Разработка веб-сервисов на Go, часть 2. Лекция 3

Что такое микросервис

Что такое микросервис, плюсы и минусы

В этой лекции мы поговорим про микросервисы. Микросервисы это популярный в последнее время паттерн для упрощения организации програмы. Для начала давайте разберемся, что такое монолит, и чем он отличается от микросервиса. Если у вас есть одна программа, которая ходит во все базы данных и делает все-все, то есть у вас есть кластер серверов, на который катится только одна программа, это называется монолит. Если же программа не большая, и делает ограниченное количество вещей, но все-таки более-менее разных, это можно назвать просто сервис. А микросервис — это очень маленькая программа, то есть в пределах от нескольких сотен до нескольких тысяч строк, и делает она только одну вещь. Для того чтобы понять, почему вообще стоит разбивать программу на микросервисы, нужно разобрать проблемы монолита.

Первая проблема монолита — это сложность. Когда ваша кодовая база переваливает за несколько миллионов строк, разобраться становится довольно тяжело. Если вам нужно подправить в одном месте, довольно часто бывает, что в монолите может стрельнуть где-то в другом месте, которое вроде бы никак не должно затрагиваться. И опять-таки код приходится выкатывать весь вместе, и чем больше изменений, тем больше вероятность, что придется откатить какие-то другие изменения из-за того, что где-то что-то в одном месте сломалось, критичное для работы вашего сервиса.

Еще одной проблемой монолита является то, что все-таки рано или поздно появляется другой сервис. Например, старый был написан на php, новый вы пишете на Go. И вам надо дублировать код, вам надо дублировать вашу бизнес-логику, хотя хорошо бы этого не делать. Либо же еще худший случай — вам нужно копипэйстить один и тот же код в разные места. Если вдруг вам нужно где-то что-то поменять, вам нужно будет обновлять везде все. Поэтому стараются следовать принципу «разделяй и властвуй».

Как обычно разделяют монолит на несколько сервисов? Вначале есть какая-то просто гора кода. И если мы хотим оттуда выделить микросервис, для начала мы должны как-то организовать нашу программу. В случае, если у нас правильная архитектура, это хорошо. У нас и так уже все изолировано, и общих пересечений совсем нету. Итак, мы изолируем наш отдельный сервис. То есть у нас, по сути, монолит, но такой, микросервисный, монолит, то есть когда есть несколько отдельных, изолированных кусков кода, которые выступают как отдельные сервисы. Но просто внутри вот одной общей кучи. Соответственно, когда мы выделили такой микросервис, такой сервис внутри вашего монолита, мы можем просто выделить его в отдельную программу. В монолите же мы можем оставить только интерфейс похода в наш микросервис. В идеале это будет выглядеть примерно вот так: у нас есть некоторое количество умеренно больших сервисов, где сосредоточена наша основная бизнес-логика. Есть такое же небольшое количество микросервисов, которые следуют принципу единой ответственности и делают одну небольшую вещь, но для всех клиентов к этому микросервису.

Однако тут надо не переусердствовать и не получить кучу-малу уже не внутри одной программы, а разнесенную на несколько программ. То есть микросервисы не нужно ставить во главу угла, не надо начинать разработку программы с микросервисов оттого, что вы будете их создавать 20-30 штук. Потому

что почему микросервис — это не всегда хорошо. Во-первых, у микросервисов есть накладные расходы на коммуникацию. Когда у вас ваш, давайте будем оставлять это — монолит, микросервис и база данных. Раньше монолит ходил сразу в базу, и все было хорошо. А потом монолит начал ходить в микросервис, то есть у вас получилось один шаг накладных расходов. В случае, если у вас нестабильная сеть, либо запросов много — это может стать проблемой. Также, если раньше вы катили одну программу, один бинарник, или набор кода, то теперь вам придется катить много программ. То есть ваша сложность может перекочевать просто из одного места в другое. Вы не решите так своих проблем. Ну и надо, наконец, понимать, что микросервис — это не панацея. Они не решат всех ваших проблем. Если ваши проблемы не в том, что у вас монолит, где что-то много стреляет, где бывают ошибки из-за того, что стрельнуло в одном месте, а какие-то организационные проблемы, либо проблема именно с написанием кода: там много ошибок, нет тестирования, то микросервисы, не факт, что решат эти проблемы. Поэтому об этом не надо забывать.

В следующих разделах мы рассмотрим, каким образом в коде можно выделить сервис и как-то изолировать его, а потом вынести в отдельный микросервис. Рассморим, какие инструменты Go предоставляет для этого. Рассмотрим встроенный пакет для RPC — вызов удаленных процедур, для JSON RPC, и рассмотрим большую библиотеку для организации микросервисов под названием gRPC.

Микросервисы внутри монолита

В начале был монолит. Потом этот монолит разбили на модули, которые стали автономными кусками программы, и после чего пришел XXI век, появился паттерн микросервисов, и эти модули вынесли в микросервисы.

В этом разделе мы рассмотрим пример, как можно немножко разбить старый код. У нас есть код, в котором есть три функции: создать сессию, проверить сессию и удалить сессию.

```
func main() {

        // создаем сессию
        sessId, err := AuthCreateSession(
                &Session{
                        Login:     "rvasily",
                        Useragent: "chrome",
                })
        fmt.Println("sessId", sessId, err)

        // проеряем сессию
        sess := AuthCheckSession(
                &SessionID{
                        ID: sessId.ID,
                })
        fmt.Println("sess", sess)

        // удаляем сессию
        AuthSessionDelete(
                &SessionID{
                        ID: sessId.ID,
                })

        // проверяем еще раз
        sess = AuthCheckSession(
                &SessionID{
                        ID: sessId.ID,
                })
        fmt.Println("sess", sess)

}
```

Представьте, что писался он в стародавние времена, когда компьютеры были большими и медленными. И было какое-то глобальное хранилище сессий, какой-то глобальный Mutex, который находится в глобальном пространстве вообще, и все плохо.

```
var (
        sessions = map[SessionID]*Session{}
        mu       = &sync.RWMutex{}
)

func AuthCreateSession(in *Session) (*SessionID, error) {
        mu.Lock()
        id := SessionID{RandStringRunes(sessKeyLen)}
        mu.Unlock()
        sessions[id] = in
        return &id, nil
}

func AuthCheckSession(in *SessionID) *Session {
        mu.RLock()
        defer mu.RUnlock()
        if sess, ok := sessions[*in]; ok {
                return sess
        }
        return nil
}

func AuthSessionDelete(in *SessionID) {
        mu.Lock()
        defer mu.Unlock()
        delete(sessions, *in)
}
```

И в каждой функции мы лочили каждый раз на Mutex и делали что-то с хранилищем. Для того чтобы осовременить код, его стоит вынести в отдельный модуль, чтобы он был как-то взаимозаменяем, или реализацию можно было как-то подменить.

Для начала выкинем глобальные переменные и создадим вместо них структуру, добавив отдельный helper, чтобы её было удобно создавать.

```
type Session struct {
        Login     string
        Useragent string
}

type SessionID struct {
        ID string
}

type SessionManager struct {
        mu       *sync.RWMutex
        sessions map[SessionID]*Session
}
```

А теперь функции, которые были, сделаем методами этой структуры.

```
func (sm *SessionManager) Create(in *Session) (*SessionID, error) {
        sm.mu.Lock()
        id := SessionID{RandStringRunes(sessKeyLen)}
        sm.mu.Unlock()
        sm.sessions[id] = in
        return &id, nil
}

func (sm *SessionManager) Check(in *SessionID) *Session {
        sm.mu.RLock()
        defer sm.mu.RUnlock()
        if sess, ok := sm.sessions[*in]; ok {
```

```
                return sess
        }
        return nil
}

func (sm *SessionManager) Delete(in *SessionID) {
        sm.mu.Lock()
        defer sm.mu.Unlock()
        delete(sm.sessions, *in)
}
```

Теперь у нас есть уже как-то более-менее изолированный модуль, который мы можем переиспользовать. Но модуль — это конечно очень круто, но вдруг мы захотим какие-то тесты. Поэтому можем сразу создать интерфейс.

```
type SessionManagerI interface {
        Create(*Session) (*SessionID, error)
        Check(*SessionID) *Session
        Delete(*SessionID)
}
```

Кроме того напишем функцию, которая создает структуру.

```
func NewSessManager() *SessionManager {
        return &SessionManager{
                mu:       &sync.RWMutex{},
                sessions: map[SessionID]*Session{},
        }
}
```

Теперь нужно отрефакторить код, который все это вызывает. Создадим переменную типа interface, сделав её глобальной, если захотим как-то переиспользовать. Вызовем функцию, которая создает объект. Наконец, переведем вызовы функций на вызовы методов этой структуры.

```
var sessManager SessionManagerI

func main() {

        sessManager = NewSessManager()

        // создаем сессию
        sessId, err := sessManager.Create(
                &Session{
                        Login:     "rvasily",
                        Useragent: "chrome",
                })
        fmt.Println("sessId", sessId, err)

        // проеряем сессию
        sess := sessManager.Check(
                &SessionID{
                        ID: sessId.ID,
                })
        fmt.Println("sess", sess)

        // удаляем сессию
        sessManager.Delete(
                &SessionID{
                        ID: sessId.ID,
                })

        // проверяем еще раз
        sess = sessManager.Check(
```

```
                &SessionID{
                        ID: sessId.ID,
                })
        fmt.Println("sess", sess)

}
```

Итак, что мы сделали? Раньше у нас была глобальная переменная, где мы хранили все, и были какие-то функции, для того чтобы работать с этой глобальной переменной. Функция — это уже неплохо. Иногда бывает так, что все прямо работает с этой переменной. За счет того, что у нас были функции, мы хотя бы смогли это изолировать. Теперь, если вдруг мы захотим подменить теперь эту структуру, заменить реализацию, нам будет гораздо проще это делать. У нас есть вроде как отдельный небольшой модуль — сессия, и его уже можно попробовать изолировать в виде отдельного сервиса, чем мы и займемся дальше.

