# gRPC Testing in Go

28 March 2019

Ákos Frohner

## Overview

Please study 'Unit Testing in Go' first!

Code:

github.com/szamcsi/golang-elte-2019-public/ (https://github.com/szamcsi/golang-elte-2019-public/)

Examples are in **ex?** subdirectories with skeleton code.

## Exercise 1

## Exercise 1: proto/lines/lines.proto

Moving the "complex" functionality into a service:

```
$ go generate linesservice.go
$ go test linesservice.go linesservice_test.go
```

```
syntax = "proto3";

package lines;

message CountRequest {
  repeated string lines = 1;
}

message CountResponse {
  int32 min = 1;
  int32 max = 2;
  int32 count = 3;
}

service LinesService {
  // Count counts lines and calculates minimum/maximum line length.
  rpc Count(CountRequest) returns (CountResponse) {
  }
}
```

## Exercise 1: ex1/linesservice/linesservice.go

```go
func (s *service) Count(_ context.Context, req *pb.CountRequest) (*pb.CountResponse, error) {
    if len(req.Lines) == 0 {
        return nil, grpc.Errorf(codes.InvalidArgument,
            "must provide at least one line in the request")
    }
    resp := &pb.CountResponse{Min: math.MaxInt32}
    for _, l := range req.Lines {
        resp.Count++
        if int32(len(l)) < resp.Min {
            resp.Min = int32(len(l))
        }
        if int32(len(l)) > resp.Max {
            resp.Max = int32(len(l))
        }
    }
    return resp, nil
}
```

## Exercise 1: comparing protobufs

- use **cmp.Equal** and **cmp.Diff**

### `ex1/linesservice/linesservice_test.go`

```go
func TestCount(t *testing.T) {
    for _, tc := range []struct {
        lines []string
        resp  *pb.CountResponse
        code  codes.Code
    }{
        {
            lines: []string{"1"},
            // TODO: resp
        },
        // ...
```

## Exercise 1: status codes

- use **grpc.Errorf** to create and check **codes.Code** for broad error categories.

`ex1/linesservice/linesservice_test.go`

```
        resp, err := s.Count(nil, &pb.CountRequest{Lines: tc.lines})
        if got := grpc.Code(err); got != tc.code {
            t.Errorf("Count(%v) err = %s; want = %s", tc.lines, got, tc.code)
            continue
        }
```

## Solution 1: ex1/linesservice/linesservice_test.go

```
func TestCount(t *testing.T) {
    for _, tc := range []struct {
        lines []string
        resp  *pb.CountResponse
        code  codes.Code
    }{
        {
            lines: []string{"1"},
            resp:  &pb.CountResponse{Min: 1, Max: 1, Count: 1},
        },
        // ...
        {
            lines: []string{},
            code:  codes.InvalidArgument,
        },
    } {
        if got := grpc.Code(err); got != tc.code {
            t.Errorf("Count(%v) err = %s; want = %s", tc.lines, got, tc.code)
            continue
        }
        if tc.code == codes.OK && !cmp.Equal(resp, tc.resp) {
            t.Errorf("Count(%v) response differs from expected (got -> want):\n%s", tc.lines, cmp.Diff(r
        }
    }
}
```

## Global setup/teardown

If a test file contains a function:

```
func TestMain(m *testing.M)
```

then the generated test will call `TestMain` to run the tests. `TestMain` must call `m.Run()` to execute the tests, then at some point call `os.Exit` with the result of `m.Run`.

```
func TestMain(m *testing.M) {
  setUpAnExpensiveEnvironment()
  ret := m.Run()
  tearDownTheEnvironment()
  os.Exit(ret)
}
```

## Exercise 2

## Exercise 2: ex2/lc.go

### Starting the grpc service:

```
func server() {
    lis, err := net.Listen("tcp", port)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    s := grpc.NewServer()
    pb.RegisterLinesServiceServer(s, linesservice.New())
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}

func main() {
    flag.Parse()

    // Starting the server locally to simplify the demo:
    go server()

    // Connecting the client:
    conn, err := grpc.Dial("localhost"+port, grpc.WithInsecure())
    if err != nil {
        log.Fatalf("failed to dial: %v", err)
    }
    defer conn.Close()
    client := pb.NewLinesServiceClient(conn)

    for _, path := range flag.Args() {
        mmc, err := lines.Count(context.Background(), client, path)
        if err != nil {
            fmt.Printf("ERROR: %s", err)
            continue
        }
        fmt.Printf("%+v\t%s\n", mmc, path)
    }
}
```

