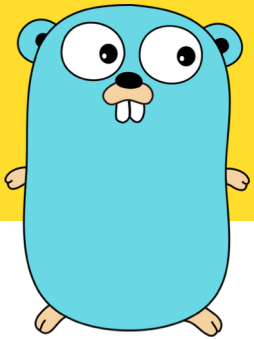


gRPC

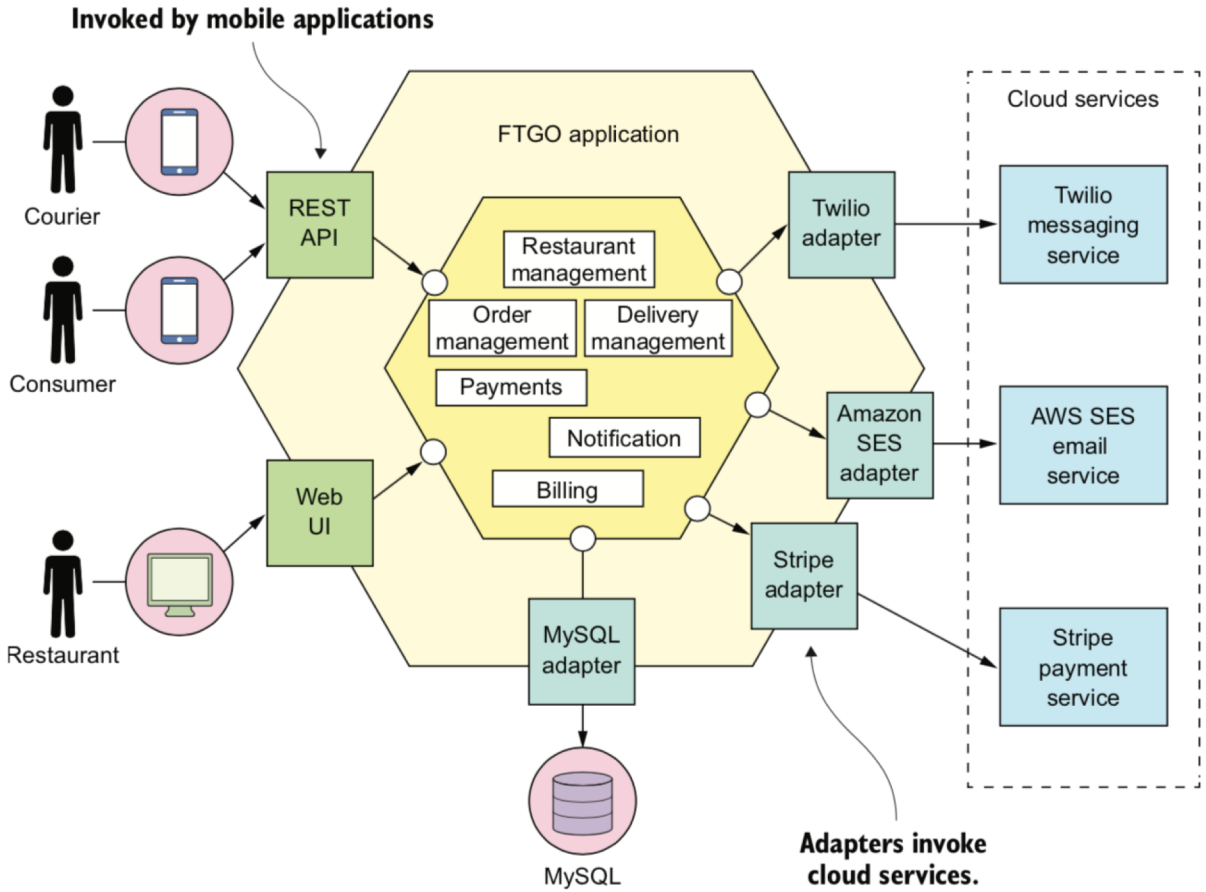
Tinkoff.ru

План занятия

1. Микросервисы
2. HTTP/2
3. Protocol Buffers
4. gRPC
5. gRPC beyond basics

Микросервисы

Пример монолитного приложения



Недостатки и преимущества монолита

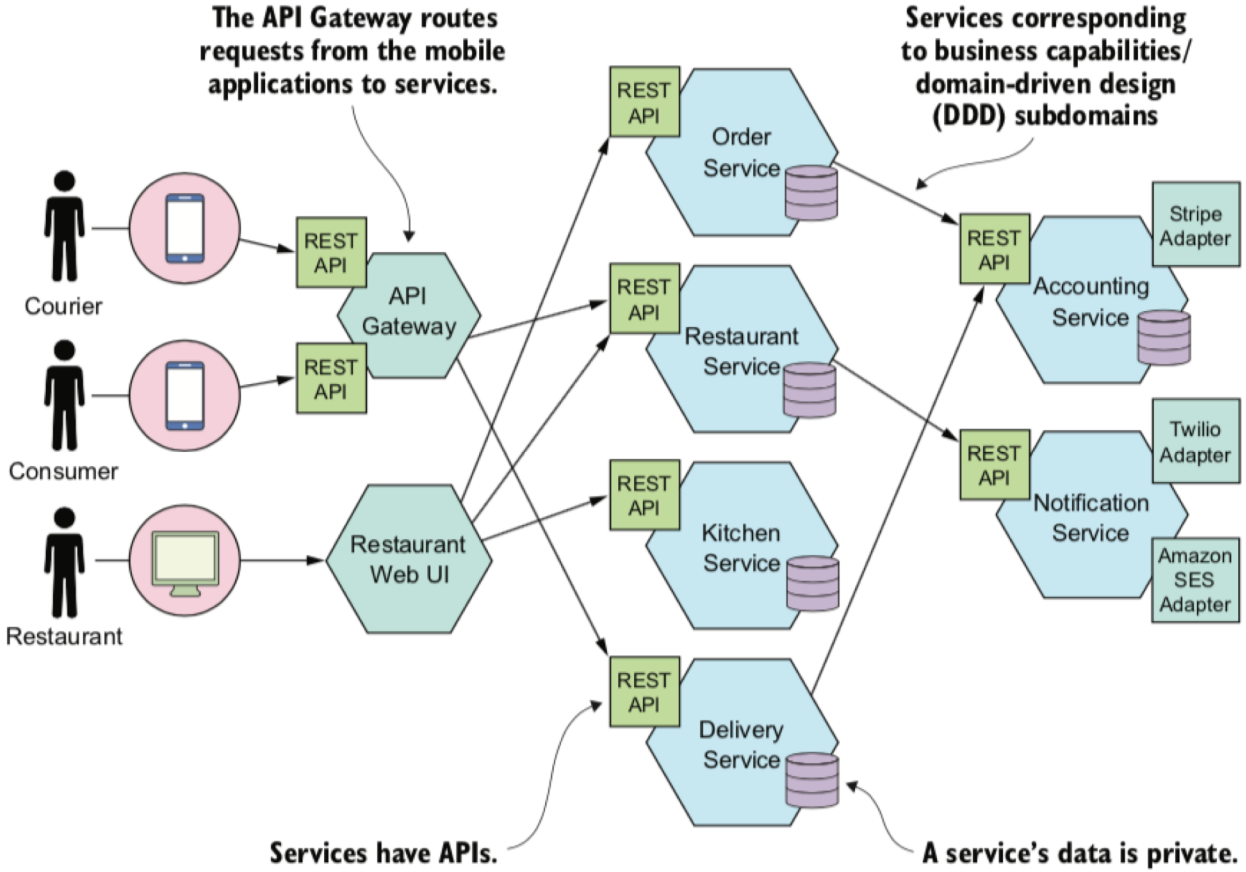
Разработка становится медленней из-за сложности монолита

