Если вы видите что-то необычное, просто сообщите мне.

Строим свой первый Rest API с помощью GO

У нас есть три необходимые части для построения.

- API
- Rest API
- Rest API на GO

API

Если вы близки к компьютерам достаточно давно, вы возможно слышали об этих вещах. Что такое API?

API значит программный интерфейс приложения. Как и большинство вещей в информатике абревиатура не сильно помогает.

На самом деле это значит, что он выставялет функциональность без выставления внутренностй. Если ваша программа написана на языке который поддерживает функции или методы(большинство языков) вы должны понимать, о чём идет речь.

```
func addNumber(a, b int) int {
    // DO AMAZING MATH HERE
    // and return the result
}
```

Даже если вы совсем новичок в go, вы можете сказать, что функция возвращает результат сложения двху чисел.

Как пользователь функции вы просто вызываете функцию и не беспокоитесь о том, функция выполняет свою работу(не доверяйтся всем функциям подряд)

Это всё что такое API. API может быть функция которую вы написали, или функция из библиотеки или метод фреймворка, или http точка доступа.

Rest API

Большинство API написаные в наши дни это web api. Только не цитируйте меня, так как я не делал никаких исследований чтобы получить настоящее число. Но учитывая количество web сервисов и прилжений я не думаю, что я далеко от правды.

What is REST

REST - это акроним для REpresentational State Transfer(передача репрезентативного состояния). Это архитекторный стиль распределнной гипермедия системы и была впервые представлена Роем Филдингом в 2000 году в его известной диссертации.

Как любые другие архитектурные стили, REST так же имеет свои 6 формирующих его ограничений, которые должны быть удовлетворены и если интерфейс хочет называться RESTful. Эти принципы приведены ниже.

Ведущие принципы REST

1. Клиент-сервер - Разделяя проблемы пользовательских интерфейсов от проблем хранения данных, вы улучшаем переносимость пользовательского интерфейса на многие платформы и улучшаем масштабируемость упрощением сервреных компонентов.
2. Не хранит состояние - каждый запрос от клиента к серверу должен содержать всю необходимую информацию для понимания запросов, и не может воспользоваться любых сохраненным содераженим на сервере. Состояние сессии отсюда хранится полностью на клиенте.

3. Кешируемость - Ограничения кэша требуют. чтобы данные в ответе на запрос были явно или не явно обозначены как кэшируемые или не кешуруемые. Если ответ кэшируемый, тогда право кэша клиента для переиспользования данные ответа в дальнейшем, похожем запросе.
4. Универсальный интерфейс - Применя принцыпи програмной инженерии в целом к компонентам интерфейса, общая архитектура системы упрощена и видимость взаимодействий улучшена. Для того, чтобы получить универсальный интерфейс, множество универсальных ограничений нужны для указания поведения компонентов. REST определяется четырьмя ограничивающими интерфейсами:
   - Определение ресурса
   - Управление ресурсами через репрезентацию
   - Самоописываемое сообщение
   - Гипермедия - как движок состояния приложения.
5. Слоёная система - Стиль слоёной системы позволяет архитектуре быть составленной из иерархических слоёв ограниченными поведением компонентов, так, что каждый компоненто не может "смотреть сковозь" слой с которым происходит взаимодействие.
6. Код по запросу(по желанию) - REST позволяет расширить функциональность клиенту скачать и выполнив код в форме аплетов или скриптов. Это урощает клиентов с помощью сокращения количества особенностей требуемых для погдготовки реализации.

HTTP глаголы

Вот несколько условностей соблюдаемы HTTP API. Это не часть спецификации REST. Но нам нужно это понять, что бы использовать REST API по-понной.

HTTP определяет набор методов-запросов чтобы указать желаемое действие для данного ресурса. Так же они могут быть существительными, эти методы-запросы иногда называются HTTP глаголы. Каждый из них реализует различную семантику, но которые общие особенности поделены на группы: например запрос может быть безопасный, неизменяемый или кэшируемый.

`GET` - метод запрашивает представление указанного ресурка. Запросы использующие `GET` должны только получать данные.

`HEAD` - метод просит ответ идентичный `GET` запросу, но опускает `body`.

`POST` - методв используется для отправки сущности указанному ресурсу, часто является причиной изменения изменения состояния или имеет побочный эффект на сервер.

`PUT` - метод заменяет все текущее представление целевого ресурса загруженным в запросе.

`DELETE` - метод удаляет указанный ресурс.

`CONNECT` - метод устанавливает тоннель к серверу указанному как целевой.

`OPTIONS` - метод используется для описания возможностей подключения к удаленному ресурсу.

`TRACE` - метод производит сообщение петлевого контроля на пути к ресурсу цели.

`PATCH` - метод используется для применения частичных изменений ресурса.

ЭТО ВСЁ ВРАНЬЕ.