Делаем микросервис руками

net/rpc и формат gob

В предыдущем разделе мы немножко порефакторили код так, что у нас получился модуль для работы с сессиями. В этом разделе мы рассмотрим, каким образом можно вынести его в отдельный микросервис.

Очень часто микросервисы основаны на подходе RPC — Remote Procedure Call. По-русски это значит «удалённый вызов процедур». Этот подход отличается тем, что код оформляется так, что вы вызываете какую-то удалённую процедуру так, словно она локальная. То есть через обертку над вызовом удалённого метода. В качестве примера рассмотрим модуль стандартной библиотеки, который называется net/rpc. Он достаточно беден по функционалу. Пожалуй, его лучше не использовать в продакшене. Однако для учебных целей он подойдёт. Рассмотрим этот код.

```
package main

import (
        "fmt"
        "log"
        "net"
        "net/http"
        "net/rpc"
)

func main() {
        sessManager := NewSessManager()

        rpc.Register(sessManager)
        rpc.HandleHTTP()

        l, e := net.Listen("tcp", ":8081")
        if e != nil {
                log.Fatal("listen error:", e)
        }

        fmt.Println("starting server at :8081")
        http.Serve(l, nil)
}
```

Мы получаем менеджер сессии, как в прошлый раз. Теперь в модуле RPC мы регистрируем нашу структуру для управления сессиями. Говорим модулю, чтоб он начинал обрабатывать http, слушаем порт и всё. Микросервис почти готов, то есть сейчас его можно уже почти можно запустить и обращаться. Однако net/rpc требует небольших правок в организации методов а именно, чтобы было ровно два параметра, а возвращалась только ошибка. Первый параметр — с входящими аргументами функции, а второй — это адрес, куда запишется результат.

```
func (sm *SessionManager) Create(in *Session, out *SessionID) error {
        fmt.Println("call Create", in)
```

```
        id := &SessionID{RandStringRunes(sessKeyLen)}
        sm.mu.Lock()
        sm.sessions[*id] = in
        sm.mu.Unlock()
        *out = *id
        return nil
}

...
```

Теперь каким-то образом нужно вызвать микросервис. Посмотрим модуль сессии, который остался в клиентской части.

```
type SessionManagerI interface {
        Create(*Session) (*SessionID, error)
        Check(*SessionID) *Session
        Delete(*SessionID)
}

type SessionManager struct {
        client *rpc.Client
}

func NewSessionManager() *SessionManager {
        client, err := rpc.DialHTTP("tcp", "localhost:8081")
        if err != nil {
                log.Fatal("dialing:", err)
        }

        return &SessionManager{
                client: client,
        }
}
```

У нас по-прежнему есть интерфейс SessionManager'a, у которого вызываются нужные нам методы. Однако сама структура претерпела изменения. Раньше в ней было уже хранилище сессии, а теперь — ссылка на rpc.Client. В функции получения менеджера мы соединяемся с удалённым сервером, и уже клиента возвращаем; то есть клиент поменялся.

Соответственно, поменялись и функции, через которые мы обращаемся к удалённым процедурам.

```
func (sm *SessionManager) Create(in *Session) (*SessionID, error) {
        id := new(SessionID)
        err := sm.client.Call("SessionManager.Create", in, id)
        if err != nil {
                fmt.Println("SessionManager.Create error:", err)
                return nil, nil
        }
        return id, nil
}

...
```

Раньше мы шли в мапку и чего-то проверяли, теперь нам нужно сначала создать переменную, в которую запишется результат, если он будет; и потом вызвать удалённую процедуру у SessionManager'a — структуры, зарегистрированной в обработчике. Метод Create вызовется уже на той стороне. Туда передаем входящие параметры и указываем, куда записать результат. То же самое для проверки и для удаления.

Итак, нам пришлось внести небольшие изменения в сигнатуры наших методов, чтобы они поддерживались библиотекой net/rpc, и пришлось сделать реализацию для вызова удалённых процедур в клиентской части библиотеки. При этом сам клиентский код для работы с сессиями абсолютно не поменялся.

Net/rpc работает на основе формата gob; это встроенный формат сериализации языка Go. Он не очень

популярен. Мы не будем останавливаться на том, как он работает. Вместо этого рассмотрим вариацию модуля net/rpc, которая называется JSON-RPC и оперирует уже форматом json.

net/http/jsonrpc

Мы только что рассмотрели, каким образом можно сделать базовый микросервис, используя пакет net/rpc. Однако он использует формат gob, что является, скорее, недостатком, потому что искать его для другого языка программирования то еще удовольствие. И плюс придется на бинарный формат накручивать какую-то авторизацию и еще какой-то сервис. Это не всегда удобно.

Есть достаточно популярный формат jsonrpc, который описывает удаленный вызов процедур в json формате. Например, вот так это выглядит. Указываем, версию jsonrpc, номер запроса, метод, который хотим вызвать, и параметры, которые туда передаем.

```
{
  "jsonrpc":"2.0",
  "id":1,
  "method":"SessionManager.Create",
  "params":[
      {
        "login":"rvasily",
        "useragent":"chrome"
      }
  ]
}
```

Хотелось бы, чтобы наш микросервис умел парсить такое и отвечать опять-таки в jsonrpc формате. В Go есть специальные коды для этого, Codec для net/rpc, которые позволяют организовать это. Давайте рассмотрим, как это реализовать, на примере. Мы не будем делать обслуживание чисто на основе net/rpc, а воспользуемся стандартным пакетом net/http, чтобы можно было обрабатывать и другие запросы или добавить больше логики. Вызывать микросервис мы будем в данном примере через curl без клиента, потому что мы хотим проверить, что мы можем отсылать json из других клиентов.

Сначала посмотрим создание сервера.

```
func main() {
        sessManager := NewSessManager()

        server := rpc.NewServer()
        server.Register(sessManager)

        sessionHandler := &Handler{
                rpcServer: server,
        }
        http.Handle("/rpc", sessionHandler)

        fmt.Println("starting server at :8081")
        http.ListenAndServe(":8081", nil)

}
```

Мы по-прежнему создаем наш sessionManager. Теперь мы получаем сервер из пакета rpc (в прошлом примере мы пользовались стандартным). Регистрируем в нем нашу структуру, чтобы вызывать ее методы, Если вы помните, у структуры может быть метод ServeHTTP, который будет обрабатывать запросы. Этим мы и воспользуемся. Создадим структуру, куда передадим наш сервер в качестве поля, и теперь начнем обрабатывать запросы, которые приходят на url rpc. Что происходит дальше?

```
type Handler struct {
        rpcServer *rpc.Server
}

func (h *Handler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
        fmt.Println("rpc auth: ", r.Header.Get("X-Auth"))
```

```
        serverCodec := jsonrpc.NewServerCodec(&HttpConn{
                in:  r.Body,
                out: w,
        })
        w.Header().Set("Content-type", "application/json")
        err := h.rpcServer.ServeRequest(serverCodec)
        if err != nil {
                log.Printf("Error while serving JSON request: %v", err)
                http.Error(w, `{"error":"cant serve request"}`, 500)
        } else {
                w.WriteHeader(200)
        }
}
```

Здесь приведена структура и метод ServeHTTP, в нем мы можем обращаться уже к хедерам, которые приходят в http запросы. В данном случае проверяем хедер X-Auth, для того чтобы авторизоваться на этом сервисе. То есть как минимум есть возможность получить доступ к чему угодно из запроса. Далее воспользуюемся небольшим хелпером, который реализует нужный интерфейс, для того чтобы передать его в Codec для json rpc. При помощи Codec'а можно распаковывать и запаковывать запрос. Далее записываем, что хедер вернем в json. Парсимся к своему rpc сервису, передам туда Codec, и уже сервер оттуда вычитает все, что нужно, при помощи jsonrpc, а Codec распакует, запакует и запишет результат. Далее идет обработка ошибок. Если что-то пошло не так, то мы напечатаем в лог либо мы выведем клиенту какую-то ошибку опять-таки в формате json, чтобы он не сходил с ума. И в случае успеха мы запишем Header.

