(stream Order);
}
message Order { ... }
```

Service Impl (Go)

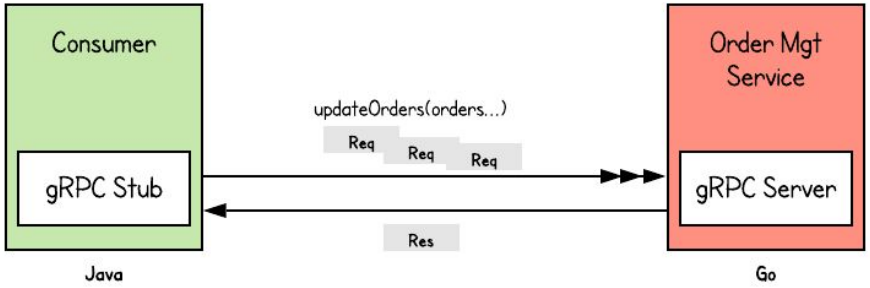
```
func (s *server) SearchOrders(searchQuery
*wrappers.StringValue, stream
pb.OrderManagement_SearchOrdersServer) error {
   // Business logic
  stream.Send(&order1)
  stream.Send(&order2)
  return nil
}
```

gRPC Client (Go)

```
   searchStream, _ := c.SearchOrders(ctx,
                        &wrapper.StringValue{Value: "Google"})
   searchOrder, err := searchStream.Recv()
```

# Client Streaming RPC

- Client sends multiple messages to the server instead of a single request.
- Server sends back a single response to the client.




order_mgt.proto

```
service OrderManagement {
   rpc updateOrders(stream Order) returns
(google.protobuf.StringValue);
}
```

Service Impl (Go)

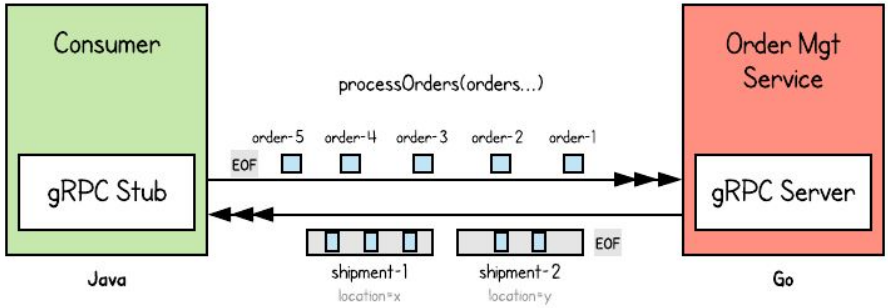
```
func (s *server) UpdateOrders(stream
pb.OrderManagement_UpdateOrdersServer)  error {
   for {
       order, err := stream.Recv()
       if err == io.EOF {
           // Finished reading the order stream.
           return stream.SendAndClose(&wrapper.StringValue{
                       Value:  "Orders processed "  + ordersStr})
       } ...
```

gRPC Client (Go)

```
   updateStream, _ := c.UpdateOrders(ctx)
   _ = updateStream.Send(&updOrder1)
   _ = updateStream.Send(&updOrder2)
   _ = updateStream.Send(&updOrder3)
   updateRes, _ := updateStream.CloseAndRecv()
```

# Bidirectional-Streaming RPC

- Client is sending a request to the server as a stream of messages.
- Server also responds with a stream of messages.
- Client has to initiated the RPC.




order_mgt.proto

```
  rpc processOrders(stream google.protobuf.StringValue)
returns (stream CombinedShipment);
```

Service Impl (Go)

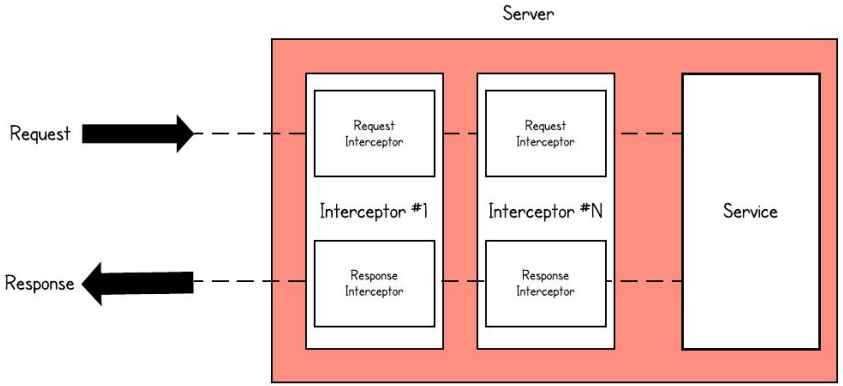
```
func (s *server) ProcessOrders(stream
pb.OrderManagement_ProcessOrdersServer) error {
...
for {
      orderId, err := stream.Recv()
            for _, comb := range combinedShipmentMap {
             stream.Send(&comb)
          } ...
```

gRPC Client (Go)