Статус коды

1xx Информация

100 Continue 101 Switching Protocols 102 Processing 2xx Success

200 OK 201 Created 202 Accepted 203 Non-authoritative Information 204 No Content 205 Reset Content 206 Partial Content 207 Multi-Status 208 Already Reported 226 IM Used 3xx Redirects

300 Multiple Choices 301 Moved Permanently 302 Found 303 See Other 304 Not Modified 305 Use Proxy 307 Temporary Redirect 308 Permanent Redirect 4xx Client Error

400 Bad Request 401 Unauthorized 402 Payment Required 403 Forbidden 404 Not Found 405 Method Not Allowed 406 Not Acceptable 407 Proxy Authentication Required 408 Request Timeout 409 Conflict 410 Gone 411 Length Required 412 Precondition Failed 413 Payload Too Large 414 Request-URI Too Long 415 Unsupported Media Type 416 Requested Range Not Satisfiable 417 Expectation Failed 418 I'm a teapot 421 Misdirected Request 422 Unprocessable Entity 423 Locked 424 Failed Dependency 426 Upgrade Required 428 Precondition Required 429 Too Many Requests

431 Request Header Fields Too Large 444 Connection Closed Without Response 451 Unavailable For Legal Reasons 499 Client Closed Request 5xx Server Error

500 Internal Server Error 501 Not Implemented 502 Bad Gateway 503 Service Unavailable 504 Gateway Timeout 505 HTTP Version Not Supported 506 Variant Also Negotiates 507 Insufficient Storage 508 Loop Detected 510 Not Extended 511 Network Authentication Required 599 Network Connect Timeout Error This also has no actual meaning.

Терминология

Ниже - самые важные понятия связанные с REST API.

- Ресурс - объект или представление чего-то, что имеет некоторую ассоциацию данных с ним и может иметь набор методов для обработки оных. К примеру: Живтоные, школы или работники ресурсы, а `delete`, `add`, `update` - операции для обработки этих данных.
- Коллекции - наборы ресурсов: Компании это наборы ресурсов компаний.
- URL - это путь по которому ресурс может быть определен и действия которые необходимо с ним произвести.

API Endpoint

Вот как выглдяит пример такой точки:

```
https://www.github.com/golang/go/search?q=http&type=Commits
```

Разделим URL на части:

```
protocol□subdomain□domain□path□Port□query
http/https□subdomain□base-url□resource/some-other-resource□some-port□key value pair
https□www□github.com□golang/go/search□80□?q=http&type=Commits
```

Протоколы

Как браузер или клиент должен взаимодействовать с сервером.

Поддомен

Подраздел главного домена.

Порт

Порт сервера на котором запущено приложения. По умолчанию это 80. Но в большинстве случаем вы его не видим.

Петь

Пусть - параметры REST API отражающие ресурсы.

```
https://jsonplaceholder.typicode.com/posts/1/comments
posts/1/comments
```

Этот пусть отражает коментарии 1го поста ресурса.

Базовой структурой является

```
top-level-resource/<some-identifier>/secondary-resource/<some-identifier>/...
```

Запрос

Запросы - пары ключ-значение информации, используемый в основном для целей фильтрования.

```
https://jsonplaceholder.typicode.com/posts?userId=1
```

Часть после `?` это параметры запроса. У нас есть только один запрос тут: `userID=1`.

Заголовки

Это не часть самого URL, но заголовки это часть сетевого компонента посылаемые клиентом или сервером. В зависимости от того, кто послал его. Есть 1 типа заголовков.

1. Заголовок запроса (client -> server)
2. Заголовок ответа (server -> client)

Тело

Вы можете добавить дополнительную инфомацию в обза запроса к серверу и к ответу от сервера.

Тип ответа

Обычно JSON или XML.

В наши дни это обычно JSON.

Rest API на GO

Это то почему вы тут. Ну или я, по крайней, мере я надеюсь на это.

Если вы пишите REST API, почему вы должны выбрать Go?

- Он компилируем. Вы получаете маленькие бинарники.
- Он быстр. Медленнее чем c/c++ или rust, но быстрее чем большинство других языков web программирования.
- Он легок в изучении.
- Он работает отлично в мире микросервисов - это причини номер 1.

net/http

Стандартная библиотека в go идущая с `net/http` пакетом, что является отличной точкой отсчета для построения REST API. И большинство других библиотек добавлют особенности тоже взаимодействуют с `net/http` пакетом, поэтому понимание пакется является критичным для использования golang в качестве REST API.

```
net/http
```

Нам, возможно, не нужно знать всё в пакете `net/http`. но есть несколько вещей, который мы должны знать для начала.

Интерфейс обработчика

нам нужно помнить интерфейс обработчика:

```
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}
```

Ну вот и он.

У него есть тольк один метод и только.