Теперь надо запускать. Для начала создим пользователя, создим сессию пользователя.

```
curl -v -X POST -H "Content-Type: application/json" -H "X-Auth: 123" -d
'{"jsonrpc":"2.0", "id": 1, "method": "SessionManager.Create", "params":
[{"login":"rvasily", "useragent": "chrome"}]}'
http://localhost:8081/rpc
```

Обратите внимание, это просто curl запрос, шлем его постом. Говорим, что в посте json, ставим хедер X-Auth. И в json непосредственно идут все данные, которые нужно передать. Шлем на адрес rpc. Попробуйте сами создать сессию, получить сессию и спровоцировать ошибку.

protobuf и gRPC

Формат сериализации protobuf

Прежде чем двигаться дальше, поговорим о форматах сериализации. В предыдущем примере мы активно использовали json. В примере с net/rpc упоминалось, что он использует формат gob. В чем разница, и почему один формат выгоднее или невыгоднее другого? Сначала давайте рассмотрим json. Вот код, который сериализует уже знакомую вам структуру Session.

```
{"login":"rvasily","useragent":"Chrome"}
```

В json эта структура занимает 40 байт, хотя данных там вдвое меньше. Дело в том, что json — это формат без схемы, схему он сериализует в виде имен полей. Соответственно, нужно каждый раз сначала распаковать поле, после этого распаковать тип данных. Причем нужно проверять каждый символ, не является ли он кавычкой, например, которая закрывает строку, или переносом строки или другим символом, который нужно экранировать. То есть формат json распаковать несколько накладнее, чем бинарный формат. И он больше по размеру, однако его плюсом является то, что он человекочитаем. То есть можно открыть логи либо какой-то текстовый файл и посмотреть глазами, что там есть.

Бинарный формат, как следует из названия, упакован гораздо плотнее, он представлен не в текстовом виде, а в виде бинарных данных. За счет этого он гораздо легче в распаковке и может быть легче в упаковке. Формат gob мы рассматривать не будем, потому что он специфичен для языка Go и не очень широко распространен за его пределами. Мы будем рассматривать формат protobuf, — формат, который разработан в компании Google. Сейчас он достаточно широко распространен. Упаковка в этот формат и распаковка очени похожи на json, выполняются тоже с помощью метода Marshal.

```
dataJson, _ := json.Marshal(sess)
dataPb, _ := proto.Marshal(sess)
```

Если в json наша структура занимает 40 байт, то в protobuf всего 17 байт, что более, чем в два раза меньше, чем json. Это происходит потому, что protobuf — это формат с жестко заданной схемой. То есть в json может быть любое количество полей, и, например, в распаковке структур вы просто возьмете себе нужные. С protobuf так не совсем получится, вам нужна схема, вам нужен какой-то декодировщик этих полей. Но сначала посмотрим, что лежит в этих 17 байтах.

```
[10 7 114 118 97 115 105 108 121 18 6 67 104 114 111 109 101]
```

```
10 // номер поля + тип
7  // длина данных
       114 118 97 115 105 108 121
18 // номер поля + тип
6  // длина данных
       67 104 114 111 109 101
```

Сначала идет байт 10, в нем запакован номер поля и его тип. После этого — у нас строковый тип — идет длина данных и сами данные. Потом следующее поле, там тоже номер поля, вместе с типом, которые упакованы в один байт, и опять-таки длина данных 6 байт. То есть для того чтобы, например, распаковать строку, не нужно проходиться по каждому символу, достаточно просто копировать 6 байт, потому что гарантируется, что там уже все хорошо, что там нет никаких проблем с экранированием или с внезапным концом строки.

Каким образом используется этот формат, каким образом задается схема? Формат protobuf описывается при помощи промежуточного языка EDL, который описывает схему этого формата.

```
syntax = "proto3";

package main;

message SessionID {
  string ID = 1;
}

message Session {
  string login = 1;
  string useragent = 2;
}
```

В данном случае у нас есть message SessionId и message Session, в Go они переводятся в структуры. Внутри задаем тип, имя и номер для каждого поля. Обратите внимание, поля пакуются в виде номеров, не в виде имени поля, как, например, в json. За счет этого получается очень компактно. Какие-то поля потом можно пропустить или, например, добавить. То есть формат расширяем, если каких-то полей в нем не будет, они будут null. После того, как сема написана, нужно запустить внешний генератор кода, который сгенерирует вам в случае с Go все структуры и необходимый код для их десериализации. Сделать это можно с помощью следующей команды.

```
protoc --go_out=. *.proto
```

После генерации получается файл, в котором довольно много служебного кода, но руками его трогать никогда не надо, потому что в следующий раз он перезапишется. В нем будут нужные структуры и вспомогательные методы: сброс, привести к строке, получить сообщение, описание схемы и getter'ы для каждого из полей.

Вроде кажется не очень удобно, каждый раз генерировать схему, однако по этой схеме можно сгенерировать код для других языков, не только для Go, но и для плюсов, для php, для json, для java, для python. Есть кодогенерация практически подо всё.

Protobuf очень удобный формат, когда у вас очень много сообщений и вы хотите сэкономить хотя бы на их сериализации, десериализации. Формат protobuf используется в gRPC как основной формат передачи данных, который мы будем рассматривать дальше. Следует заметить, что protobuf, конечно же, не единственный формат в бинарной сериализации, который распространен. Еще есть MessagePack и формат

AVR. Но мы их рассматривать не будем, а сосредоточимся на protobuf. Далее перейдем уже к gRPC, посмотрим, как при помощи protobuf можно сгенерировать весь нужный код и вдобавок ко всем структурам сгенерировать сразу еще и сервис.

Делаем сервис на gRPC

Ранее мы рассмотрели возможность выноса какого-то кода в микросервисы, с использованием стандартной библиотеку и пакетов net/rpc и json/rpc. Однако эти пакеты довольно бедные, там очень мало фич; например, там сложности с авторизацией, балансингом и прочим. В этом видео мы поговорим про фреймворк GRPC, который значительно богаче на возможности, чем предыдущие варианты, и основан на формате protobuf, который мы рассмотрели в предыдущем разделе.

Сначала давайте посмотрим на сервис. У нас уже есть знакомый вам protobuf-файл, где описаны структуры SessionID и session. Однако добавляется ещё описание сервиса.

```
syntax = "proto3";

package session;

message SessionID {
  string ID = 1;
}

message Session {
  string login = 1;
  string useragent = 2;
}

message Nothing {
  bool dummy = 1;
}

// grpc-сервис проверки авторизации
service AuthChecker {
    rpc Create (Session) returns (SessionID) {}
    rpc Check (SessionID) returns (Session) {}
    rpc Delete (SessionID) returns (Nothing) {}
}
```

Message описывает структуру, сервис описывает GRPC-шный service, называется он AuthChecker. В нем описано несколько методов: Create, Check и Delete; для Delete'а сделана заглушка в качестве возвращаемого ответа, потому что GRPC не может не возвращать ответ. Сделано это для того, чтобы когда вы будете расширять сервис, у вас не возникло трудностей с тем, чтобы добавлять туда что-то.

Итак, для генерации кода самого сервиса необходимо указать плагины, что мы хотим не только сгенерировать сам protobuf-файл, но и сгенерировать GRPC-обвязку для сервисов, которые там есть.

```
protoc --go_out=plugins=grpc:. *.proto
```

После запуска получается здоровый файл, который, ещё раз повторимся, не надо править, потому что при следующей генерации он все равно перезапишется. Там есть структуры, которые вы в целом уже видели, а теперь там добавляются уже следующие сервисы. Есть интерфейс для клиента, который вы будете вызывать и создавать со стороны клиента реализацию этого сервиса.

```
type AuthCheckerClient interface {
        Create(ctx context.Context, in *Session, opts
        ...grpc.CallOption) (*SessionID, error)
        Check(ctx context.Context, in *SessionID, opts
        ...grpc.CallOption) (*Session, error)
        Delete(ctx context.Context, in *SessionID, opts
        ...grpc.CallOption) (*Nothing, error)
}
```

То есть вот интерфейс, а вот реализация.

```
type authCheckerClient struct {
        cc *grpc.ClientConn
}
```

Реализация внутренняя, не публичная, за пределы пакета не выходит; получается она вот из этой функции.

```
func NewAuthCheckerClient(cc *grpc.ClientConn) AuthCheckerClient {
        return &authCheckerClient{cc}
}
```

позднее рассмотрим, как это использовать. И есть вызов.

```
func (c *authCheckerClient) Create
     (ctx context.Context, in *Session, opts ...grpc.CallOption)
     (*SessionID, error) {
        out := new(SessionID)
        err := grpc.Invoke(ctx, "/session.AuthChecker/Create",
                                        in, out, c.cc, opts...)
        if err != nil {
                return nil, err
        }
        return out, nil
}

...
```

Этот код уже сгенерирован для того, чтобы вызвать на той стороне нужную нам функцию в нужном пакете.

Кроме клиента есть сервер.

```
type AuthCheckerServer interface {
        Create(context.Context, *Session) (*SessionID, error)
        Check(context.Context, *SessionID) (*Session, error)
        Delete(context.Context, *SessionID) (*Nothing, error)
}
```

То есть GRPC-генератор сервиса сразу генерирует вам интерфейс, в котором жёстко заданы входящие параметры и исходящие параметры. И вам нужно будет только реализовать структуру, методы которой удовлетворяют этому интерфейсу.

Теперь давайте посмотрим на реализацию. Реализация мало отличается от того, что мы видели в самом первом варианте.

```
type SessionManager struct {
        mu       sync.RWMutex
        sessions map[session.SessionID]*session.Session
}

func NewSessionManager() *SessionManager {
        return &SessionManager{
                mu:       sync.RWMutex{},
                sessions: map[session.SessionID]*session.Session{},
        }
}
```

У нас есть Mutex, хранилище сессии в виде мапки. NewSessionManager возвращает нам указатель на эту структуру.