```
  streamProcOrder, _ := c.ProcessOrders(ctx)
  _ = streamProcOrder.Send(&wrapper.StringValue{Value:"102"})
  _ = streamProcOrder.Send(&wrapper.StringValue{Value:"103"})
...

  channel := make(chan bool, 1)
  go asncClientBidirectionalRPC(streamProcOrder, channel)
...
```

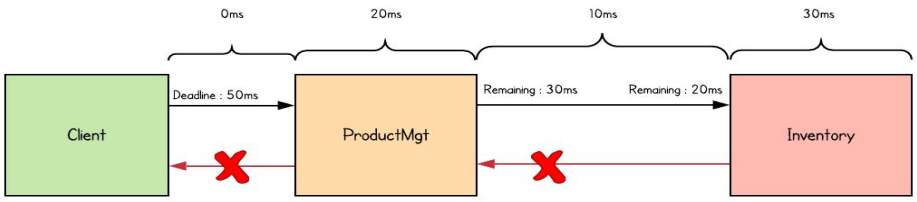
# gRPC Interceptors

- Mechanism to execute some common logic before or after the execution of the remote function for server or client application.
- Server Side and Client Side interceptors.
- Unary Interceptors
  - Phases : preprocessing, invoking the RPC method, and postprocessing
- Streaming interceptors
  - Intercepts any streaming RPC
- Useful for logging, authentication, metrics etc.



# Deadlines

- A deadline is expressed in absolute time from the beginning of a request and applied across multiple service invocations.
- gRPC client applications sets deadline when invoking remote functions.

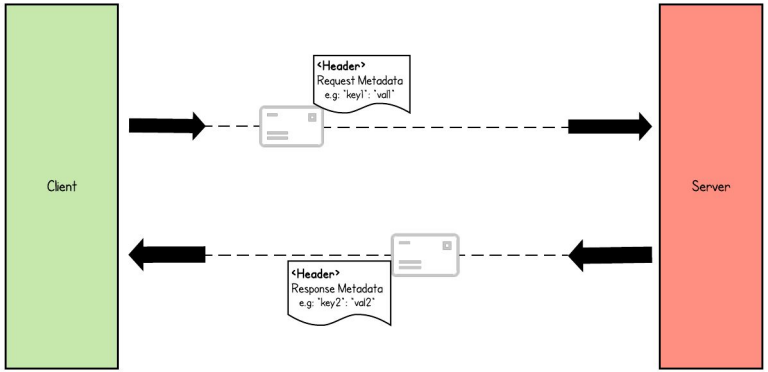



gRPC Client App