Доставка до прода становится медленной (увеличивается TTM (Time To Market))

Масштабирование становится неоправданно дорогим

Применение новых технологий становится практически невозможным

Пример микросервисной архитектуры



Недостатки и преимущества микросервисов

Разработка становится быстрее

Time to market снижается

Появляется гибкость при масштабировании сервисов

Появляется возможность экспериментировать с новыми технологиями и применять их в боевых условиях

Дефект в одном из сервисов не «топит» всю систему

HTTP/2

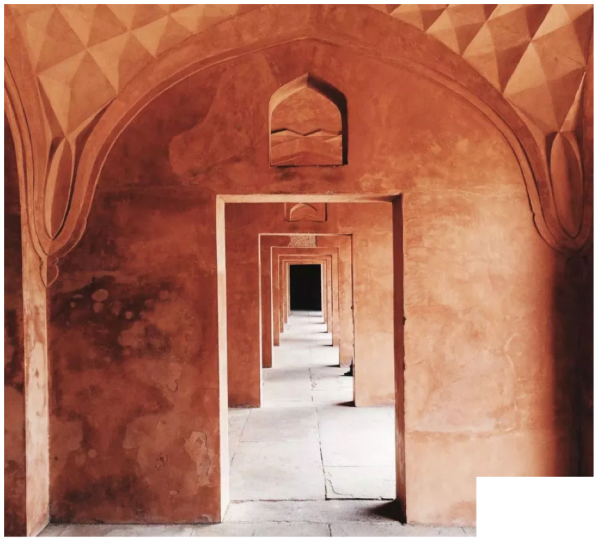
HTTP/2 vs HTTP/1.1

https://imagekit.io/demo/http2-vs-http1

Photograph clicked by: not.sosubtle

HTTP/2

HTTP/1.1



Вспомним про HTTP/1.1

Работает по принципу запрос/ответ (нет server push)

Создает новое TCP соединение на каждый запрос

Заголовки всегда передаются PLAINTEXT

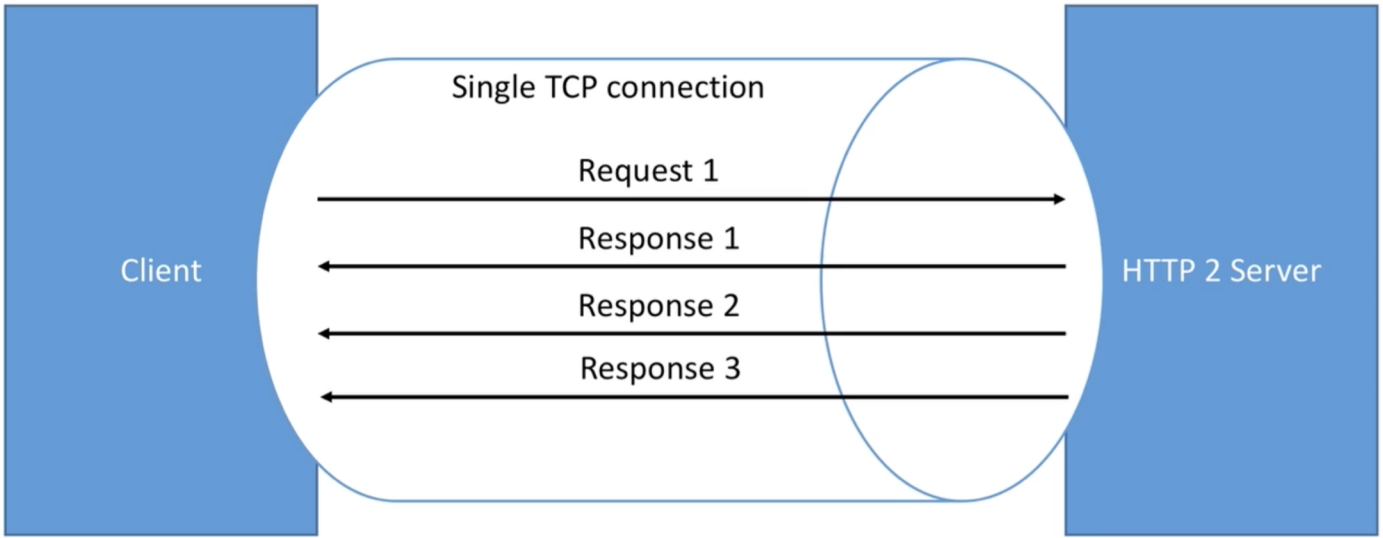
HTTP/2 (old name SPDY)

