Задание 3. Разработка веб-сервиса на основе gRPC

Цель: на языке высокого уровня (Java, C#, Python, Go и др. – на выбор обучающегося) реализовать gRPC веб-сервис и клиента для него, которые бы обеспечивали функционирование социальной сети. Сервис должен предоставлять API для просмотра ленты сообщений, в которую пользователи могут загружать сообщения из своих клиентов. Другие клиенты могут «лайкать» чужие посты и оставлять к ним комментарии.

1.1 Критерии оценивания

№	Задача	**Баллы**
1.	Реализовать и разместить локально простейший gRPC-сервис «Reverse» который принимает от клиента строку и возвращает ее в обратном порядке.	5
2.	Реализовать и разместить локально gRPC веб-сервис, и клиента для социальной сети обеспечивающий сбор и отображение ленты сообщений от пользователей.	5
3.	Сдача в срок: задачи 1-2 сданы до 26 апреля – 2 балла; до 9 мая– 1 балл; после 9 мая – 0 баллов.	2
4.	Реализовать и разместить локально gRPC веб-сервис, и клиента для социальной сети обеспечивающий возможность лайкать сообщения и оставлять комментарии к чужим постам.	4
5.	Разместить разработанный gRPC-сервис в облаке и продемонстрировать его работу.	4
6*	**Реализовать функционал прямой отправки текстовых сообщений от одного пользователя вашей социальной сети другому**	**5**

1.2 Методические указания

Рассмотрим пример реализации gRPC веб-сервиса, обеспечивающего «разворот» входящий строки текста на языке Go (на основе статьи https://habr.com/ru/post/461279/)

1.3 Примеры реализации

1.3.1 Разработка gRPC сервиса на основе Go

1.3.2 .proto-файл

.proto-файл описысывает, какие операции наш сервис будет осуществлять и какими данными он при этом будет обмениваться. Создаем в проекте папку ***proto***, а в ней — файл ***reverse.proto***

```
syntax = "proto3";

package reverse;

service Reverse {
    rpc Do(Request) returns (Response) {}
}

message Request {
    string message =1;
}

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

Функция, которая вызывается удаленно на сервере и возвращает данные клиенту, помечается как ***rpc***. Структуры данных, служащие для обмена информацией между взаимодействующими узлами, помечаются как ***message***. Каждому полю сообщения необходимо присвоить порядковый номер. В данном случае наша функция принимает от клиента сообщения типа ***Request*** и возвращает сообщения типа ***Response***.
Как только мы создали **.proto**-файл, необходимо получить **.go**-файл нашего сервиса. Для этого нужно выполнить седующую консольную команду в папке ***proto***:

```
$ protoc -I . reverse.proto --go_out=plugins=grpc:.
```

Разумеется, сначала вам нужно выполнить сборку gRPC.
Выполнение вышеприведенной команды создаст новый **.go**-файл, содержащий методы для создания клиента, сервера и сообщений, которыми они обмениваются. Если мы вызовем **godoc**, то увидим следующее:

```
$ godoc .
PACKAGE DOCUMENTATION

package reverse
    import "."

Package reverse is a generated protocol buffer package.

It is generated from these files:

reverse.proto

It has these top-level messages:

Request
 Response
....
```

1.3.3 Клиент

Было бы неплохо, если бы наш клиент работал вот так:

```
reverse "this is a test"
```

```
tset a si siht
```

Вот код, который создает **gRPC**-клиент, используя структуры данных, сгенерированные из **.proto**-файла:

```
package main

import (
    "context"
    "fmt"
    "os"
     pb "github.com/matzhouse/go-grpc/proto"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)
```

```
func main() {
    opts := []grpc.DialOption{
        grpc.WithInsecure(),
    }
    args := os.Args
    conn, err := grpc.Dial("127.0.0.1:5300", opts...)

    if err != nil {
        grpclog.Fatalf("fail to dial: %v", err)
    }

    defer conn.Close()

    client := pb.NewReverseClient(conn)
    request := &pb.Request{
        Message: args[1],
    }
    response, err := client.Do(context.Background(), request)

    if err != nil {
        grpclog.Fatalf("fail to dial: %v", err)
    }

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

1.3.4 Сервер

Сервер использует тот же самый сгенерированный **.go**-файл. Однако он определяет только интерфейс сервера, логику же нам придется реализовать самостоятельно:

```
package main

import (
    "net"
    pb "github.com/matzhouse/go-grpc/proto"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)

func main() {
    listener, err := net.Listen("tcp", ":5300")

    if err != nil {
        grpclog.Fatalf("failed to listen: %v", err)
    }

    opts := []grpc.ServerOption{}
    grpcServer := grpc.NewServer(opts...)
```

```
    pb.RegisterReverseServer(grpcServer, &server{})
    grpcServer.Serve(listener)
}

type server struct{}

func (s *server) Do(c context.Context, request *pb.Request)
(response *pb.Response, err error) {
    n := 0
    // Create an array of runes to safely reverse a string.
    rune := make([]rune, len(request.Message))

    for _, r := range request.Message {
        rune[n] = r
        n++
    }

    // Reverse using runes.
    rune = rune[0:n]

    for i := 0; i < n/2; i++ {
        rune[i], rune[n-1-i] = rune[n-1-i], rune[i]
    }

    output := string(rune)
    response = &pb.Response{
        Message: output,
    }

    return response, nil
}
```

После того, как мы подготовили исходный код сервера, мы можем запустить его следующей командой

```
$ go build -o reverse
$ ./reverse "this is a test"
tset a si siht
```

1.4 Задание на самостоятельную работу

На основе представленных примеров вам предлагается самостоятельно реализовать gRPC веб-сервис, обеспечивающий решение задач 1-5.

1.5 Ссылки

Для дальнейшего самостоятельного изучения данной темы можно воспользоваться следующими ресурсами:

1. Руководство для начинающих в платформе Amazon Web Services доступны по ссылке: https://aws.amazon.com/ru/getting-started/?nc1=h_ls.
2. Как создать простой микросервис на Golang и gRPC и выполнить его контейнеризацию с помощью Docker https://habr.com/ru/post/461279/
3. Создание gRPC сервиса на Node.js https://codelabs.developers.google.com/codelabs/cloud-grpc-ru/index.html?index=..%2F..lang-ru#0