gRPC and Go: Developing Efficient and Type-Safe Services

Vladimir vivien
Vladimir Vivien  →   Clint Kitson (@clintkitson)

Open Source Engineer (we’re hiring)

Vlad is a Go book author

@VladimirVivien
Objectives

Why gRPC

A gRPC Service in Go

gRPC Feature Walkthrough
Why gRPC?
Minimize Complexity

All we want is interoperability between components. It is the components that provide the value and not how they communicate. Make it simple and efficient!
The Situation Today

- Mobile technology is driving demand
  - Multiple languages and platforms
  - Efficiency
    - ie. efficiencies in sending media turns into happier users and massive data center savings

- Software architectures evolving
  - Cloud native, componentization, and interoperability
  - Micro-services and containers
    - ie. Scaling, failures, and continuous delivery
Scenario - A Currency Service
Scenario - A Currency Service
Scenario - A Currency Service
Scenario - A Currency Service

Currency Service

Currency Lookup

Mobile Android
Android Java

Mobile iPhone
Objective-C
Scenario - A Currency Service

Currency Service

Currency Lookup

Mobile Android
Android Java

Mobile iPhone
Objective-C

Backend Processes
Java, Python
Scenario - A Currency Service

Currency Service

Currency Lookup

- Mobile Android
  - Android Java
- Mobile iPhone
  - Objective-C
- Backend Processes
  - Java, Python
- Desktop Report tool
  - C#, Java
Scenario - A Currency Service

Currency Service

Currency Lookup

- Mobile Android: Android Java
- Mobile iPhone: Objective-C
- Backend Processes: Java, Python
- Desktop Report tool: C#, Java
- Website backend: NodeJs
The Problem

- Mobile Android: Android Java
- Mobile iPhone: Objective-C
- Backend Processes: Java, Python
- Desktop Report tool: C#, Java
- Website backend: NodeJs
The Problem

Multiple business systems

Mobile
- Android
  - Android Java
- iPhone
  - Objective-C
Backend Processes
  - Java, Python
Desktop Report tool
  - C#, Java
Website backend
  - NodeJs
The Problem

Multiple business systems

Bandwidth-constrained access (mobile)
The Problem

Multiple business systems

Bandwidth-constrained access (mobile)

Real-time data access
The Problem

Multiple business systems

Bandwidth-constrained access (mobile)

Real-time data access

Different platforms and languages
The Problem

Multiple business systems

Bandwidth-constrained access (mobile)

Real-time data access

Different platforms and languages

Need to feature-scale with versioning

Mobile Android      Android Java
Mobile iPhone       Objective-C
Backend Processes   Java, Python
Desktop Report tool C#, Java
Website backend     NodeJs
The Problem

Multiple business systems
Bandwidth-constrained access (mobile)
Real-time data access
Different platforms and languages
Need to feature-scale with versioning
Maintain backward compatibility
RPC over HTTP/JSON
RPC over HTTP/JSON

It’s a good solution. RPC, HTTP and JSON makes a simple, flexible, and universally accepted build your own approach.
RPC over HTTP/JSON

It’s a good solution. RPC, HTTP and JSON makes a simple, flexible, and universally accepted build your own approach.

However:

- JSON has weak data typing
- Inefficient text-based encoding
- Services are simple state transfers via JSON docs
- Business semantics do not map to HTTP verbs
- Versioning, update, and backward compatibility are problematic
- Clients are likely implemented manually and inconsistently
REST

It sounds like a good solution, but it works via frameworks and implementations on top of some flaws of using RPC over HTTP/JSON. This is difficult to do right.
REST

It sounds like a good solution, but it works via frameworks and implementations on top of some flaws of using RPC over HTTP/JSON. This is difficult to do right.

Rooted in a concept called a hypermedia API

- Can we create a shared common way that developers can communicate
- Point any language at an endpoint, and that language naturally knows how to get something done for you (self-discoverable)
Hypermedia - Think of it this way

As a human, I understand how to make use of this interface. But that is learned, so how can computers learn how to make use of something through an interface?

You teach it how to navigate a maze. A simple pattern, the maze: Left, Right, Enter, Up, Down, Execute.
A Self-Discoverable REST API

Get ready to read some books..

Leads to frameworks like Swagger and Apiary that support IDLs for defining interfaces
Forget about REST.. mostly

I’ve met a number of organisations who have done just this … and they are now struggling for obvious reasons.