Поддерживает двусторонний обмен

Поддерживает multiplexing (одновременно клиент и сервер могут отправлять запросы)

Поддерживает компрессию заголовков

BINARY



Protocol Buffer (ProtoBuf)

Недостатки и достоинства JSON

Человекочитаемый формат

Компактнее чем XML

Нет схемы, которую можно использовать при разработке

Есть повторяющиеся элементы, которые помогают человеку читать формат, но увеличивают в размере

Но можно и лучше (ProtoBuf)

Строгая типизация

Данные занимают меньше места

Потребляется меньше CPU для маршалинга и анмаршалинга

Есть строгая схема, которая используется при разработке

Обширный набор поддерживаемых языков программирования (Java, Scala, Go, Python etc.)

Круто! Как это работает?

```
syntax = "proto3";

message LoginRequest {
    string login = 1;
    string password = 2;
}

message LoginResponse {
    string result = 1;
}
```

Версия протокола

Сообщение

Тип, название и порядковый номер (или tag) поля

Тэг может принимать значения от 1 до 2^29-1.

Рэндж 19000-1999использоваться для тэгирования9 зарезервирован и не может

Пример сообщения

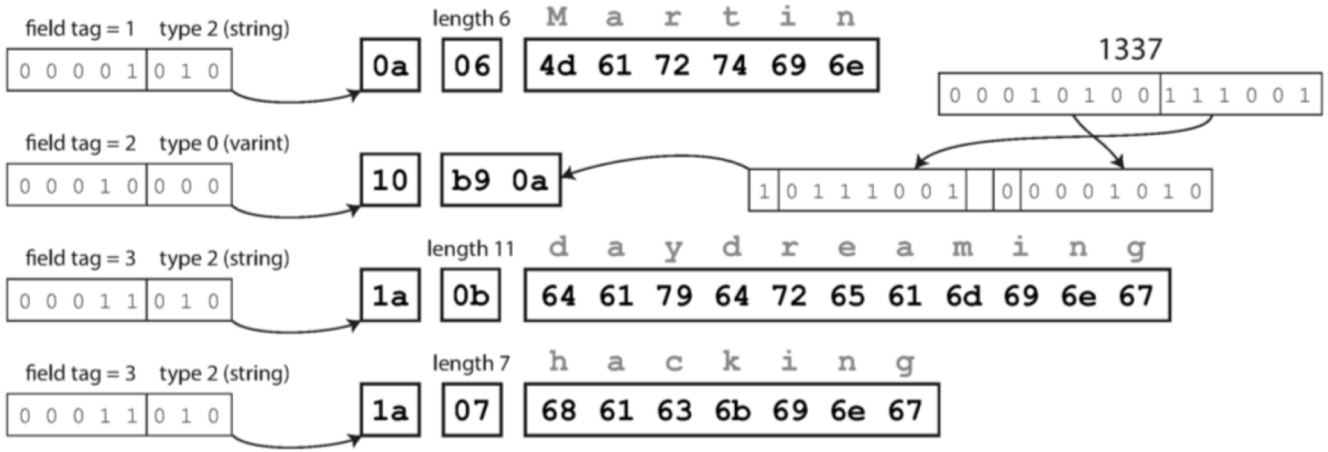
```
message Person {
    required string user_name        = 1;
    optional int64  favorite_number = 2;
    repeated string interests        = 3;
}
```

Protocol Buffers

Byte sequence (33 bytes):

```
0a 06 4d 61 72 74 69 6e 10 b9 0a 1a 0b 64 61 79 64 72 65 61
6d 69 6e 67 1a 07 68 61 63 6b 69 6e 67
```

Breakdown:



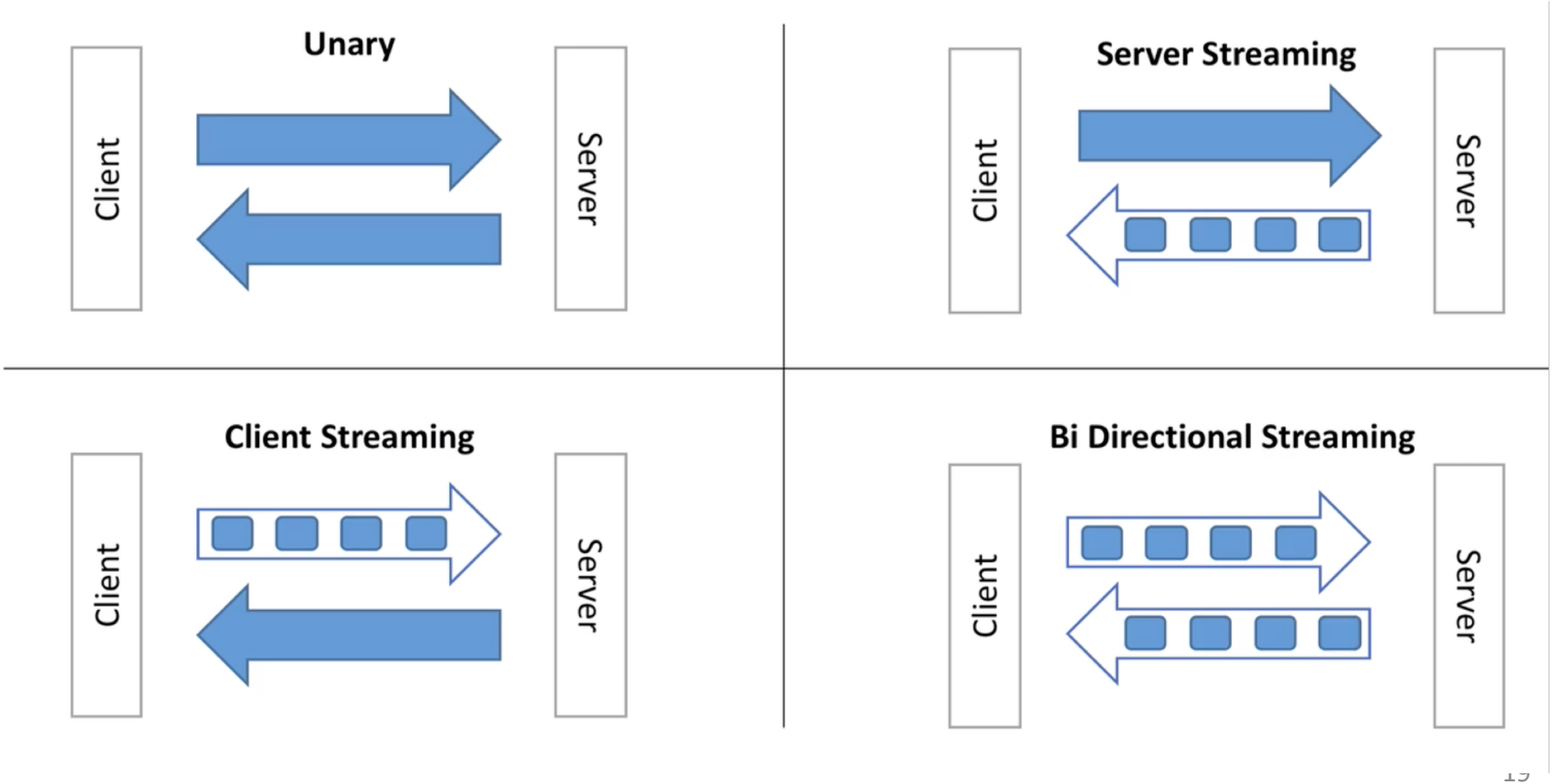
proto2 vs proto3

https://github.com/protocolbuffers/protobuf/releases/tag/v3.0.0

We recommend that new Protocol Buffers users use proto3. However, we do not generally recommend that existing users migrate from proto2 from proto3 due to API incompatibility, and we will continue to support proto2 for a long time.

gRPC
(теория)

Типы взаимодействия



gRPC vs REST

GRPC	REST
Protocol Buffers - smaller, faster	JSON - text based, slower, bigger
HTTP/2 (lower latency) – from 2015	HTTP1.1 (higher latency) – from 1997
Bidirectional & Async	Client => Server requests only
Stream Support	Request / Response support only
API Oriented – "What" (no constraints – free design)	CRUD Oriented (Create – Retrieve – Update – Delete / POST GET PUT DELETE)
Code Generation through Protocol Buffers in any language – 1st class citizen	Code generation through OpenAPI / Swagger (add-on) – 2nd class citizen
RPC Based - gRPC does the plumbing for us	HTTP verbs based – we have to write the plumbing or use a 3rd party library

Более подробный разбор насколько gRPC быстрый: https://husobee.github.io/golang/rest/grpc/2016/05/28/golang-rest-v-grpc.html

Почему gRPC?

RPC, который поддерживает более 11-ти языков (включая C, C++, Python, Go, Java, Scala и т.д.)

Используется поверх http/2

Поддерживает разные схемы стриминга

Фокусирует на разработке API, а не на проектирование ресурсов (что характерно для REST)

gRPC
(настройка среды)

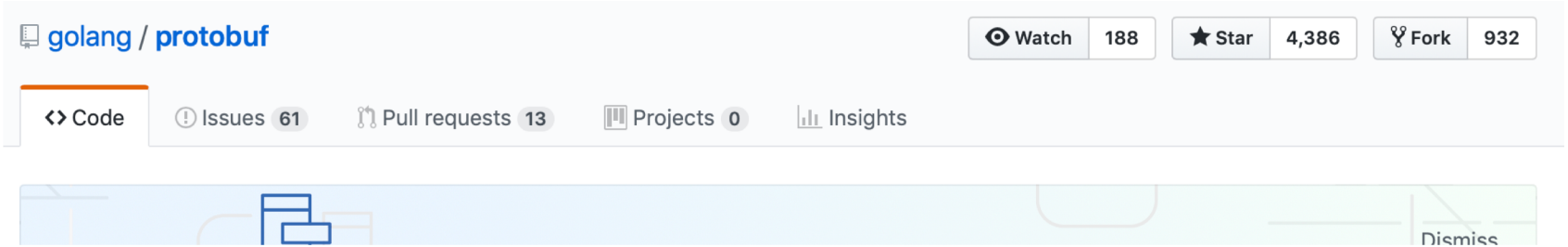
Установка protoc

https://github.com/protocolbuffers/protobuf/releases

protoc-3.7.1-linux-aarch_64.zip	1.36 MB
protoc-3.7.1-linux-ppcle_64.zip	1.5 MB
protoc-3.7.1-linux-x86_32.zip	1.41 MB
protoc-3.7.1-linux-x86_64.zip	1.46 MB
protoc-3.7.1-osx-x86_32.zip	2.71 MB
protoc-3.7.1-osx-x86_64.zip	2.68 MB
protoc-3.7.1-win32.zip	1.04 MB
protoc-3.7.1-win64.zip	1.35 MB