Методы.

```
func (sm *SessionManager) Create(ctx context.Context, in *session.Session)
     (*session.SessionID, error) {
        fmt.Println("call Create", in)
```

```
        id := &session.SessionID{RandStringRunes(sessKeyLen)}
        sm.mu.Lock()
        sm.sessions[*id] = in
        sm.mu.Unlock()
        return id, nil
}

func (sm *SessionManager) Check(ctx context.Context, in *session.SessionID)
     (*session.Session, error) {
        fmt.Println("call Check", in)
        sm.mu.RLock()
        defer sm.mu.RUnlock()
        if sess, ok := sm.sessions[*in]; ok {
                return sess, nil
        }
        return nil, grpc.Errorf(codes.NotFound, "session not found")
}

func (sm *SessionManager) Delete(ctx context.Context, in *session.SessionID)
     (*session.Nothing, error) {
        fmt.Println("call Delete", in)
        sm.mu.Lock()
        defer sm.mu.Unlock()
        delete(sm.sessions, *in)
        return &session.Nothing{Dummy: true}, nil
}
```

Отличие от самого-самого первого варианта в методах в том, что теперь первым аргументом принимается context. В нём могут быть какие-то дополнительные параметры; либо можно получить какую-то информацию о запросе, например, вход, с которого он пришёл. В остальном реализация практически не отличается от первоначального варианта. В Create получаем id'шник и кладём в мапку. В Check'е пытаемся проверить из мапки. Если нет, возвращаем ошибку со статусом «Не найдено». Удаление тоже ничем не отличается.

Теперь надо создать сервер. В базовом варианте это выглядит вот так.

```
func main() {
        lis, err := net.Listen("tcp", ":8081")
        if err != nil {
                log.Fatalln("cant listet port", err)
        }

        server := grpc.NewServer()

        session.RegisterAuthCheckerServer(server, NewSessionManager())

        fmt.Println("starting server at :8081")
        server.Serve(lis)
}
```

Создаем Listener на 8081 порту. Создаем новый GRPC-шный сервис, регистрируем в нем наш сервис и — всё. Следует обратить внимание только на то, что метод RegisterAuthCheckerServer вызываем из нашего сгенерированного пакета session.

Как теперь это вызывать? Вот код, который будет вызывать методы микросервиса.

```
func main() {

        grcpConn, err := grpc.Dial(
                "127.0.0.1:8081",
                grpc.WithInsecure(),
        )
```

```
        if err != nil {
                log.Fatalf("cant connect to grpc")
        }
        defer grcpConn.Close()

        sessManager := session.NewAuthCheckerClient(grcpConn)

        ctx := context.Background()

        // создаем сессию
        sessId, err := sessManager.Create(ctx,
                &session.Session{
                        Login:     "rvasily",
                        Useragent: "chrome",
                })
        fmt.Println("sessId", sessId, err)

        // проеряем сессию
        sess, err := sessManager.Check(ctx,
                &session.SessionID{
                        ID: sessId.ID,
                })
        fmt.Println("sess", sess, err)

        // удаляем сессию
        _, err = sessManager.Delete(ctx,
                &session.SessionID{
                        ID: sessId.ID,
                })

        // проверяем еще раз
        sess, err = sessManager.Check(ctx,
                &session.SessionID{
                        ID: sessId.ID,
                })
        fmt.Println("sess", sess, err)
}
```

Для начала нам к нему нужно подключиться; указываем адрес и grpc-опцию WithInsecure. Это значит, что там нет никакого шифрования, голые бинарные данные по TCP гоняем. Обязательное закрытие через defer. Далее создаем клиента к своему сервису; опять-таки пользуюясь сгенерированным кодом, в котором уже есть все нужные методы, достаточно просто передать соединение. Уже готов connect и готов клиент к менеджеру сессии на той стороне, который крутится внутри микросервиса. Теперь это нужно вызвать. Как уже упоминалось, добавился context; в данном случае берем context Background, однако можно использовать, например, context с Timeout'ом или передать туда какие-то дополнительные опции. Вызываем Create. Опять-таки это очень близко к тому, что вы видели первоначально; единственное, что добавилось — context. После создания проверяем сессию: опять всё то же самое плюс context. Единственное, что я обращаемся не к структуре в данном пакете, а к структуре в другом пакете, добавляется префикс к session. Затем идут удаление и еще одна проверка. Попробуйте сами это запустить.

gRPC успешно используется в продакшене, благодаря производительности и набору возможностей, которые подробнее рассмотри далее.

Продвинутая работа с gRPC

Мы продолжаем рассматривать те возможности, которые нам предоставляет grpc. Ранее мы научились дергать удаленные функции. Однако мы не делали ни проверок авторизации, ни логирования,ни timing'а. Хотелось бы научиться все это делать. Что grpc может нам предложить для этого? У grpc есть возможность делать UnaryInterceptor. UnaryInterceptor — это перехватчик одиночных запросов, по сути это как раз таки MiddleWare, который будет выполняться, перед тем как запрос уйдет на удаленный сервер. Также в grpc есть возможность передачи метаданных, как с запросом на сервер, так и со стороны сервера. То

есть это какие-то данные, которые приходят помимо самого запроса, помимо самого результата удаленной функции. Зачем это нужно? В частности, например, чтобы передать токен, то есть какие-то общесистемные вещи, которые относятся не непосредственно к бизнес-логике самого удаленного сервиса, а скорее к движку.

В этом разделе мы на примерах рассмотрим, как это делать. Для начала UnaryInterceptor.

```
func main() {

        grcpConn, err := grpc.Dial(
                "127.0.0.1:8081",
                grpc.WithUnaryInterceptor(timingInterceptor),
                grpc.WithPerRPCCredentials(&tokenAuth{"100500"}),
                grpc.WithInsecure(),
        )
...
```

При подключении к grpc можно указать n-е количество опций. Указываем UnaryInterceptor и в скобках функцию. Вот код этой функции, сюда передается куча всяких параметров: контекст, метод, запрос, ответ, коннект, invoker — это как раз та функцию, которую нужно вызвать.

```
func timingInterceptor(
        ctx context.Context,
        method string,
        req interface{},
        reply interface{},
        cc *grpc.ClientConn,
        invoker grpc.UnaryInvoker,
        opts ...grpc.CallOption,
) error {
        start := time.Now()
        err := invoker(ctx, method, req, reply, cc, opts...)
        fmt.Printf(`--
        call=%v
        req=%#v
        reply=%#v
        time=%v
        err=%v
`, method, req, reply, time.Since(start), err)
        return err
}
```

Что мы делаем? Мы просто оборачиваем все в тайминг и выводим всю сопутствующую информацию, которая, возвращаем ошибку. Интерсептор очень простой на стороне клиента. Очень похоже на то, что мы рассматривали в MiddleWare.

Теперь метаданные. Метаданные можно передавать несколькими способами. Первый способ передать общие метаданные ко всем запросам — это установить PerRPCCredentials (смотри main выше). PerRPCCredentials принимает в себя интерфейс, который можно реализовать. В данном случае у нас есть авторизация по токену — то есть мы создаем структуру, в ней используем токен.

```
type tokenAuth struct {
        Token string
}
```

Когда будет происходить вызов, будет запрашиваться GetRequestMetadata.

```
func (t *tokenAuth) GetRequestMetadata(context.Context, ...string)
     (map[string]string, error) {
        return map[string]string{
                "access-token": t.Token,
        }, nil
}
```

Из неё возвращаем map, где как раз указываем токен. И еще один метод, который запрашивает информацию: нужна ли для этого способа авторизация с шифрованием. Должно ли она работать по TLS/SSL.

```
func (c *tokenAuth) RequireTransportSecurity() bool {
        return false
}
```

В данном примере говорим, что шифрование не нужно. То есть этот метод будет вызываться перед тем, как мы пошлем запрос, и данные, в данном случае access-token, будут каждый раз добавляться к каждому запросу.

Что делать, если нужно указывать запросы для каждого вызова? Например, есть request ID, по которому надо отследить вызов процедур, чтоб узнать, что было не так. Для этого тоже есть способ: метаданные передаются через контекст.

```
ctx := context.Background()
        md := metadata.Pairs(
                "api-req-id", "123",
                "subsystem", "cli",
        )
ctx = metadata.NewOutgoingContext(ctx, md)
```

Для этого сначала создаем контекст, потом создаю метаданные. В данном случае pairs просто принимает четное число строк: первая строка — это ключ, вторая — значение. Так же есть и способ получить данные из map. Потом создаем контекст уже с этими метаданными, то есть метаданные туда присваиваются через WithValue. И функции вызывать можно уже с этим контекстом.