## Exercise 2: ex2/lc.go

Connecting with a client:

```
func main() {
    flag.Parse()

    // Starting the server locally to simplify the demo:
    go server()

    // Connecting the client:
    conn, err := grpc.Dial("localhost"+port, grpc.WithInsecure())
    if err != nil {
        log.Fatalf("failed to dial: %v", err)
    }
    defer conn.Close()
    client := pb.NewLinesServiceClient(conn)

    for _, path := range flag.Args() {
        mmc, err := lines.Count(context.Background(), client, path)
        if err != nil {
            fmt.Printf("ERROR: %s", err)
            continue
        }
        fmt.Printf("%+v\t%s\n", mmc, path)
    }
}
```

## Exercise 2: ex2/lines.go

Using the grpc service:

```go
func Count(ctx context.Context, client pb.LinesServiceClient, path string) (*MinMaxCount, error) {
	lines, err := load(path)
	if err != nil {
		return nil, err
	}
	resp, err := client.Count(ctx, &pb.CountRequest{Lines: lines})
	if err != nil {
		return nil, err
	}
	return &MinMaxCount{
		Min:   resp.Min,
		Max:   resp.Max,
		Count: resp.Count,
	}, nil
}
```

## Exercise 2: testing client with in-process server

Create a client, similar to lc.go!

## Solution 2: ex2/lines/lines_test.go

```
func TestCount(t *testing.T) {
    conn, err := grpc.Dial("localhost"+testPort, grpc.WithInsecure())
    if err != nil {
        t.Fatalf("failed to dial: %v", err)
    }
    defer conn.Close()
    client := pb.NewLinesServiceClient(conn)
```

## Exercise 3

## Exercise 3: ex3/lines/lines_test.go

Try using a local fake, instead of a real gRPC server!

Hint: check the generated proto library!

The code you have to complete:

```go
func (f *fake) Count(_ context.Context, req *pb.CountRequest, _ ...grpc.CallOption) (*pb.CountResponse,
    // TODO: implement a stub that will always return Min 1, Max 2, Count 3.
}
```

This is dependency injection using a different implementation,
therefore you will have to use an interface at an unmarked place.

## Solution 3: testing client with fake (ex3/lines/lines_test.go)

```
func (f *fake) Count(_ context.Context, req *pb.CountRequest, _ ...grpc.CallOption) (*pb.CountResponse,
    return &pb.CountResponse{
        Min:   1,
        Max:   2,
        Count: 3,
    }, nil
}
```

## Solution 3: ex3/lines/lines.go

Faking the pre-generated interface:

```
func Count(ctx context.Context, client pb.LinesServiceClient, path string) (...) {
```

## Exercise 4

## Exercise 4: proto/texts/texts.proto

Instead of the simple 'lines' service we use the 'texts' service, which has many more interesting methods -- although we only use a single one: **Count**

```
service TextsService {
  // Count counts lines and calculates minimum/maximum line length.
  rpc Count(CountRequest) returns (CountResponse) {
  }

  // Synonym returns synonym words.
  rpc Synonym(WordRequest) returns (WordsResponse) {
  }

  // Definition returns the definition of a word.
  rpc Definition(WordRequest) returns (DefinitionResponse) {
  }
}
```

## Exercise 4: testing client of a complex server with fake

The `lines` package takes `pb.TextsServiceClient' as argument:

```
func Count(ctx context.Context, client pb.TextsServiceClient, path string) (*MinMaxCount, error) {
```

Using the same test code will be problematic, because this does not implement all the required methods:

```
type fake struct{} // TODO: use fake for texts_proto.TextsServiceClient
```

Find a solution to be able to run `lines_test`!

## Non-solution 4: unimplemented

```
type fake struct {}

func (c *fake) Count(...) (*CountResponse, error) {
  // TODO: actual fake code
}

func (c *fake) Synonym(...) (*WordsResponse, error) {
  return errors.new("unimplemented")
}

func (c *fake) Definition(...) (*DefinitionResponse, error) {
  return errors.new("unimplemented")
}
```

## Solution 4: narrower interface

### ex4/lines/lines.go

```go
type Counter interface {
    Count(_ context.Context, req *pb.CountRequest, _ ...grpc.CallOption) (*pb.CountResponse, error)
}
```

```go
func Count(ctx context.Context, client Counter, path string) (*MinMaxCount, error) {
```

### ex4/lines/lines_test.go

```go
type fake struct{}

func (f *fake) Count(_ context.Context, req *pb.CountRequest, _ ...grpc.CallOption) (*pb.CountResponse,
```

## Advanced Solution 4 (may skip)

### ex4/lines/lines_test.go

```
type fake struct {
    pb.TextsServiceClient
}

func (f *fake) Count(_ context.Context, req *pb.CountRequest, _ ...grpc.CallOption) (*pb.CountResponse,
```

## Further reading

- golang.org/pkg/net/http/httptest/ (https://golang.org/pkg/net/http/httptest/)

## Thank you

Ákos Frohner
szamcsi@google.com (mailto:szamcsi@google.com)