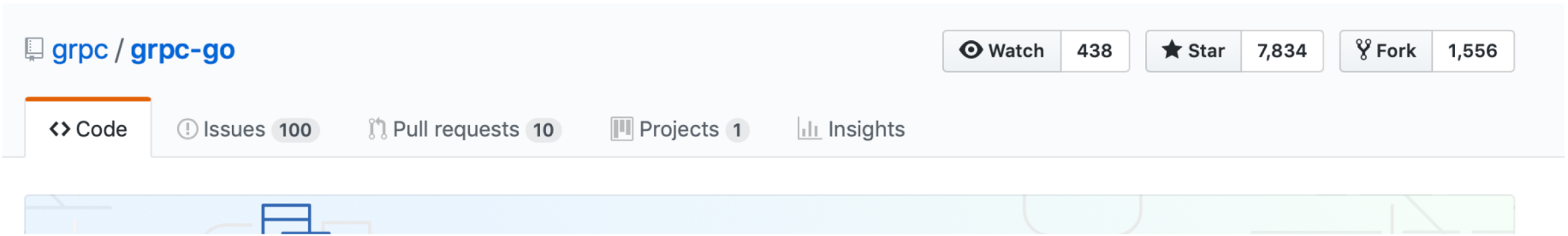
Добавляете в path папку с утилитой `protoc`

golang/protobuf




```
go get -u github.com/golang/protobuf/protoc-gen-go
```

grpc/grpc-go




```
go get -u google.golang.org/grpc
```

Создаем простой сервис

```
1  syntax = "proto3";
2
3  package greet;
4  option go_package = "greetpb";
5
6  service GreetService {}
```

protoc ./greet/greetpb/GreetService.proto --go_out=plugins=grpc:.

```
1  // Code generated by protoc-gen-go. DO NOT EDIT.
2  // source: greet/greetpb/GreetService.proto
3  package greetpb
4  …
5  var _GreetService_serviceDesc = grpc.ServiceDesc{
6      ServiceName: "greet.GreetService",
7      HandlerType: (*GreetServiceServer)(nil),
8      Methods:     []grpc.MethodDesc{},
9      Streams:     []grpc.StreamDesc{},
10     Metadata:    "greet/greetpb/GreetService.proto",
11 }
```

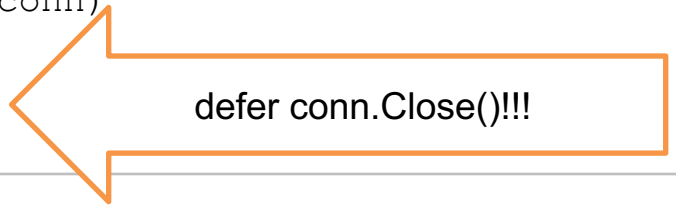
gRPC
(основы)

gRPC сервер

```go
import (
    "log"
    "tfs19s/L8/code/grpc/greet/greetpb"
    "net"
    "google.golang.org/grpc"
)
type server struct{}
func main() {
    listen, err := net.Listen("tcp", ":5000")
    if err != nil {
        log.Fatalf("can't listen on port: %v", err)
    }
    s := grpc.NewServer()
    greetpb.RegisterGreetServiceServer(s, server{})
    if err := s.Serve(listen); err != nil {
        log.Fatalf("can't register service server: %v", err)
    }
}
```

gRPC клиент

```
1  import (
2      "fmt"
3      "log"
4      "tfs19s/L8/code/grpc/greet/greetpb"
5
6      "google.golang.org/grpc"
7  )
8
9  func main() {
10     conn, err := grpc.Dial("localhost:5000", grpc.WithInsecure())
11     if err != nil {
12         log.Fatalf("can't connect to server: %v", err)
13     }
14     client := greetpb.NewGreetServiceClient(conn)
15     fmt.Printf("created client: %v", client)
16     conn.Close()
17 }
```



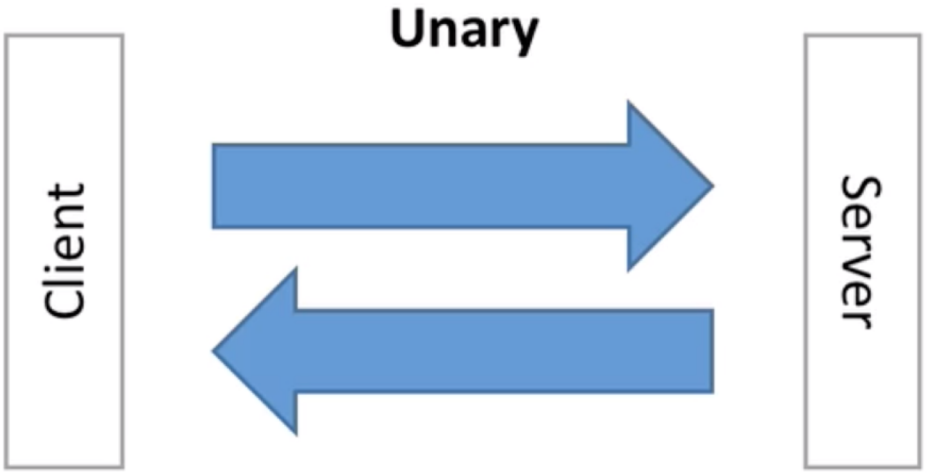
gRPC
(Тип взаимодействия: Unary)

Типы взаимодействия Unary

Наиболее простой

Клиент отправляет запрос и получает ответ от сервера

Наиболее распространненый



Описание сервиса

```
1  syntax = "proto3";
2
3  package login;
4  option go_package = "loginpb";
5
6  message LoginRequest {
7      string login = 1;
8      string password = 2;
9  }
10
11 message LoginResponse {
12     string result = 1;
13 }
14
15 service LoginService{
16     // Unary
17     rpc Login(LoginRequest) returns (LoginResponse) {};
18 }
```

```
protoc ./login/loginpb/LoginService.proto --go_out=plugins=grpc:.
```

Автогенеренный stub

```go
// LoginServiceServer is the server API for LoginService service.
type LoginServiceServer interface {
    // Unary
    Login(context.Context, *LoginRequest) (*LoginResponse, error)
}
```

```go
type LoginRequest struct {
    Login                string
`protobuf:"bytes,1,opt,name=login,proto3" json:"login,omitempty"`
    Password             string   `protobuf:"bytes,2,opt,name=password,proto3"
json:"password,omitempty"`
    XXX_NoUnkeyedLiteral struct{} `json:"-"`
    XXX_unrecognized     []byte   `json:"-"`
    XXX_sizecache        int32    `json:"-"`
}
```

```go
// Code generated by protoc-gen-go. DO NOT EDIT.
// source: login/loginpb/LoginService.proto
```