Как получить метаданные, которые мне вернулись со стороны запроса?

```
var header, trailer metadata.MD

// создаем сессию
sessId, err := sessManager.Create(ctx,
        &session.Session{
                Login:     "rvasily",
                Useragent: "chrome",
        },
        grpc.Header(&header),
        grpc.Trailer(&trailer),
)
fmt.Println("sessId", sessId, err)
fmt.Println("header", header)
fmt.Println("trailer", trailer)

// проеряем сессию
sess, err := sessManager.Check(ctx,
        &session.SessionID{
                ID: sessId.ID,
        })
fmt.Println("sess", sess, err)
...
```

Создаем две переменные header, trailer: header — это то, что придет в начале запроса, trailer — то, что придет в конце запроса. Раньше в функцию передавали только аргумент функции, однако вызовы grpc принимают еще дополнительные опции — наши переменные метаданных. Не используйте header и trailer как параллели API для передачи параметров в удаленной функции. Туда должны передаваться только какие-то общесистемные вещи, типа requestID или токена авторизации. На стороне клиента вроде бы все несложно. Передали пару опций, указали функцию, указали, в какую переменную записать результат.

Теперь, каким образом это выглядит на сервере? Давайте смотреть.

```
func main() {
        lis, err := net.Listen("tcp", ":8081")
        if err != nil {
```

```
                log.Fatalln("cant listet port", err)
        }

        server := grpc.NewServer(
                grpc.UnaryInterceptor(authInterceptor),
                grpc.InTapHandle(rateLimiter),
        )

        session.RegisterAuthCheckerServer(server, NewSessionManager())

        fmt.Println("starting server at :8081")
        server.Serve(lis)
}
```

У сервера тоже есть UnaryInterceptor, то есть перехватчик одиночных запросов. Он тоже принимает функцию, что он внутри этой функции делает?

```
func authInterceptor(
        ctx context.Context,
        req interface{},
        info *grpc.UnaryServerInfo,
        handler grpc.UnaryHandler,
) (interface{}, error) {
        start := time.Now()

        md, _ := metadata.FromIncomingContext(ctx)

        reply, err := handler(ctx, req)

        fmt.Printf(`--
        after incoming call=%v
        req=%#v
        reply=%#v
        time=%v
        md=%v
        err=%v
`, info.FullMethod, req, reply, time.Since(start), md, err)
        return reply, err
}
```

Фактически то же самое, что перехватчик в клиенте. Получаем данные из запроса, вызываю handler, то есть ту функцию, которая проделает реальную работу, выводим всю нужную информацию. До вызова handler'а можно вкрутить авторизацию по токену, например. То есть проверять в метаданных, есть ли там access-token, правильный ли он, и разрешен ли для этого access-token вызов данного метода.

Теперь, каким образом добавить метаданные к ответам? Рассмотрим session. Тут все тоже просто.

```
func (sm *SessionManager) Create(ctx context.Context, in *session.Session)
     (*session.SessionID, error) {
        fmt.Println("call Create", in)

        header := metadata.Pairs("header-key", "42")
        grpc.SendHeader(ctx, header)

        trailer := metadata.Pairs("trailer-key", "3.14")
        grpc.SetTrailer(ctx, trailer)

        id := &session.SessionID{RandStringRunes(sessKeyLen)}
        sm.mu.Lock()
        sm.sessions[*id] = in
        sm.mu.Unlock()
        return id, nil
```

```
}
...
```

Создаем структуру с метаданными, и используя функцию SendHeader из grpc, и указывая сами метаданные. С trailer аналогично: просто указываем те значения, которые хотим отправить. Все это в таком виде никак не документировано, и все проверки будут только в runtime. Это параллельно API тому, что проверяется через compare time. Поэтому использовать это следует с осторожностью, ни в коем случае не как параллельно API для вашей бизнес-логики.

И есть еще одна функция, которую мы расмотрим. Это InTapHandle. InTapHandle — это функция, которая выполняется при приходе запроса, любого запроса на соединение. В случае, когда вызывается UnaryInterceptor то все уже распаковано, уже есть метаданные, параметры запроса, метод, куда их надо отправить. Вызов же функции, переданной в InTapHandle — в данном случае это функция rateLimiter, хотя реальной проверки для простоты примера там нет — вызывается, как только пришло само соединение, когда еще ничего не распаршено. Кроме того она не выполняется в отдельной горутине, в отличие от того же UnaryInterceptor. Она нужна, для того чтобы вы могли проверить rateLimiter или посчитать какую-то жесткую статистику до того, как все начнет парситься. Потому что если у нас сервер перезагружается, нам не нужно парсить дополнительные данные, чтобы потом выяснить, что мы не справляемся в работой. Мы должны в самом-самом начале соединения, на самом-самом приходе запроса это определить. Определить это мы можем по методу, ну и если что, мы можем докопаться до метаданных и прочего. Однако тут не стоит сажать какие-то блокирующие операции, потому что этот перехватчик TapHandle выполняется в одном потоке на соединении.

Мы рассмотрели средства, которые предоставляет grpc для того, чтобы вставить какие-то перехватчики и передать какие-то значения. Они используются в реальной работе и удобны. Однако, это, конечно же, не все, что предоставляет grpc, но одно из самых основных.

Стриминг сообщений

В этом разделе рассмотрим еще одну особенность grpc, которая называется streaming. Допустим, у нас есть какой-то сервис с транслитерацией, причем мы заранее не знаем, сколько туда текста придет. Например, кто-то говорит в реальном времени, это уходит на удаленный сервис по grpc, там это транслитерируется и возвращается обратно. Нет возможности отправить это одним большим куском, потому что тогда ответ придет тоже одним куском в самом конце, а это может получиться не очень быстро, поэтому не устраивает. Поэтому нужно разбивать большой запрос на маленькие части и организовать постоянный поток сообщений, и соответственно постоянный поток сообщений в ответ. Такой подход называется стримингом.

Рассмотрим следующий protobuf пакет и объявление сервиса.

```
syntax = "proto3";

// protoc --go_out=plugins=grpc:. *.proto

package translit;

message Word {
  string Word = 1;
}

// grpc-сервис транслитерации
service Transliteration {
    rpc EnRu (stream Word) returns (stream Word) {}
}
```

Раньше для одиночных вызовов он выглядел так: я отправляю слово, мне присылается слово, но стриминга там нет.

```
service Transliteration {
            rpc EnRu (Word) returns (Word) {}
        }
```

Если дописать всего лишь одно слово stream либо во входящем аргументе, либо в исходящем, то сразу же появляется стриминг. В данном случае и входящий аргумент является потоком, и исходящий аргумент

является потоком. Это значит, что у меня и в ту и в другую сторону идут непрерывные потоки данных, пока их не завершим. Если stream оставить только в исходящем аргументе, то это будет значить, что мы отправляем одиночный запрос, а в ответ идет поток постоянно. Если же сделать наоборот, то это будет значитт, что мы отправляем поток постоянно, а ответ вернется всего один раз. Но в данном примере для сервиса транслитерации требуется двусторонний стриминг.

Рассмотрим, как это выглядит в коде. Нужно сгенерировать по protobuf код, написать реализацию. Рассмотри функцию EnRu (используем внешний пакет для транслитерации).

```
func (srv *TrServer) EnRu(inStream translit.Transliteration_EnRuServer) error {
        for {
                inWord, err := inStream.Recv()
                if err == io.EOF {
                        return nil
                }
                if err != nil {
                        return err
                }
                out := &translit.Word{
                        Word: tr.Translit(inWord.Word),
                }
                fmt.Println(inWord.Word, "->", out.Word)
                inStream.Send(out)
        }
        return nil
}
```

У нас двусторонний стриминг, поэтому на вход приходит только один аргумент как раз этот стрим. Из него будем и читать, и писать в него же. В самой функции бесконечный цикл. Читаем, приходящий в protopath пакет. Если пришел EOF, это значит, что с той стороны стрим закрылся. Если пришел не EOF и не было никаких ошибок, значит успешно прочитали пакет, там есть word. Теперь конструируем обратный пакет, создаем translit.Word и, применив функцию транслитерации, и делаем send. То есть Recv читает, Send отправляет. Из цикла выйдем, если придет EOF, это будет значить, что с той стороны пакет закрылся, либо вернется ошибка. Также можно написать в любом месте return для завершения. Обратите внимание, обработка происходит синхронно: прочитали одно слово, записали одно. Потенциально это можно разделить по разным горутинам и делать операции абсолютно асинхронно.

Рассмотрим теперь клиента.

```
func main() {

        grcpConn, err := grpc.Dial(
                "127.0.0.1:8081",
                grpc.WithInsecure(),
        )
        if err != nil {
                log.Fatalf("cant connect to grpc")
        }
        defer grcpConn.Close()

        tr := translit.NewTransliterationClient(grcpConn)

        ctx := context.Background()
        client, err := tr.EnRu(ctx)

        ...
```

Здесь обычное подключение к grpc, создание клиента, контекст. Затем создаем поток client. В клиенте как раз уже и отправлять и читать будем асинхронно. Создадим WaitGroup и начинаем туда писать.

```
        ...

        wg := &sync.WaitGroup{}
```

```
wg.Add(2)

go func(wg *sync.WaitGroup) {
        defer wg.Done()
        words := []string{"privet", "kak", "dela"}
        for _, w := range words {
                fmt.Println("-> ", w)
                client.Send(&translit.Word{
                        Word: w,
                })
                time.Sleep(time.Millisecond)
        }
        client.CloseSend()
        fmt.Println("\tsend done")
}(wg)

...
```

Будем отправлять три слова: privet, kak, dela. Будем идти по ним в цикле, делать Send и немножко спать, чтобы хотя бы как-то была видна асинхронность. Когда все слова закончились, выходим из цикла и отправляем на ту сторону Send, то есть я говорим клиенту: закройся. Нельзя его закрыть сразу, потому что оттуда еще могут что-то читать. Поэтому отправляем признак закрытия. Когда этот признак закрытия сработает, на сервер придет EOF. КОгда отправка закончилась, горутина вышла. Теперь посмотрим чтение.

```
...
go func(wg *sync.WaitGroup) {
        defer wg.Done()
        for {
                outWord, err := client.Recv()
                if err == io.EOF {
                        fmt.Println("\tstream closed")
                        return
                } else if err != nil {
                        fmt.Println("\terror happed", err)
                        return
                }
                fmt.Println(" <-", outWord.Word)
        }
}(wg)
...
```

По-прежнему в цикле читаю Recv так же, как и на стороне сервера. Получаем очередное слово и выводим его на экран. Общение с сервером происходит в двух отдельных горутинах: почти параллельно одна горутина шлет, вторая читает. Что-то типа такого канала между серверами, либо что-то похожее на pushsub.