And I’ve heard advice such as “break your monolith up into microservices, with REST APIs between them” a number of times this week. 😞
## Attempts at solving this problem

<table>
<thead>
<tr>
<th>Method</th>
<th>Idiomatic Client/Server</th>
<th>Curl Friendly</th>
<th>Framework</th>
<th>Serialization</th>
<th>Efficient</th>
<th>Predictable</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAP</td>
<td>via Frameworks</td>
<td>No</td>
<td>By language</td>
<td>XML</td>
<td>No</td>
<td>Somewhat</td>
</tr>
<tr>
<td>RPC over HTTP/JSON</td>
<td>No</td>
<td>Yes</td>
<td>Custom</td>
<td>none</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Custom REST</td>
<td>No</td>
<td>Yes</td>
<td>Custom</td>
<td>*</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>REST</td>
<td>via Frameworks</td>
<td>Yes</td>
<td>Swagger, others</td>
<td>Likely JSON</td>
<td>No</td>
<td>Somewhat</td>
</tr>
<tr>
<td>gRPC</td>
<td>Yes</td>
<td>No</td>
<td>Included</td>
<td>Binary</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The Linux philosophy applied..
- Write focused tools that do things well and interoperate

Is there a “Linux pipe for boring distributed components?”
A universal open-source RPC framework designed to **create efficient and fast polyglot services** with usage ranging from **datacenter scale computing** to **bandwidth-constrained devices**.
GRPC

Uses Protocol Buffers (serialize structured data)

Efficient and typed binary format

Simple interface definition language (IDL)

Uses HTTP/2 for secure and fast multiplexed transport

Bi-directional and streaming between clients and servers

Extensible middleware API for authentication, authorization, tracing, service policy
Generates idiomatic client stubs and server interfaces in 10 languages:
Performance Examples - Dashboard

Streaming secure ping pong median latency (in microsec)

Language comparison of streaming ping pong between two GCE VMs (in the same zone). Secure connection is used.
Benchmark results

Tests send 300,000 requests to key/value stores. One with `jsonrpc`, the other with `gRPC`. Both `jsonrpc` and `gRPC` code use only one TCP connection. And another `gRPC` case with one TCP connection but with multiple clients:

<table>
<thead>
<tr>
<th>RPC</th>
<th># of requests</th>
<th># of clients</th>
<th>total time</th>
<th>per-request time</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsonrpc</td>
<td>300,000</td>
<td>1</td>
<td>8m7.270s</td>
<td>1.624ms</td>
</tr>
<tr>
<td>gRPC</td>
<td>300,000</td>
<td>1</td>
<td>36.715s</td>
<td>122.383μs</td>
</tr>
<tr>
<td>gRPC</td>
<td>300,000</td>
<td>100</td>
<td>7.167s</td>
<td>23.892μs</td>
</tr>
</tbody>
</table>

And if compared on memory usage:

<table>
<thead>
<tr>
<th>RPC</th>
<th>jsonrpc</th>
<th>gRPC</th>
<th>delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>NsPerOp</td>
<td>487271046903</td>
<td>36716116701</td>
<td>-92.46%</td>
</tr>
<tr>
<td>AllocsPerOp</td>
<td>32747687</td>
<td>25221256</td>
<td>-22.98%</td>
</tr>
<tr>
<td>AllocatedBytesPerOp</td>
<td>3182814152</td>
<td>1795122672</td>
<td>-43.60%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RPC</th>
<th>jsonrpc</th>
<th>gRPC with 100 clients</th>
<th>delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>NsPerOp</td>
<td>487271046903</td>
<td>7168591678</td>
<td>-98.53%</td>
</tr>
<tr>
<td>AllocsPerOp</td>
<td>32747687</td>
<td>25230286</td>
<td>-22.96%</td>
</tr>
<tr>
<td>AllocatedBytesPerOp</td>
<td>3182814152</td>
<td>1795831944</td>
<td>-43.58%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RPC</th>
<th>gRPC</th>
<th>gRPC with 100 clients</th>
<th>delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>NsPerOp</td>
<td>36716116701</td>
<td>7168591678</td>
<td>-80.48%</td>
</tr>
<tr>
<td>AllocsPerOp</td>
<td>25221256</td>
<td>25230286</td>
<td>+0.04%</td>
</tr>
<tr>
<td>AllocatedBytesPerOp</td>
<td>1795122672</td>
<td>1795831944</td>
<td>+0.04%</td>
</tr>
</tbody>
</table>
Why is it faster?