Сервер

```
1  type server struct{}
2  func (s *server) Login(ctx context.Context, req *loginpb.LoginRequest)
3  (*loginpb.LoginResponse, error) {
4     if req.Login == "login" && req.Password == "passwd" {
5        res := loginpb.LoginResponse{Result: "Ok"}
6        return &res, nil
7     }
8     res := loginpb.LoginResponse{Result: "Error"}
9     return &res, nil
10 }
11 func main() {
12    listen, err := net.Listen("tcp", ":5000")
13    if err != nil {
14       log.Fatalf("can't listen on port: %v", err)
15    }
16    s := grpc.NewServer()
17    loginpb.RegisterLoginServiceServer(s, &server{})
18 }
```

s.Server && If err != nil {}

Клиент

```
1  func main() {
2     conn, err := grpc.Dial("localhost:5000", grpc.WithInsecure())
3     if err != nil {
4        log.Fatalf("can't connect to server: %v", err)
5     }
6     defer conn.Close()
7     client := loginpb.NewLoginServiceClient(conn)
8
9     req := loginpb.LoginRequest{
10       Login:    "login",
11       Password: "passwd"}
12    resp, err := client.Login(context.Background(), &req)
13    if err != nil {
14       log.Fatalf("can't login: %v", err)
15    }
16    fmt.Printf("response %+v", resp)
17 }
```

response result:"Ok"

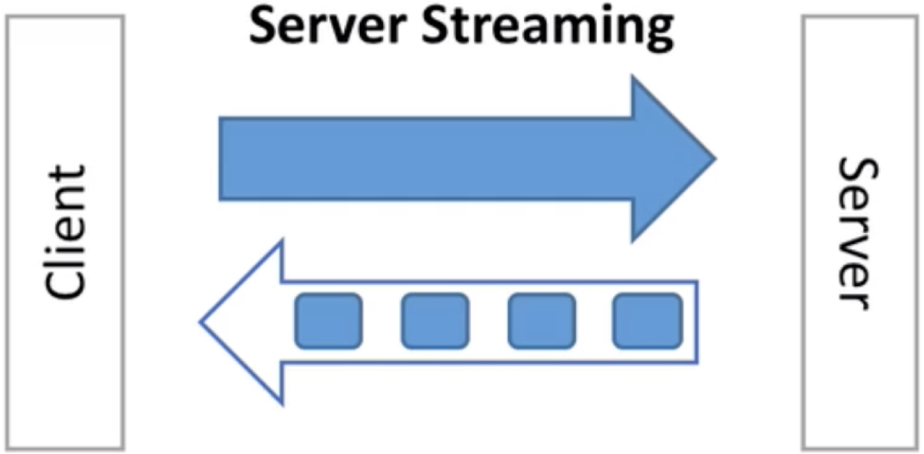
gRPC
(Тип взаимодействия: Server Streaming)

Типы взаимодействия Server Streaming

Мгновенная доставка изменений клиенту

Мы так уже делали – это ведь WebSocket

И да и нет, потому что gRPC использует возможности HTTP/2 для стримминга



Описание сервиса

```
1  syntax = "proto3";
2  package lots;
3  option go_package =  "lotspb";
4
5  message LotsRequest {
6      int32 limit = 1;
7  }
8  message Lot {
9      int32 ID = 1;
10     string desc = 2;
11     double price = 3;
12 }
13 message LotsResponse {
14     Lot lot = 1;
15 }
16 service LotsService {
17     rpc ActiveLots(LotsRequest) returns(stream LotsResponse) {};
18 }
```


```
protoc ./lots/lotspb/LotsService.proto --go_out=plugins=grpc:.
```

Автогенеренный stub

```
1  type LotsService_ActiveLotsServer interface {
2     Send(*LotsResponse) error
3     grpc.ServerStream
4  }
5
6  type lotsServiceActiveLotsServer struct {
7     grpc.ServerStream
8  }
9
10 func (x *lotsServiceActiveLotsServer) Send(m *LotsResponse) error {
11    return x.ServerStream.SendMsg(m)
12 }
```

```
// Code generated by protoc-gen-go. DO NOT EDIT.
// source: lots/lotspb/LotsService.proto
```

Сервер

```
1  func (s *server) ActiveLots(req *lotspb.LotsRequest, resp
2  lotspb.LotsService_ActiveLotsServer) error {
3     startPrice := rand.Intn(100)
4     for i := 1; i < 11; i++ {
5        res := lotspb.LotsResponse{
6           Lot: &lotspb.Lot{
7              ID:    int64(i),
8              Desc:  "Description",
9              Price: float64(startPrice * i),
10          },
11       }
12       resp.Send(&res)
13       time.Sleep(time.Second * 3)
14       if req.Limit < int64(i)+1 {
15          break
16       }
17    }
18    return nil
19 }
```

Клиент

```
1  req := lotspb.LotsRequest{
2     Limit: 3,
3  }
4
5  resp, err := client.ActiveLots(context.Background(), &req)
6  if err != nil {
7     log.Fatalf("can't get active lots: %v", err)
8  }
9
10 for {
11    lots, err := resp.Recv()
12    if err == io.EOF {
13       break
14    }
15    fmt.Printf("active lot: %+v\n", lots.Lot)
16 }
```

What if err != nil && err != io.EOF?

```
active lot: ID:1 desc:"Description" price:81
active lot: ID:2 desc:"Description" price:162
active lot: ID:3 desc:"Description" price:243
```