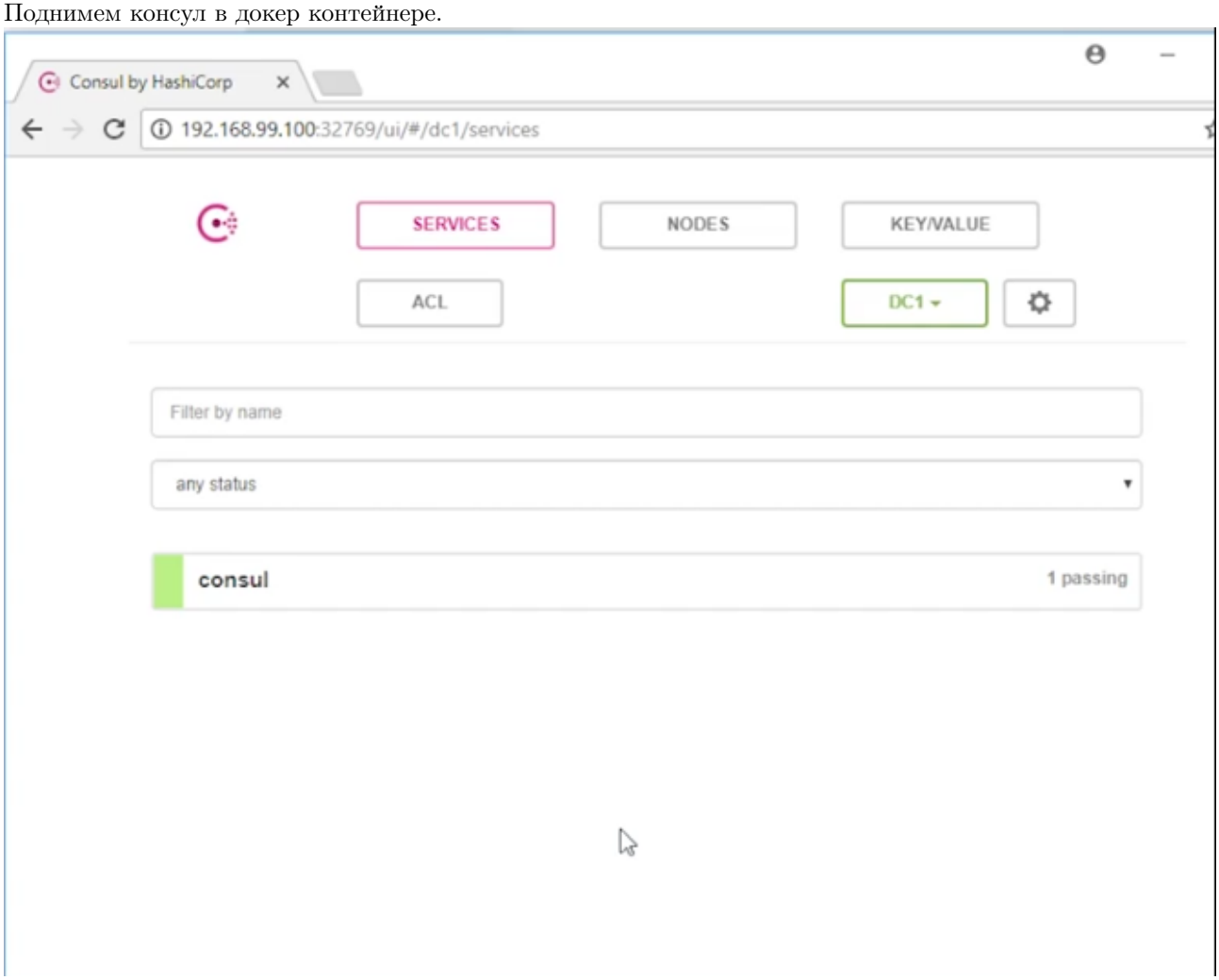
Стриминг — очень мощный инструмент. Например, в Google сервис распознавания речи работает как раз через grpc streaming. То есть вы отправляете аудиопоток, как в примере мы отправляли одиночные слова, вы отправляете в аудиопоток просто бинарные данные на ту сторону, а вам возвращается распознанный текст. Так же, например, телеком и CISC, Jupiter или Alcatel-Lucent используют в своих железках grpc стриминг для отправки телеметрии. Поэтому, если вам нужно отправить большой набор данных либо принять большой набор данных, причем вы это хотите делать не одним большим куском, а по мере того, как они поступают, вам поможет стриминг.

Балансировка нагрузки и использование Consul

В этом разделе мы поговорим про балансировку нагрузки и сервис Discovery. Это два очень актуальных вопроса в мире микросервисов. Дело в том, что микросервисы часто запускаются на большом количестве нод для обеспечения высокой доступности, соответственно эти ноды могут удаляться, добавляться и вашему сервису нужно узнавать, что какой-то сервис ушел, либо какая-то нода появилась, чтобы определять, куда пускать нагрузку. Мы будем рассматривать оба этих вопроса на примере консула. Консул — это довольно мощный инструмент для организации сервис Discovery, health-чеков, балансировки нагрузки, key-value хранилища, распределенных логов. Мы не будем очень углубляться в его работу, потому

что он достоин отдельного курса. Однако мы рассмотри с практической точки зрения, как можно его использовать. Кстати, консул тоже написан на Go.

Поднимем консул в докер контейнере.



У него есть небольшая админка, где можно посмотреть какой-то статус нод, какие они вообще есть, какие сервисы там зарегистрированы. Сейчас, как видно, в нем зарегистрирован только сам консул, больше ничего нет. И поэтому начнем с того, что зарегистрируем в нем наш микросервис авторизации, который мы рассматривали в предыдущих примерах. Теперь рассмотрим код.

```
package main

import (
        ...
        consulapi "github.com/hashicorp/consul/api"
)

var (
        grpcPort   = flag.Int("grpc", 8081, "listen addr")
        consulAddr = flag.String("consul", "192.168.99.100:32769",
                        "consul addr (8500 in original consul)")
)

/*
        go run *.go --grpc="8081" --consul="192.168.99.100:32769"
        go run *.go --grpc="8082" --consul="192.168.99.100:32769"
*/
```

Консул — это внешний пакет, его нужно поставить через Go get. Поскольку нужно будет запустить сразу несколько нод, чтобы они уходили, приходили, все-таки это сервис Discovery, сразу сделаем сервер конфигурируемым, используя пакет flag. У нас будет порт и адрес консула конфигурируемыми.

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

        port := strconv.Itoa(*grpcPort)

        lis, err := net.Listen("tcp", ":"+port)
        if err != nil {
                log.Fatalln("cant listen port", err)
        }

        server := grpc.NewServer()

        session.RegisterAuthCheckerServer(server,
                NewSessionManager(port))

        config := consulapi.DefaultConfig()
        config.Address = *consulAddr
        consul, err := consulapi.NewClient(config)

        serviceID := "SAPI_127.0.0.1:" + port

        err = consul.Agent().ServiceRegister(&consulapi.AgentServiceRegistration{
                ID:      serviceID,
                Name:    "session-api",
                Port:    *grpcPort,
                Address: "127.0.0.1",
        })
        if err != nil {
                fmt.Println("cant add service to consul", err)
                return
        }
        fmt.Println("registered in consul", serviceID)

        defer func() {
                err := consul.Agent().ServiceDeregister(serviceID)
                if err != nil {
                        fmt.Println("cant add service to consul", err)
                        return
                }
                fmt.Println("deregistered in consul", serviceID)
        }()

        fmt.Println("starting server at " + port)
        go server.Serve(lis)