```
   clientDeadline := time.Now().Add(time.Duration(2 *
time.Second))
   ctx, cancel := context.WithDeadline(context.Background(),
clientDeadline)
// Invoke RPC
```

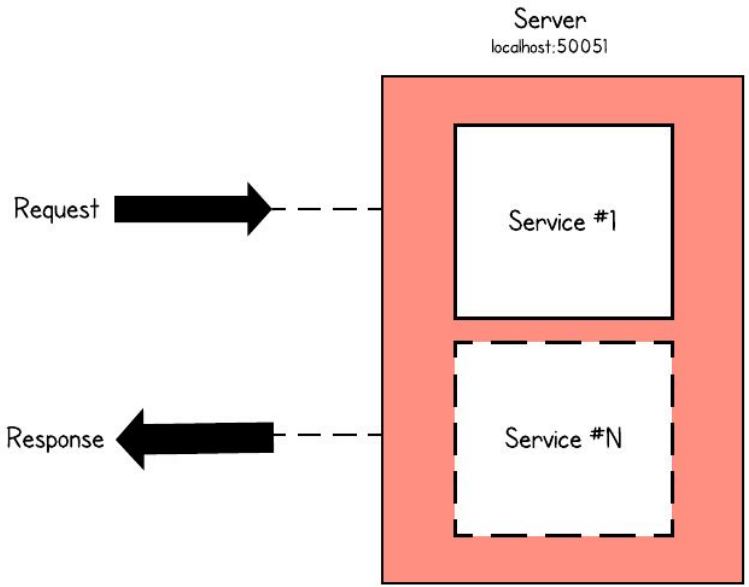
# Metadata

- Information directly related to the service's business logic and consumer is part of the remote method invocation arguments.
- Use Metadata to share information about the RPC calls that are not related to the business context of the RPC (e.g. Security Headers)
- Structured as K-V pairs.
- Exchanged as gRPC headers.



# Multiplexing

- Running multiple gRPC services on the same gRPC server.




gRPC Server App

```
grpcServer := grpc.NewServer()

// Register Order Management service on gRPC orderMgtServer
ordermgt_pb.RegisterOrderManagementServer(grpcServer,
                                          &orderMgtServer{})

// Register Greeter Service on gRPC orderMgtServer
hello_pb.RegisterGreeterServer(grpcServer, &helloServer{})
```

# Cancellation

- When either the client or server application wants to terminate the RPC this can be done by *Cancelling* the RPC.
- No further RPCs can be done over that connection.
- When one party cancels the RPC, the other party can determine it by checking the context of the RPC.
  - E.g. `stream.Context().Err() == context.Canceled`.

# API Specification with gRPC

- Default gRPC service definitions caters to:
  - Service, Remote methods and Types.
- Advanced API specification options for:
  - Authentication schemes are JWT, Basic Auth and API Key
  - Authorization
    - Granular authorization at method level.
  - Rate limiting/Throttling
    - Applied for per-service or per-method basis.
  - Versioning
  - Policy Enforcement

# API Versioning with gRPC

- Services should strive to remain backwards compatible with old clients.
- Service versioning allows us to introduce breaking changes to the gRPC service.
- gRPC package → specify a version number for your service and its messages.

order_mgt.proto

```
syntax = "proto3";
package ecommerce.v1;
service OrderManagement {
   rpc addOrder(Order) returns
(google.protobuf.StringValue);
   rpc getProduct(google.protobuf.StringValue) returns
(Order);
}
```

```
:method POST
:path   /<package_name>.<service_name>/<method_name>


E.g: AddOrder Remote Call:

:method POST
:path   /ecommerce.v1.OrderManagement>/addOrder
```

# Extending Service Definition

- Service level, method level and field level options in service definition.
- Access those options at runtime/implementation.

Service Options

```
import "google/protobuf/descriptor.proto" ;

// custom service options
extend google.protobuf.ServiceOptions {
 string oauth2Provider = 50003;
}

service OrderManagement {
 option(oauth2Provider) =
"https://localhost:9444/oauth2/introspect";
}
```

Method Options

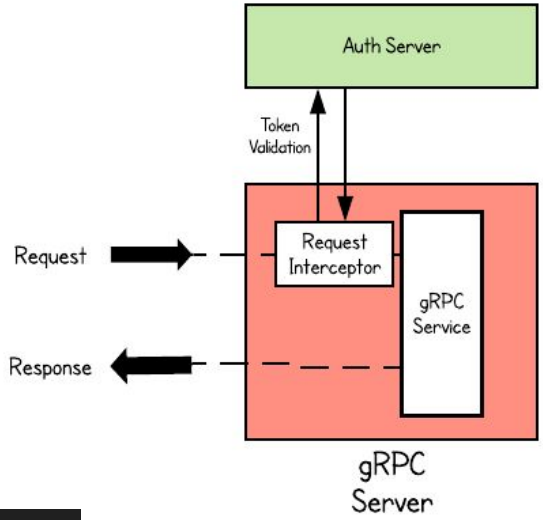
```
import "google/protobuf/descriptor.proto" ;