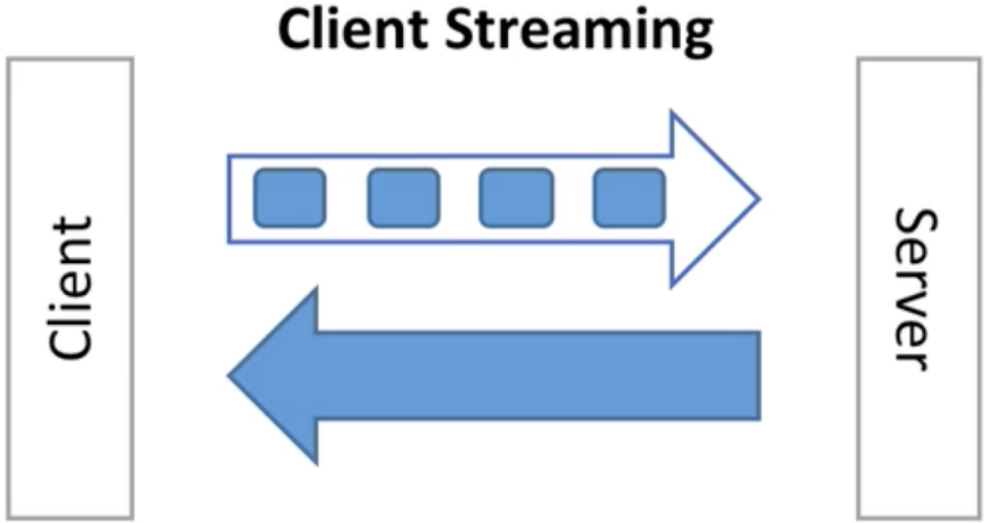
gRPC
(Тип взаимодействия: Client Streaming)

Типы взаимодействия Server Streaming

Разбиение объекта на более маленькие

Мы так уже делали – это ведь WebSocket

И да и нет, потому что gRPC использует возможности HTTP/2 для стримминга



Описание сервиса

```
1  syntax = "proto3";
2
3  package lots;
4  option go_package =  "orderspb";
5
6  message OrderRequest {
7      double price = 1;
8      int64 quantity = 2;
9  }
10
11 message OrdersResponse {
12     int64 executedOrders = 1;
13 }
14
15 service OrdersService {
16     rpc PostOrder(stream OrderRequest) returns(OrdersResponse) {};
17 }
```


```
protoc ./orders/orderspb/OrdersService.proto --go_out=plugins=grpc:.
```

Автогенеренный stub

```
1  type OrdersService_PostOrderClient interface {
2     Send(*OrderRequest) error
3     CloseAndRecv() (*OrdersResponse, error)
4     grpc.ClientStream
5  }
6
7  type ordersServicePostOrderClient struct {
8     grpc.ClientStream
9  }
10
11 func (x *ordersServicePostOrderClient) Send(m *OrderRequest) error {
12    return x.ClientStream.SendMsg(m)
13 }
```

```
// Code generated by protoc-gen-go. DO NOT EDIT.
// source: orders/orderspb/OrdersService.proto
```

Сервер

```
1  func (s *server) PostOrder(stream orderspb.OrdersService_PostOrderServer) error {
2     var executedOrders int
3     for {
4        req, err := stream.Recv()
5        if err != nil {
6           if err == io.EOF {
7              resp := orderspb.OrdersResponse{
8                 ExecutedOrders: int64(executedOrders),
9              }
10             stream.SendAndClose(&resp)
11             break
12          }
13          log.Fatalf("can't receive message from client: %v", err)
14       }
15       if rand.Intn(1000)%2 == 0 {
16          fmt.Printf("executed order with price %.2f and quantity %d", req.Price,
17 req.Quantity)
18          executedOrders++
19       }
20    }…
```

Клиент

```
1  stream, err := client.PostOrder(context.Background())
2  if err != nil {
3     log.Fatalf("can't post order: %v", err)
4  }
5
6  for i := 0; i < 10; i++ {
7     order := orderspb.OrderRequest{
8        Price:    float64(rand.Intn(1000)),
9        Quantity: int64(rand.Intn(10)),
10    }
11    stream.Send(&order)
12    time.Sleep(time.Second)
13 }
14
15 resp, err := stream.CloseAndRecv()
16 if err != nil {
17    log.Fatalf("can't get response from server: %v", err)
18 }
19 fmt.Printf("executed orders: %d", resp.ExecutedOrders)
```

Output на сервер и клиенте

Сервер

```
executed order with price 694.00 and quantity 1
executed order with price 728.00 and quantity 4
executed order with price 211.00 and quantity 5
executed order with price 237.00 and quantity 6
```