A little known fact is that approximately 47% of CPU usage across a typical Kubernetes cluster is invested translating between JSON and YAML.

2:28 PM - 17 Aug 2017
A gRPC Service in Go
Let us walk through the creation of the currency service mentioned earlier.

Source available: https://github.com/vladimirvivien/go-grpc
Creating a gRPC Service

1) Define service contract in Protocol Buffers IDL
2) Compile IDL into service interfaces source code
3) Implement service methods to satisfy contract
IDL for CurrencyService
IDL for CurrencyService

currency.proto

```protobuf
syntax = "proto3";
package protobuf;

message Currency {
  string code = 1;
  string name = 2;
  int32 number = 3;
  string country = 4;
}

message CurrencyList {
  repeated Currency items = 1;
}

message CurrencyRequest {
  string code = 1;
  int32 number = 2;
}

service CurrencyService {
  rpc GetCurrencyList(CurrencyRequest) returns (CurrencyList);
}
```
Protocol Buffers version 3

syntax = "proto3";
package protobuf;

message Currency {
  string code = 1;
  string name = 2;
  int32 number = 3;
  string country = 4;
}

message CurrencyList {
  repeated Currency items = 1;
}

message CurrencyRequest {
  string code = 1;
  int32 number = 2;
}

service CurrencyService {
  rpc GetCurrencyList(CurrencyRequest) returns (CurrencyList){}
}
IDL for CurrencyService

Protocol Buffers version 3

Messages represent data structures used in the service methods.

```proto3
currency.proto

syntax = "proto3";
package protobuf;

message Currency {
  string code = 1;
  string name = 2;
  int32 number = 3;
  string country = 4;
}

message CurrencyList {
  repeated Currency items = 1;
}

message CurrencyRequest {
  string code = 1;
  int32 number = 2;
}

service CurrencyService {
  rpc GetCurrencyList(CurrencyRequest) returns (CurrencyList){}
}
Protocol Buffers version 3

Messages represent data structures used in the service methods.

Field-level versioning
IDL for CurrencyService

Protocol Buffers version 3

Messages represent data structures used in the service methods.

Field-level versioning

The service block declares the remote methods, with the `rpc` keyword, that make up the service.

GetCurrencyList is a unary method: it sends a single request and receives a single response.

```
syntax = "proto3";
package protobuf;

message Currency {
  string code = 1;
  string name = 2;
  int32 number = 3;
  string country = 4;
}

message CurrencyList {
  repeated Currency items = 1;
}

message CurrencyRequest {
  string code = 1;
  int32 number = 2;
}

service CurrencyService {
  rpc GetCurrencyList(CurrencyRequest) returns (CurrencyList);
}
```
Compile IDL into Code
Compile IDL into Code

currency.proto

```proto3
syntax = "proto3";
package protobuf;

message Currency {
  string code = 1;
  string name = 2;
  int32 number = 3;
  string country = 4;
}

message CurrencyList {
  repeated Currency items = 1;
}

message CurrencyRequest {
  string code = 1;
  int32 number = 2;
}

service CurrencyService {
  rpc GetCurrencyList(CurrencyRequest) returns (CurrencyList){
```
Compile IDL into Code

currency.proto

```protobuf
syntax = "proto3";
package protobuf;

message Currency {
  string code = 1;
  string name = 2;
  int32 number = 3;
  string country = 4;
}

message CurrencyList {
  repeated Currency items = 1;
}

message CurrencyRequest {
  string code = 1;
  int32 number = 2;
}

service CurrencyService {
  rpc GetCurrencyList(CurrencyRequest) returns (CurrencyList);
}
```
Compile IDL into Code (Go language example)

```
syntax = "proto3";
package protobuf;

message Currency {
  string code = 1;
  string name = 2;
  int32 number = 3;
  string country = 4;
}

message CurrencyList {
  repeated Currency items = 1;
}

message CurrencyRequest {
  string code = 1;
  int32 number = 2;
}

service CurrencyService {
  rpc GetCurrencyList(CurrencyRequest) returns (CurrencyList);
}
```