        fmt.Println("Press any key to exit")
        fmt.Scanln()
}
```

Начинаем мы нашу программу с того, что распарсим флаги, с которыми будем запускать программу, потому что автоматически они никак не распарсятся. После этого приводим их к строке, просто для удобства. Затем создаем grpc-сервис и регистрируем в нем наш сервис. Пока практически нет отличий от предыдущих, за исключением двух. Первое отличие: добавили в менеджер сессий порт, чтобы в логе можно было видеть, на какой сервис мы пришли, потому что grpc в штатном режиме не позволяет это делать. И также добавили заглушку, потому что у нас не настоящий сервис авторизации, и все просто хранится все в мапке. Соответственно, между собой два сервиса никак не контактируют, поэтому нужна заглушка. Далее создаем config для консула, пытаемся к нему подключиться, после этого задаем сервис ID, используя какой-то префикс, IP-адрес и порт, чтобы идентифицировать. На самом деле ID может быть любой: MD5, UUID, что угодно. Далее обращаемся к консулу и пытаемся зарегистрировать там наш сервис: обращаемся к агенту, вызываем функцию "зарегистрировать сервис"и передаем туда структуру, в которой все будет зарегистрировано. ID — это как раз идентификатор конкретного сервиса. Name — это

имя предоставляемого сервиса. ID должен быть уникальным в рамках разных нод, а Name в рамках ноды одного кластера одних и тех же микросервисов должен быть одинаковым. Кроме этого указываем порт и адрес, на котором он доступен. Поскольку это учебный, а не продакшн пример, не указываем никаких health-чеков, сервис сразу же становится доступен. После того, как зарегистрировали сервер сразу же через отложенное выполнение defer регистрируем функцию, которая этот сервис выведет из нагрузки, точнее сообщит консулу о том, что сервис ушел. Таким образом, когда программа закончится более-менее штатно, выполнится defer и скажет консулу, что на этот сервер слать нагрузки больше не надо. Затем стартуем непосредственно grpc-сервис и просто ждем, пока его не остановят нажатием клавиши. Попробуйте поднять два сервера на разных портах и посмотрите, как это отобразится в консуле.

Далее мы рассмотрим, как написать клиента, который будет обращаться к этим серверам.

```
import (
        ...
        consulapi "github.com/hashicorp/consul/api"
)

var (
        consulAddr = flag.String("addr", "192.168.99.100:32769",
                        "consul addr (8500 in original consul)")
)

var (
        consul       *consulapi.Client
        nameResolver *testNameResolver
)
```

По-прежнему подключаем консул api. В переменных адрес консула конфигурируемый. Клиент консулу и Resolver сделаны глобальными, чтобы можно было обращаться к ним из внешних функций.

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

        var err error
        config := consulapi.DefaultConfig()
        config.Address = *consulAddr
        consul, err = consulapi.NewClient(config)

        health, _, err := consul.Health().Service("session-api", "", false, nil)
        if err != nil {
                log.Fatalf("cant get alive services")
        }

        servers := []string{}
        for _, item := range health {
                addr := item.Service.Address +
                        ":" + strconv.Itoa(item.Service.Port)
                servers = append(servers, addr)
        }

        nameResolver = &testNameResolver{
                addr: servers[0],
        }

        grcpConn, err := grpc.Dial(
                servers[0],
                grpc.WithInsecure(),
                grpc.WithBlock(),
                grpc.WithBalancer(grpc.RoundRobin(nameResolver)),
        )
        if err != nil {
                log.Fatalf("cant connect to grpc")
```

```
        }
        defer grcpConn.Close()

        if len(servers) > 1 {
                var updates []*naming.Update
                for i := 1; i < len(servers); i++ {
                        updates = append(updates, &naming.Update{
                                Op:   naming.Add,
                                Addr: servers[i],
                        })
                }
                nameResolver.w.inject(updates)
        }

        sessManager := session.NewAuthCheckerClient(grcpConn)

        // тут мы будем периодически опрашивать консул на предмет изменений
        go runOnlineServiceDiscovery(servers)

        ctx := context.Background()
        step := 1
        for {
                // проверяем несуществующую сессию
                // потому что сейчас между сервисами нет общения
                // получаем загшулку
                sess, err := sessManager.Check(ctx,
                        &session.SessionID{
                                ID: "not_exist_" + strconv.Itoa(step),
                        })
                fmt.Println("get sess", step, sess, err)

                time.Sleep(1500 * time.Millisecond)
                step++
        }
}
```

В main'е мы по-прежнему мы подключаемся к консулу, дальше доступ к сервисам получаем уже, обращаясь к сервису health-чеков — говорим ему, дай мне, пожалуйста сервисы session-api и указываем дальнейшие параметры. В примере не передаем никаких, однако можно указать теги, можно указать сервисы, которые только прошли проверку, и еще некоторые вещи. У нас в примере нет health-чеков, и сервис стартует живым, доступным для нагрузки. В продакшне так делать нельзя, лучше, чтобы сначала все health-чеки прошли, чтобы убедиться, что все действительно работает и мы нигде не ошиблись в ноде и в ее конфигурации. Дальше по всем этим сервисам итерируюемся и создаем из них слайс адресов (из консула возвращается структура, где есть отдельно IP, отдельно адрес). Далее создаем структуру NameResolver и указываем там только нулевой адрес, чтобы что-то было в самом начале. После этого создаем подключение к grpc, указывая опцию WithBlock, это значит, что соединение не вернет результат, не вернет connect grpc, пока не будет установлено хотя бы одно соединение с сервисом. Кроме этого передаем еще одну опцию, которая говорит, что нужно использовать балансер между разными нодами, то есть нод будет много. В качестве балансера я используем RoundRobin. В grpc политика такая, что у вас есть простой балансер, а если вы хотите какой-то более сложный балансер, например, со взвешенным весами или что-то еще, то будьте добры реализуйте его в виде отдельного внешнего сервиса. В данном примере мы будем использовать RoundRobin. В него надо передать нашу структуру NameResolver, мы ее взяли из тестового пакета самого grpc. Это структура будет сообщать нашему балансеру о том, что нужно добавить ноду в список серверов, либо убрать какую-то ноду. Итак, создали соединение, нужно с помощью defer указать, что оно должно быть закрыто при выходе из программы.

Идем далее. Если вдруг больше одного сервера вернулось из консула, то будем как раз использовать NameResolver, и буду их туда добавлять. Далее все просто, создаем слайс и итерируемся по всем сервисам, добавляем туда определенного рода структуру — структуру update, которая говорит, что нужно что-то сделать в списке серверов уже у соединения grpc. В структуре есть операция Add, помимо Add, конечно, будем использовать Delete потом. И в структуре есть адрес, на котором нужно провести опера-

цию. Затем выполняем обновление. Итак, мы создали соединение grpc, добавили туда все адреса, теперь оно будет коннектиться к разным нодам. Дальше, я создаем sessManager (вы его видели уже), и запускаем в отдельной горутине обновление, то есть я будем опрашивать consul, на предмет того, были ли там уже какие-то обновления, пришли ли какие-то новые сервисы, нужно ли что-то удалить или добавить. К этой функции мы вернемся после. Далее в бесконечном цикле будем просто пытаться проверить какую-то сессию. Напомним, что у нас микросервисы друг с другом никак не связаны, они используют отдельные автономные map'ки внутри, поэтому будем имитировать деятельность, запрашивая несуществующий айдишник и засыпая, и на той стороне будет возвращаться заглушка.

Теперь давайте посмотрим, каким образом будет происходить удаление сервисов либо же добавление. За это отвечает как раз функция runOnlineServiceDiscovery, запущенная в отдельной горутине.

```
func runOnlineServiceDiscovery(servers []string) {
        currAddrs := make(map[string]struct{}, len(servers))
        for _, addr := range servers {
                currAddrs[addr] = struct{}{}
        }
        ticker := time.Tick(5 * time.Second)
        for _ = range ticker {
                health, _, err := consul.Health().Service("session-api", "", false, nil)
                if err != nil {
                        log.Fatalf("cant get alive services")
                }

                newAddrs := make(map[string]struct{}, len(health))
                for _, item := range health {
                        addr := item.Service.Address +
                                ":" + strconv.Itoa(item.Service.Port)
                        newAddrs[addr] = struct{}{}
                }

                var updates []*naming.Update
                // проверяем что удалилось
                for addr := range currAddrs {
                        if _, exist := newAddrs[addr]; !exist {
                                updates = append(updates, &naming.Update{
                                        Op:   naming.Delete,
                                        Addr: addr,
                                })
                                delete(currAddrs, addr)
                                fmt.Println("remove", addr)
                        }
                }
                // проверяем что добавилось
                for addr := range newAddrs {
                        if _, exist := currAddrs[addr]; !exist {
                                updates = append(updates, &naming.Update{
                                        Op:   naming.Add,
                                        Addr: addr,
                                })
                                currAddrs[addr] = struct{}{}
                                fmt.Println("add", addr)
                        }
                }
                if len(updates) > 0 {
                        nameResolver.w.inject(updates)
                }
        }
}
```

В начале строим map'ку из текущих адресов - с map'кой удобно работать. Далее создаем ticker. Напомню, что этот ticker не имеет функции stop, и если наша функция завершится, то будет утечка ресурсов.

Однако мы не планируем, что она будет завершаться, поэтому просто пускаем цикл по тикеру. То есть идет чтение из канала раз в пять секунд. Когда в очередной раз в канал что-то пришло, мы обращаемся к consul'у, использую уже знакомую функцию Service, которая получает service-api и текущие адреса. Из этих адресов также строим map'у. А дальше итерируемся по двум map'ам и смотрим, добавился ли какой-то сервис, либо же какой-то сервис удалился.

В данном случае тот способ, которым опрашиваем consul, называется polling. Периодически обращаемся к нему и спрашиваем: есть что-то? Возможны и другие варианты, их мы рассмотрим отдельно.

