gRPC

Servisno orijentisane arhitekture

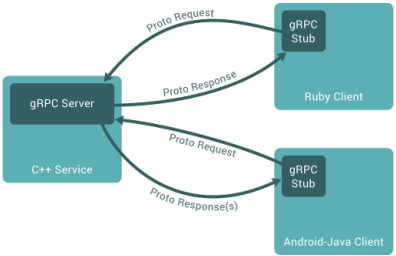
Univerzitet u Novom Sadu
Fakultet tehničkih nauka

REST vs RPC

- RPC (Remote Procedure Call) i REST predstavljaju dva različita pristupa za formiranje API-ja servisa dostupnih preko mreže
- Osnova ideja RPC-a je iniciranje operacije udaljenih programa kao da su oni dostupni lokalno
- Na taj način, skriva se komunikacija koja se odvija preko mreže
- Dok je REST orijentisan na resurse, u osnovi različitih RPC radnih okvira i protokola nalaze se akcije
- U situacijama kada je potrebno obezbediti efikasnu i brzu komunikaciju i razmenu što manje količine podataka preko mreže, RPC rešenja mogu biti pogodnija opcija od REST-a

gRPC

- gRPC je radni okvir za komunikaciju preko mreže zasnovan na RPC mehanizmu
- Razvila ga je kompanija Google i javno je dostupan od 2015. godine
- Zbog toga što koristi HTTP/2 i Protobuf (Protocol Buffers) binarni format poruka, vrlo je brz i poruke koje se razmenjuju su znatno manje u odnosu na one koju su tekstualnog, na primer JSON ili XML formata
- Za veliki broj programskih jezika dostupni su alati i biblioteke potrebne za rad sa gRPC-jem



Instalacija

Za implementaciju gRPC servisa potrebni su vam:

1. *Go*
2. *protoc* kompajer - na osnovu definicije servisa i poruka generiše serverski i klijentski kod u željenom programskom jeziku (uputstvo za instalaciju)
3. Go plugin-ovi za kompajer:

```
go install google.golang.org/protobuf/cmd/protoc-gen-go@latest
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
# izmeniti PATH environment varijablu tako da protoc
# može da pronadje instalirane go plugin-ove
export PATH="$PATH:$(go env GOPATH)/bin"
```

Postupak implementacije gRPC servisa

Uključuje sledeće korake:

1. Definisanje servisa u okviru .proto fajla
2. Generisanje serverskog i klijentskog koda na osnovu .proto fajla
3. Implementacija procedura na serverskoj strani
4. Pozivanje procedura na klijentskoj strani

Definicija servisa I

- Za definiciju servisa (procedura i poruka), koristi se IDL (Interface Definition Language) i sadržaj se čuva u datoteci sa ekstenzijom .proto
- Primer definicije Protobuf poruke:

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

- Za svako polje prvo se navodi tip, zatim naziv i nakon toga broj koji mora biti jedinstven na nivou polja u okviru poruke (potreban za serijalizaciju poruke)
- Ako polje predstavlja listu vrednosti, na početku se navodi ključna reč *repeated*
- Detaljno uputstvo za upotrebu jezika možete pronaći na ovom linku

Definicija servisa II

Primer jedne .proto datoteke:

```
syntax = "proto3";

option go_package = "proto/product";

service ProductService {
    rpc GetProduct(GetProductRequest) returns (GetProductResponse) {}
    rpc UpsertProduct(UpsertProductRequest) returns (UpsertProductResponse) {}
    rpc DeleteProduct(DeleteProductRequest) returns (DeleteProductResponse) {}
}

message Product {
    int32 id = 1;
    enum Category {
        CLOTHES = 0;
        ELECTRONICS = 1;
        BOOKS = 2;
    };
    Category category = 2;
    string description = 3;
    double price = 4;
}
. . .
```

```
. . .
message GetProductRequest {
    int32 id = 1;
}

message GetProductResponse {
    Product product = 1;
}

message UpsertProductRequest {
    Product product = 1;
}

message UpsertProductResponse {
    Product product = 1;
}

message DeleteProductRequest {
    int32 id = 1;
}

message DeleteProductResponse {
}
```

Generisanje koda

- Ako se datoteka prikazana na prethodnom slajdu naziva *product_service.proto*, ovo je komanda na osnovu koje će se generisati klijentski i serverski kod (u *product* direktorijumu, koji se nalazi u trenutnom radnom direktorijumu):

```
protoc
      --go_out=./product --go_opt=paths=source_relative
      --go-grpc_out=./product --go-grpc_opt=paths=source_relative
      product_service.proto
```

- Nakon poziva komande, generisaće se datoteke *product_service.pb.go* i *product_service_grpc.pb.go*, čiji sadržaj **ne treba** da modifikujete

Server I

- U datoteci *product_service_grpc.pb.go* nalazi se definicija interfejsa koji gRPC server treba da implementira
- Metode interfejsa predstavljaju procedure opisane kroz .proto fajl
- U go-u, pored parametra funkcije koji predstavlja zahtev, kao prvi parametar uvek se navodi promenljiva Context tipa
- Deo implementacije gRPC servisa:

```
type Server struct {
        product.UnimplementedProductServiceServer
        products map[int32]*product.Product
}

func (s Server) GetProduct(ctx context.Context, request *product.GetProductRequest) (*product.GetProductResponse,
↪  error) {
        p, ok := s.products[request.Id]
        if !ok {
                return nil, status.Error(codes.NotFound, "product not found")
        }
        response := &product.GetProductResponse{Product: p}
        return response, nil
}
```