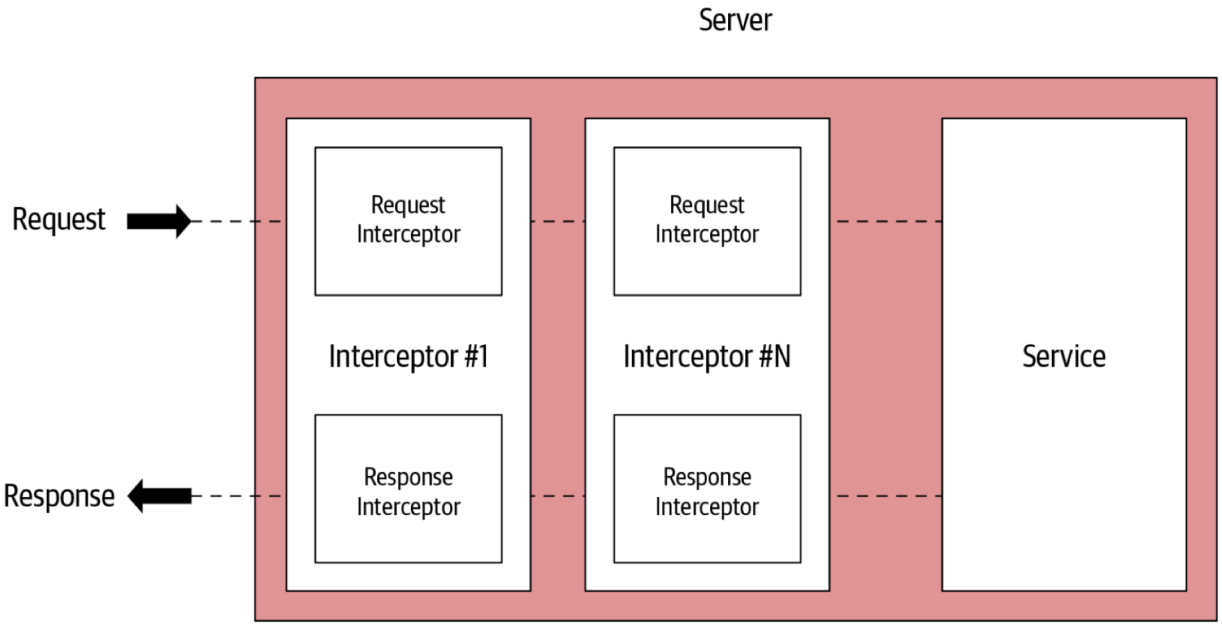
Клиент

```
executed orders: 4
```

gRPC
(чуть глубже)

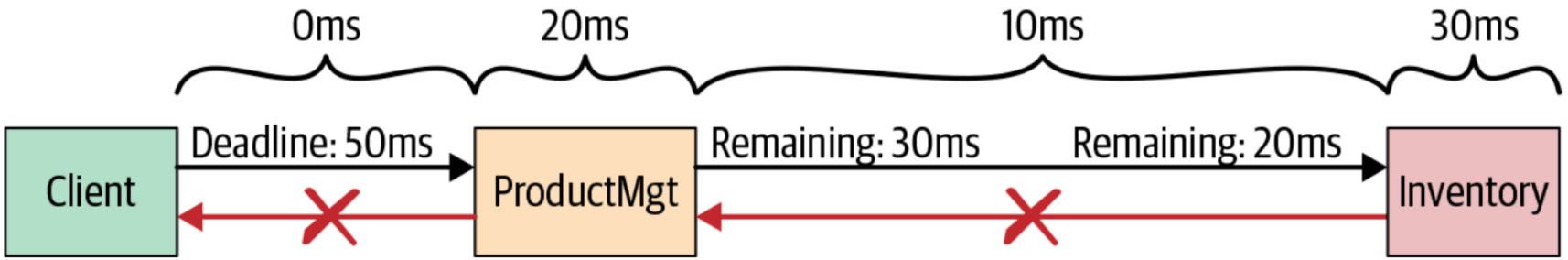
Interceptor

https://github.com/grpc/grpc-go/tree/master/examples/features/interceptor



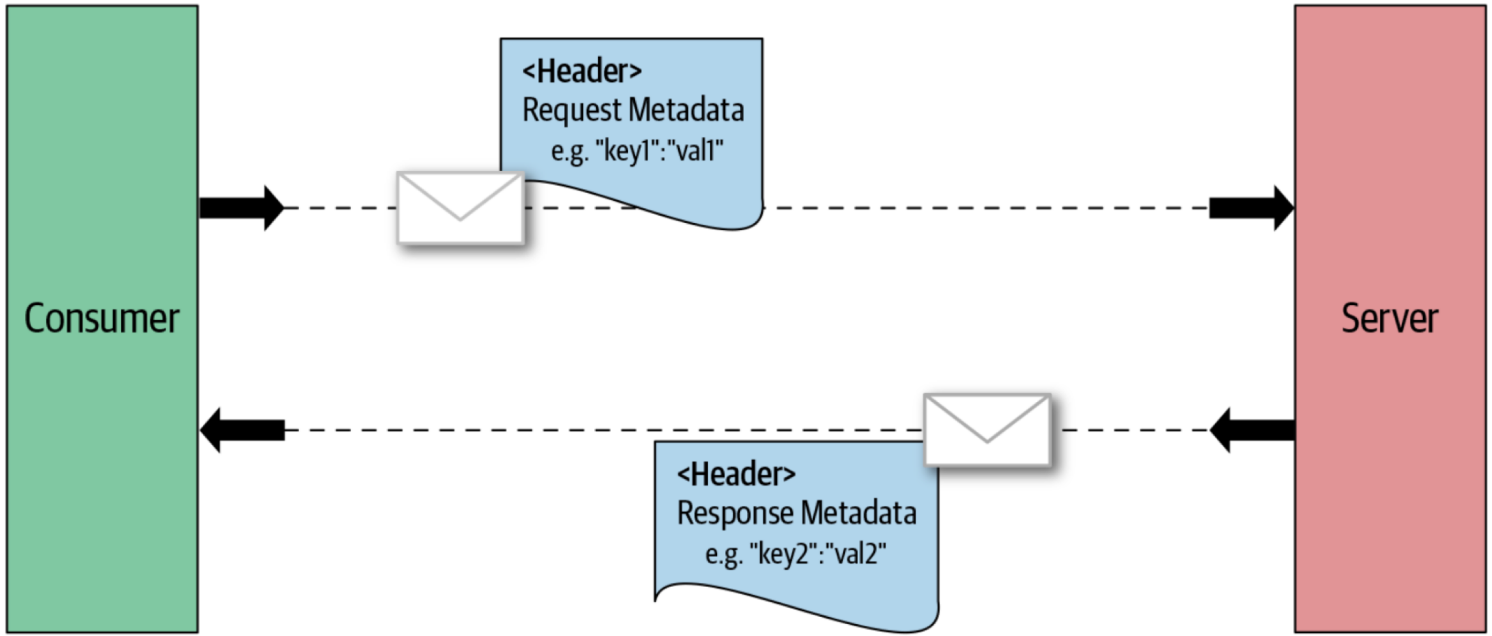
Deadlines

https://grpc.io/blog/deadlines/



Metadata

https://github.com/grpc/grpc-go/blob/master/Documentation/grpc-metadata.md



Обратная связь

Спасибо за внимание

Tinkoff.ru