```
package protobuf

type Currency struct {
  Code string `protobuf:"bytes,1,opt,name=code" json:"code,omitempty"`
  Name string `protobuf:"bytes,2,opt,name=name" json:"name,omitempty"`
  Number int32 `protobuf:"varint,3,opt,name=number" json:"number,omitempty"`
  Country string `protobuf:"bytes,4,opt,name=country" json:"country,omitempty"

  ...
}

type CurrencyList struct {
  Items []*Currency `protobuf:"bytes,1,rep,name=items" json:"items,omitempty"
}

...

type CurrencyRequest struct {
  Code string `protobuf:"bytes,1,opt,name=code" json:"code,omitempty"
  Number int32 `protobuf:"varint,2,opt,name=number" json:"number,omitempty"

  ...
}

type CurrencyServiceServer interface {
  GetCurrencyList(context.Context, *CurrencyRequest) (*CurrencyList, error)
}
```

`currency.proto`

`currency.pb.go`
Implement Service Methods
Implement Service Methods (Go language example)

type CurrencyService struct {
    data []*pb.Currency
}

func (c *CurrencyService) GetCurrencyList(
    ctx context.Context,
    req *pb.CurrencyRequest,
) (*pb.CurrencyList, error) {
    var items []*pb.Currency
    for _, cur := range c.data {
            items = append(items, cur)
        }
    }
    return &pb.CurrencyList{Items: items}, nil
Implement Service Methods (Go language example)

```go
type CurrencyService struct {
    data []*pb.Currency
}

func (c *CurrencyService) GetCurrencyList(ctx context.Context,
    req *pb.CurrencyRequest,
) (*pb.CurrencyList, error) {
    var items []*pb.Currency
    for _, cur := range c.data {
        if cur.GetNumber() == req.GetNumber() ||
            items = append(items, cur)
    }
    return &pb.CurrencyList{Items: items}, nil
}

func main() {
    data := util.LoadPbFromCsv("./../curdata.csv")
    lstnr, err := net.Listen("tcp", ":50050")
    if err != nil {
        log.Fatal("failed to start server!", err)
    }
    curService := newCurrencyService(data)
    grpcServer := grpc.NewServer()
    pb.RegisterCurrencyServiceServer(grpcServer, curService)
    log.Println("starting currency rpc service on", port)
    if err := grpcServer.Serve(lstnr); err != nil {
        log.Fatal(err)
    }
}
```
Calling Service from Client
Calling Service from Client

```go
import pb "github.com/vladimirvivien/go-grpc/protobuf"

func main() {
    conn, err := grpc.Dial("localhost:50050", grpc.WithInsecure())
    if err != nil {
        log.Fatal(err)
    }

    client := pb.NewCurrencyServiceClient(conn)

    printUSD(client)
}

func printUSD(client pb.CurrencyServiceClient) {
    curReq := &pb.CurrencyRequest{Code: "USD"}
    curList, err := client.GetCurrencyList(context.Background(), curReq)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("\nUSD Countries")
    fmt.Println("----------------")
    for _, cur := range curList.Items {
        fmt.Printf("%-50s%-10s\n", cur.GetCountry(), cur.GetCode())
    }
}```
Calling Service from Client

```go
import pb "github.com/vladimirvivien/go-grpc/protobuf"

func main() {
    conn, err := grpc.Dial("localhost:50050", grpc.WithInsecure())
    if err != nil {
        log.Fatal(err)
    }

    client := pb.NewCurrencyServiceClient(conn)

    printUSD(client)
}

func printUSD(client pb.CurrencyServiceClient) {
    curReq := &pb.CurrencyRequest{Code: "USD"}
    curList, err := client.GetCurrencyList(context.Background(), curReq)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("USD Countries")
    fmt.Println("-------------")
    for _, cur := range curList.Items {
        fmt.Printf("%-50s%-10s\n", cur.GetCountry(), cur.GetCode())
    }
}
```
Calling Service from Client

```go
import pb "github.com/vladimirvivien/go-grpc/protobuf"

func main() {
    conn, err := grpc.Dial("localhost:50050", grpc.WithInsecure())
    if err != nil {
        log.Fatal(err)
    }

    client := pb.NewCurrencyServiceClient(conn)

    printUSD(client)
}

func printUSD(client pb.CurrencyServiceClient) {
    curReq := &pb.CurrencyRequest{Code: "USD"}
    curList, err := client.GetCurrencyList(context.Background(), curReq)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("\nUSD Countries")
    fmt.Println("---")
    for _, cur := range curList.Items {
        fmt.Printf("%50s%10s\n", cur.GetCountry(), cur.GetCode())
    }
}
```
gRPC Features