// custom method options
extend google.protobuf.MethodOptions {
 int32 throttling_tier_per_min = 50001;
}
service OrderManagement {
 rpc addOrder(Order) returns (google.protobuf.StringValue) {
   option(throttling_tier_per_min) = 10000;
 }
}
```

Field Options

```
import "google/protobuf/descriptor.proto" ;

// custom field options
extend google.protobuf.FieldOptions {
 bool sensitive = 50000;
}

message Order {
 string id = 1;
 string destination = 5 [(ecommerce.sensitive) = true];
}
```

# Securing gRPC services with OAuth2

- Example use case
  - Order management service secured with OAuth2
  - Service definition contains OAuth2 provider endpoint and other metadata
  - Interceptor access the metadata

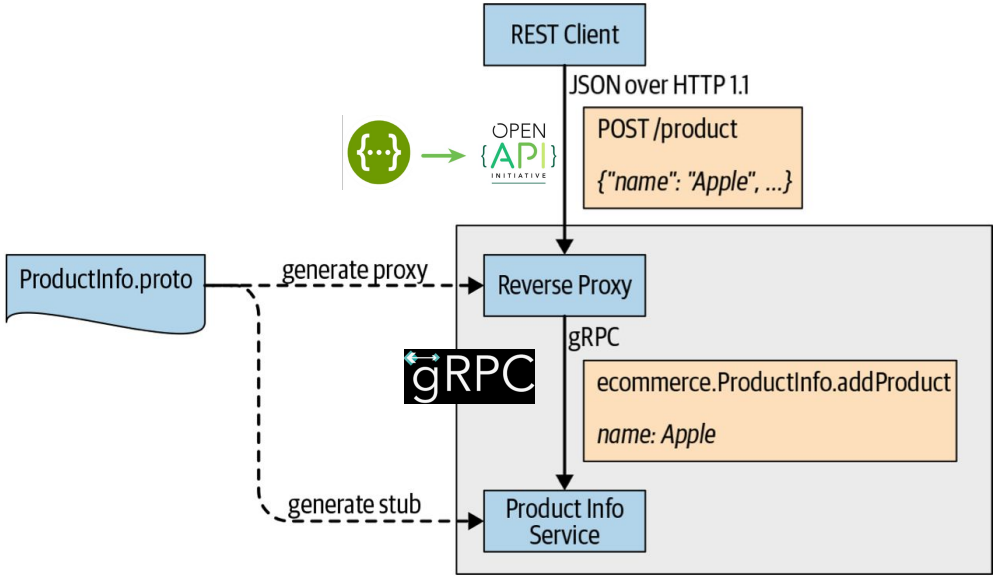



```
import "google/protobuf/descriptor.proto" ;

// custom service options
extend google.protobuf.ServiceOptions {
 string oauth2Provider = 50003;
}

service OrderManagement {
 option(oauth2Provider) = "https: //localhost:9444/oauth2/introspect";
}
```

# REST/Open API ↔ gRPC Bridge

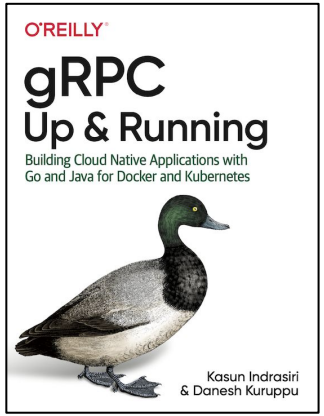
- gRPC Gateway : REST/HTTP 1.1 -> gRPC Bridge.
- Also known as HTTP/JSON Transcoding for gRPC
- gRPC gateway plug-in enables the protocol buffer compiler to read the gRPC service definition and generate a reverse proxy server, which translates a RESTful JSON API into gRPC.

```
import "google/api/annotations.proto" ;

service OrderManagement {
 rpc addOrder(Order) returns (google.protobuf.StringValue)
{
   option (google.api.http) = {
     post: "/v1/order"
     body: "*"
   };
 }
}
```



# Resources

- gRPC Up and Running Book.
  - Gives you a comprehensive understanding of gRPC.
  - gRPC communication patterns and advanced concepts.
  - Running gRPC in production.
  - Dozens of samples written in Go and Java.
- Use cases and source code in Java and Go - *https://grpc-up-and-running.github.io/*
- grpc.io



# Thank You