Поэкспериментируйте и посмотрите сами, как все это работает. Сервис discovery и балансировка нагрузки — это очень большая тема, но этот пример должен послужить хорошей отправной точкой того, каким образом это можно реализовать: либо со стороны grpc, либо со стороны самого consul'a.

Дополнительные темы

grpc-gateway - получаем доступ к grpc-сервисам через HTTP

В предыдущих разделах мы рассматривали примеры, в которых успешно связывали между собой микросервисы по протоколы grpc. Однако иногда может возникнуть необходимость сделать http-интерфейс к grpc-сервису. Например, когда для вашего языка нет генератора protobuf-файлов или вы не хотите усложнять и идти дальше обычного http. На этот случай есть утилита grpc-gateway, которая позволяет сгенерировать reverse proxy до вашего сервиса, который будет принимать в себя json по http, а в ваш сервис ходить уже по grpc.

Давайте посмотрим, каким образом с этим работать не практике. У нас есть описание сервиса AuthChecker, уже знакомого вам, и методы Create, Check, Delete.

```
// grpc-сервис проверки авторизации
service AuthChecker {
    rpc Create (Session) returns (SessionID) {
      option (google.api.http) = {
        post: "/v1/session/create"
        body: "*"
      };
    }
    rpc Check (SessionID) returns (Session) {
      option (google.api.http) = {
        get: "/v1/session/check/{ID}"
      };
    }
    rpc Delete (SessionID) returns (Nothing) {
      option (google.api.http) = {
        post: "/v1/session/delete"
        body: "*"
      };
    }
}
```

Мы расширяем эти методы при помощи некоторых опций. В данном случае это опция google.api.http. И в этой опции указываем метод post с урлом и то, что body принимаем прямо целиком, или в функции Check указывает, что проверку осуществляем через метод get и арамент указываем прямо в самом урле. Сам плагин для grpc-gateway умеет обрабатывать эти опции и сгенирирует файла, где будет обвязка, которая будет делать роутинг и ходить в сервис по grpc. Генерация самого grpc-сервиса осуществляется командой

```
protoc -I/usr/local/include -I. \
  -I$GOPATH/src \
  -I$GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/third_party/googleapis \
  --go_out=plugins=grpc:. \
  session.proto
```

Следующая команда генирирует http-обвязку для вашего сервиса

```
protoc -I/usr/local/include -I. \
  -I$GOPATH/src \
  -I$GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/third_party/googleapis \
  --grpc-gateway_out=logtostderr=true:. \
  session.proto
```

И, наконец, последняя команда генирирует описание для swagger'а

```
protoc -I/usr/local/include -I. \
  -I$GOPATH/src \
  -I$GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/third_party/googleapis \
  --swagger_out=logtostderr=true:. \
  session.proto
```

В обвязке сгенинрируется довольно большой файл. В нем будет собственный мультиплексор, своя валидация и много других своих небольших утилит. Не будем вдаваться в автосгенеренный код, посмотрим лучше, как это исполоьзовать на практике.

```
package main

import (
        ...
        // "../session"
        "coursera/microservices/gateway/session"
)

func main() {
        proxyAddr := ":8080"
        serviceAddr := "127.0.0.1:8081"

        go gRPCService(serviceAddr)
        HTTPProxy(proxyAddr, serviceAddr)
}
```

В mail'е подключаем сессию, где у нас уже все сгенирировано. В одном сервисе стартуем и grpc-сервис, куда будет ходить пользователь, и саму проксю. В функции gRPCService нет ничего интересного, мы её уже видели в предыдущих примерах. В HTTPProxy

```
func HTTPProxy(proxyAddr, serviceAddr string) {
        grcpConn, err := grpc.Dial(
                serviceAddr,
                grpc.WithInsecure(),
        )
        if err != nil {
                log.Fatalln("failed to connect to grpc", err)
        }
        defer grcpConn.Close()

        grpcGWMux := runtime.NewServeMux()

        err = session.RegisterAuthCheckerHandler(
                context.Background(),
                grpcGWMux,
                grcpConn,
        )
        if err != nil {
                log.Fatalln("failed to start HTTP server", err)
        }

        mux := http.NewServeMux()
        // отправляем в прокси только то что нужно
        mux.Handle("/v1/session/", grpcGWMux)
```

```
        mux.HandleFunc("/", helloWorld)

        fmt.Println("starting HTTP server at " + proxyAddr)
        log.Fatal(http.ListenAndServe(proxyAddr, mux))

}
```

мы подключаемся к grpc, создаем серверный мультиплексор для обработки запросов от grpc-gateway, В нем регистрируем клиент к микросервису, передаем туда контекст, указываем мультиплексор и коннект. Далее объявляем свой собственный стандартный мультиплексор. И все запросы на v1/session отправляем в grpc. Здесь стоит обратить внимание на то, что в одной проксе может быть доступ не к одному, а нескольким сервисам. Далее начинаем слушать сокет по http. собственно, это все, что нужно. Попробуйте собрать код и обратится к сервису следующими командами через curl.

```
curl -X POST -k http://localhost:8080/v1/session/create -H "Content-Type: text/plain"
-d '{"login":"rvasily", "useragent": "chrome"}'
curl http://localhost:8080/v1/session/check/XVlBzgbaiC
curl -X POST -k http://localhost:8080/v1/session/delete -H "Content-Type: text/plain"
-d '{"ID":"XVlBzgbaiC"}'
```

Теперь все клиенты, которые не хотят тащить за собой какие-то дополнительные библиотеки, могут ходить в ваш сервис п о http. Еще у нас остался неразобранным файл для swagger'a. Ему мы посвятим следующий раздел.

Swagger - генерируем клиент и сервер из документации к api

В этом разделе мы поговорим про Swagger. Swagger — это набор утилит для OpenAPI Specification. Это спецификация для описания API, на основе которого генерируется документация, клиентский код, серверный код, тесты и предоставляет удобный инструмент для проектирования API. Составить представления, что это за формат вы можете, посмотрев json с этим форматом в папке с кодом с лекции. Однако, воспринимать информацию в json не так уж просто, поэтому лучше смотреть на документацию. У swagger'a есть команда serve, в которую нужно указать файл с кодом и порт после ключа -p. С её помощью генерируется документация, в которой красиво видны все методы, информация, какие значения они принимают и возвращают. Если у вас много микросервисов, большой набор методов — полезно иметь хорошую документацию.

Пойдем дальше и посмотрим, каким образом можно сгенирировать клиента для вашего api, используя этот формат. Есть следующая команда, которая все сделает.

```
rm -rf sess-client && \
    mkdir sess-client && \
    swagger generate client -f ../gateway/session/session.swagger.json \
    -A sess-client/ -t ./sess-client/
```

Указываем, из какого файла следует генирировать, называем это sess-client и указываем, в какую папку положить. Сгенирируется огромное количество файлов — самоого клиента, сервис и модели.

Каким образом, используя полученного клиента, обратиться к сервису? У нас есть код consumer.

```
package main

import (
        "fmt"

        httptransport "github.com/go-openapi/runtime/client"
        "github.com/go-openapi/strfmt"

        // "../sess-client/client"
        apiClient "coursera/microservices/swagger/sess-client/client"
        auth "coursera/microservices/swagger/sess-client/client/auth_checker"
        models "coursera/microservices/swagger/sess-client/models"
)
```

```
func main() {

        transport := httptransport.New("127.0.0.1:8080", "", []string{"http"})

        client := apiClient.New(transport, strfmt.Default)
        sessManager := client.AuthChecker

        // создаем сессию
        sessId, err := sessManager.Create(auth.NewCreateParams().WithBody(
                &models.SessionSession{
                        Login:     "rvasily",
                        Useragent: "chrome",
                },
        ))
        fmt.Println("sessId", sessId, err)

        // проверяем сессию
        sess, err := sessManager.Check(auth.
                NewCheckParams().
                WithID(sessId.Payload.ID))
        fmt.Println("after create", sess, err)

        // удаляем сессию
        _, err = sessManager.Delete(auth.NewDeleteParams().WithBody(
                &models.SessionSessionID{
                        ID: sessId.Payload.ID,
                },
        ))

        // проверяем еще раз
        sess, err = sessManager.Check(auth.
                NewCheckParams().
                WithID(sessId.Payload.ID))
        fmt.Println("after delete", sess, err)
}
```

Подключаем довольно много разных includ'ов. Создаем транспорт, указывая. на какой адрес идти. Создаем клиента, создаем непосредственно менеджер сессий, потому что внутри одного api может быть несколько подсервисов. И собственно вызываем создание сессии. Получаем sessionID, теперь можно проверить эту сессию, удалить и проверить еще раз.

Стоит отметить, что, поскольку код сгенирирован на основе swagger-описания, то при запуске он идет в ту revers-proxy grps-gateway, который потом ходит в grpc-сервис. То есть у нас получается http-клиент к http-серверу, который явряется grpc-клиентом к grpc-сервису. Конечно, при помощь swagger'а можно сгенирировать и серверное описание, это будет очень забавно... Если у вас в работе используется очень много микросеривисов, swagger позволяет как-то их самортизировать и добиться контроля над их ответами. Swagger очень хороший инструмент, очень мощный, с огромным количеством опций и параметров и достоин более детального изучения.