A walkthrough of the many features of gRPC
gRPC supports streaming where client can stream data to server, server to client and bi-directionally.
Streaming

Large requests can be streamed to server

Server can stream large responses to client

Strategy to keep client or server from running out of memory

Easy to declare in the IDL

Framework provides API support to use stream
Update IDL for Streaming

Streaming declared in IDL with stream keyword
Update IDL for Streaming

Streaming declared in IDL with `stream` keyword

```idl
service CurrencyService {
    rpc GetCurrencyList(CurrencyRequest) returns (CurrencyList){}
    rpc GetCurrencyStream(CurrencyRequest) returns (stream Currency){}
    rpc SaveCurrencyStream(stream Currency) returns (CurrencyList){}
    rpc FindCurrencyStream(stream CurrencyRequest) returns (stream Currency){}
}
```
Update IDL with Streaming

Streaming declared in IDL with `stream` keyword
Update IDL with Streaming

Streaming declared in IDL with `stream` keyword

```idl
service CurrencyService {
    rpc GetCurrencyList(CurrencyRequest) returns (CurrencyList){}
    rpc GetCurrencyStream(CurrencyRequest) returns (stream Currency){}
    rpc SaveCurrencyStream(stream Currency) returns (CurrencyList){}
    rpc FindCurrencyStream(stream CurrencyRequest) returns (stream Currency){}
}
```
A Server Stream Example - Server

Server streaming RPCs where the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages.
A Server Stream Example - Server

```go
func (c *CurrencyService) GetCurrencyStream(
    req *pb.CurrencyRequest,
    stream pb.CurrencyService_GetCurrencyStreamServer,
) error {

    for _, cur := range c.data {
            if err := stream.Send(cur); err != nil {
                return err
            }
        }
    }

    return nil
}
```
A Server Stream Example - Server

```go
func (c *CurrencyService) GetCurrencyStream(
    req *pb.CurrencyRequest,
    stream pb.CurrencyService_GetCurrencyStreamServer,
) error {

    for _, cur := range c.data {
            if err := stream.Send(cur); err != nil {
                return err
            }
        }
    }

    return nil
}
```

Server sends data elements to client via a stream server.
A Server Stream Example - Client

```go
func printEUR(client pb.CurrencyServiceClient) {
    curReq := &pb.CurrencyRequest{Code: "EUR"}
    stream, err := client.GetCurrencyStream(context.Background(), curReq)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("\nEUR Countries")
    fmt.Println("-----------")
    for {
        cur, err := stream.Recv()
        if err != nil {
            if err == io.EOF {
                break
            }
            fmt.Println(err)
            continue
        }
        fmt.Printf("%-50s%-10s\n", cur.GetCountry(), cur.GetCode())
    }
}
```
A Server Stream Example - Client

```go
func printEUR(client pb.CurrencyServiceClient) {
    curReq := &pb.CurrencyRequest{Code: "EUR"}
    stream, err := client.GetCurrencyStream(context.Background(), curReq)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("\nEUR Countries")
    fmt.Println("-------------")
    for {
        cur, err := stream.Recv()
        if err != nil {
            if err == io.EOF {
                break
            }
            fmt.Println(err)
            continue
        }
        fmt.Printf("%-50s%-10s\n", cur.GetCountry(), cur.GetCode())
    }
}
```

Client streams data elements from the server until io.EOF.
Secure gRPC Service

The gRPC framework makes it easy to add secure TLS transport between client and servers.
TLS Setup - Server
TLS Setup - Server

```go
func main() {
    lstnr, err := net.Listen("tcp", ":50050")
    if err != nil {
        log.Fatal("failed to start server:", err)
    }

    tlsCreds, err := credentials.NewServerTLSFromFile(srvCertFile, srvKeyFile)
    if err != nil {
        log.Fatal(err)
    }

    // setup and register currency service
    curService := newCurrencyService(data)
    grpcServer := grpc.NewServer(grpc.Creds(tlsCreds))
    pb.RegisterCurrencyServiceServer(grpcServer, curService)

    log.Println("starting secure currency rpc service on ", port)
    if err := grpcServer.Serve(lstnr); err != nil {
        log.Fatal(err)
    }
}
```
Create TLS creds from cert and key files.