Server II

- Server se registruje i pokreće na sledeći način:

```
lis, err := net.Listen("tcp", "localhost:8000")
if err != nil {
      log.Fatalf("failed to listen: %v", err)
}

var opts []grpc.ServerOption
grpcServer := grpc.NewServer(opts...)

product.RegisterProductServiceServer(grpcServer, Server{products: products})
grpcServer.Serve(lis)
```

Klijent

- Kako bismo kontaktirali kreirani gRPC servis i pozivali procedure, potrebno je da registrujemo gRPC klijenta, na sledeći način:

```
conn, err := grpc.Dial("localhost:8000",
↪  grpc.WithTransportCredentials(insecure.NewCredentials()))
if err != nil {
      log.Fatal(err)
}
defer conn.Close()

productService := product.NewProductServiceClient(conn)
```

- Nakon ovoga, procedure pozivamo kao i nad objektom koji nam je dostupan lokalno

```
getResp, err := productService.GetProduct(context.Background(),
↪  &product.GetProductRequest{Id: 1})
```

Postman

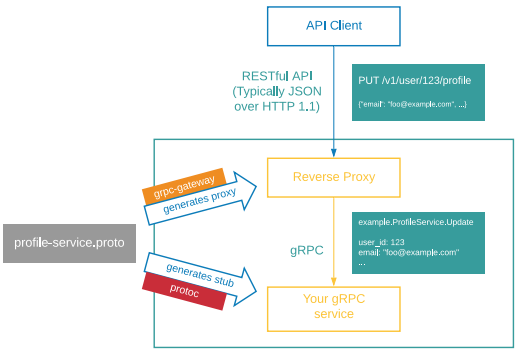
- Kako biste proverili da li vaš gRPC servis korektno radi, možete koristiti Postman alat za rad sa API-jima
- Video uputstvo za slanje gRPC zahteva možete pronaći na ovom linku
- Ako dostupne procedure želite da učitate upotrebom refleksije servera, potrebno je da prilikom registracije servisa dodate sledeći kod:

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

product.RegisterProductServiceServer(grpcServer, Server{products: products})
reflection.Register(grpcServer)
. . .
```

REST vs gRPC klijenti

- gRPC koristi binarni format poruka, što može otežati testiranje aplikacije ili ograničiti tipove klijenata koji mogu koristiti naš servis
- grpc-gateway je plugin protoc kompajlera koji za nas generiše reverse proxy kod koji će da izvrši translaciju JSON - protobuf i obrnuto
- Vaš servis može da sadrži i REST i gRPC server i tako servira zahteve, ali mi ćemo ovaj plugin koristiti kao odvojenu komponentu - API gateway
- Prihvata REST zahtev, transformiše da u gRPC i prosleđuje odgovarajućem servisu, kasnije protobuf odgovor transformiše u JSON



Instalacija paketa

- Komanda kojom instalirate sve što plugin-u treba za rad:

```
$ go install \
    github.com/grpc-ecosystem/grpc-gateway/v2/protoc-gen-grpc-gateway \
    github.com/grpc-ecosystem/grpc-gateway/v2/protoc-gen-openapiv2 \
    google.golang.org/protobuf/cmd/protoc-gen-go \
    google.golang.org/grpc/cmd/protoc-gen-go-grpc
```

Mapiranje

- Kako biste mapirali gRPC zahtev na odgovarajući REST zahtev, za svaki RPC iz .proto fajla treba dodatno da vežete barem URI i HTTP metodu

```
syntax = "proto3";
option go_package = "proto/greeter";
import "google/api/annotations.proto";

service GreeterService {
  rpc Greet(Request) returns (Response) {
    option (google.api.http) = {
      post: "/"
      body: "*"
    };
  }
}
```

```
. . .

message Request {
  string name = 1;
}

message Response {
  string greeting = 2;
}
```

Generisanje koda

- Komanda kojom generišete protobuf poruke, gRPC server i reverse proxy:

```
$ protoc
        --go_out=./greeter
        --go_opt=paths=source_relative
        --go-grpc_out=./greeter
        --go-grpc_opt=paths=source_relative
        --grpc-gateway_out ./greeter
        --grpc-gateway_opt paths=source_relative
        greeter-service.proto
```

Inicijalizacija i pokretanje API gateway-a

```
cfg := config.GetConfig()

conn, err := grpc.DialContext(
        context.Background(),
        cfg.GreeterServiceAddress,
        grpc.WithBlock(),
        grpc.WithTransportCredentials(
                insecure.NewCredentials()),
)

if err != nil {
        log.Fatalln("Failed to dial server:", err)
}

gwmux := runtime.NewServeMux()
// Register Greeter
client := greeter.NewGreeterServiceClient(conn)
err = greeter.RegisterGreeterServiceHandlerClient(
        context.Background(),
        gwmux,
        client,
)
```

```
. . .

if err != nil {
        log.Fatalln("Failed to register gateway:", err)
}

gwServer := &http.Server{
        Addr:    cfg.Address,
        Handler: gwmux,
}

go func() {
        if err := gwServer.ListenAndServe(); err != nil {
                log.Fatal("server error: ", err)
        }
}()
```