server.go
TLS Setup - Client
TLS Setup - Client

```go
func main() {
    serverAddr := net.JoinHostPort(server, serverPort)

    tlsCreds, err := credentials.NewClientTLSFromFile(certFile, "")
    if err != nil {
        log.Fatal(err)
    }

    conn, err := grpc.Dial(serverAddr, grpc.WithTransportCredentials(tlsCreds))
    if err != nil {
        log.Fatal(err)
    }

    client := pb.NewCurrencyServiceClient(conn)
    ...
}
```

Setup client-side TLS credentials and use it to dial the server.
Request Timeout

Use gRPC request timeouts to prevent client from waiting an excessively long time for responses.
Request Timeouts

Specify timeouts using Go’s `Context` type
Request Timeouts

Specify timeouts using Go's `Context` type

```go
func printUSD(client pb.CurrencyServiceClient) {
    ctx, cancel := context.WithTimeout(context.Background(), 200*time.Millisecond)
    defer cancel()

    curReq := &pb.CurrencyRequest{Code: "USD"} 
    curList, err := client.GetCurrencyList(ctx, curReq) 
    if err != nil {
        fmt.Println("error in printUSD:", err)
        return
    }

    fmt.Println("\nUSD Countries")
    fmt.Println("----------")
    for _, cur := range curList.Items {
        fmt.Printf("%-50s%-10s\n", cur.GetCountry(), cur.GetCode())
    }
}
```
Request Timeouts

Specify timeouts using Go’s Context type

```go
func printUSD(client pb.CurrencyServiceClient) {
    ctx, cancel := context.WithTimeout(context.Background(), 200*time.Millisecond)
    defer cancel()

    curReq := &pb.CurrencyRequest{Code: "USD"}
    curList, err := client.GetCurrencyList(ctx, curReq)
    if err != nil {
        fmt.Println("error in printUSD:", err)
        return
    }

    fmt.Println("\nUSD Countries")
    fmt.Println("-------------")
    for _, cur := range curList.Items {
        fmt.Printf("%-50s%-10s\n", cur.GetCountry(), cur.GetCode())
    }
}
```

Context is used to specify timeout value before it is passed to the remote method call.

gRPC will generate an error if call does not complete within deadline.
Error Handling

gRPC Errors are structured and can return valuable information to clients with rich detail.
gRPC Errors

All errors have an associated numeric status code

Errors can contain complex objects

Wrap your server errors in status.Error()

Server-side anti-pattern: don’t return naked errors
Creating Server Errors with Status Code

```go
func (c *CurrencyService) GetCurrencyStream(
    req *pb.CurrencyRequest,
    stream pb.CurrencyService_GetCurrencyStreamServer,
) error {
    if req.GetNumber() == 0 && req.getCode() == ""
    return status.Errorf(
        codes.InvalidArgument,
        "must provide currency number or code",
    )
    ...
    return nil
}
```

Wrap errors in `status.Errorf` calls.
Specify status code and error message.
Handling Errors on Client

Use status code for smarter error handling

```go
if stat, ok := status.FromError(err); ok {
    switch stat.Code() {
    case codes.InvalidArgument:
        fmt.Println("error in printEUR:", err)
        return
    default:
        // other err type, do something with it
        fmt.Println(err)
    }
}
```
Create a gRPC request status. 

Add any complex type to the status object. 

Return status detail as an error.
Extracting Objects from Error

```go
if err != nil {
    if stat, ok := status.FromError(err); ok {
        switch stat.Code() {
        case codes.InvalidArgument:
            // see if detail object sent, if so assert to Currency
            if len(stat.Details()) > 0 {
                detail := stat.Details()[0]
                if cur, ok := detail.(*pb.Currency); ok {
                    fmt.Printf("%v [Detail: currency=%v]\n", err, cur)
                }
                return
            }
        default:
            // handle other errors here
            fmt.Println(err)
            return
        }
    }
}
```
Other Features
Additional gRPC Features

- Extensible middleware API using interceptors and tap handlers
- Intercept requests to implement retry/backoff logic
- Add tracing using interceptors
- Add HTTP stream taps to implement rate limits
- Support for pluggable authorization
- Ability to limit message size for both request and response
Thank You!

@codeDellEMC

grpc.io