Структура или объект будет обработан если он имеет один метод `ServeHTTP` который принимает `ResponseWriter` и указатель на `Request`.

Теперь, с этим знанием, мы готовы к некоторым увечьям.

Начнем

Я думаю мы готовы начать.

Мы узнали много теории. Я обещал что вы создадите свой первый RestAPI.

Простой Rest API

В папке где вы хотите писать ваш код выполните команду в терминале:

```
go mod init api-test
```

Созайте новый файл, можно назвать его как угодно.

Я назвал свой main.go

```
package main
```

```
import (
    "log"
    "net/http"
)

type server struct{}

func (s *server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "hello world"}`))
}

func main() {
    s := &server{}
    http.Handle("/", s)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Пройдемся по коду.

В самом начала, у нас стоит строка `package main`, все go исполняемые файлы должны иметь `main` пакет. Дальше идут иморты:

- `log` для логирования ошибок, если случаются.
- `net/http` потому что мы пишем rest api.

Далее идет структура под названием `server`. Она не имеет полей. Добавим метод `ServerHTTP` этому `server`, чтобы удовлетворить обработчик интерфейс. Одна вещь, которую вы можете заметить в go, нам не нужно явно говорить какой из интерфейсов мы реализуем. Компилятор достаточно умен, чтобы выяснить это самостоятельно. В `ServerHTTP` методе мы настраиваем `httpStatus` 200, чтобы обозначить, что запрост прошел успешно. Мы видим тип содержания `application/json` так, что клиент понимает когда мы отправляет ему что-то полезное. В мы пишем `{"message": "hello world"}` в ответ.

Запустим наш сервер.

```
go run main.go
```

Для проверки выполняем команду из соседнего терминала:

```
curl localhost:8000
```

В ответ получае json описанный выше. Отличная работа!

Но подождите.

Давайте посмотрим какие другие HTTP глаголы обрабатывает наше приложение.

Теперь попробуем выполлнить POST запрос:

```
curl localhost:8000 -X POST
```

Когда мы выполняем запрос, то получаем тот же самый результат.

Это вовсе не баг. Но часто мы будем хотеть чтобы выполнялись разные задачи в зависимости от типа запроса.

Посмотрим, что мы тут можем сделать.

Изменим наш `ServerHTTP` метод следующим образом:

```
func (s *server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    switch r.Method {
    case "GET":
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(`{"message": "get called"}`))
    case "POST":
        w.WriteHeader(http.StatusCreated)
        w.Write([]byte(`{"message": "post called"}`))
    case "PUT":
        w.WriteHeader(http.StatusAccepted)
        w.Write([]byte(`{"message": "put called"}`))
    case "DELETE":
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(`{"message": "delete called"}`))
    default:
        w.WriteHeader(http.StatusNotFound)
```

```
        w.Write([]byte(`{"message": "not found"}`))
    }
}
```

Если наш сервер уже запущен, останавливаем его нажимая CTRL-C.

И заново запускаем, чтобы изменения из файла применились.

```
go run main.go
```

Проверяем, как теперь себя ведет reas api, в зависимости от типа запроса.

```
$ curl localhost:8000
{"message": "get called"}
$ curl localhost:8000 -X POST
{"message": "post called"}
$ curl localhost:8000 -X PUT
{"message": "put called"}
$ curl localhost:8000 -X DELETE
{"message": "delete called"}
$ curl localhost:8000/test
{"message": "not found"}
```

Можно заметить, что мы используем структуру нашего сервера в том числе с помощью метода.

Go команда знала что это будет не удобно, и дала нам `HandleFunc` метод в `http` пакете который позволяет нам передавать функцию которая имеет ту же подпись что и `ServerHTTP` и может обслуживать маршруты.

Немного упростим наш код.

```
package main

import (
    "log"
    "net/http"
)
```

```
func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    switch r.Method {
    case "GET":
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(`{"message": "get called"}`))
    case "POST":
        w.WriteHeader(http.StatusCreated)
        w.Write([]byte(`{"message": "post called"}`))
    case "PUT":
        w.WriteHeader(http.StatusAccepted)
        w.Write([]byte(`{"message": "put called"}`))
    case "DELETE":
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(`{"message": "delete called"}`))
    default:
        w.WriteHeader(http.StatusNotFound)
        w.Write([]byte(`{"message": "not found"}`))
    }
}

func main() {
    http.HandleFunc("/", home)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Функциональность не должна измениться.

Gorilla Mux

`net/http` со встроенными методами - это отлично. Мы можем написать сервер без внешних библиотек. Но `net/http` имеет ограничения. Нет прямого пути взаимодействия с параметрами. Так же как и с запросами, нам нужно обрабатывать и параметры запроса в ручную.

Gorilla Mux очень популярная библиотека которая работает отлично по сравнению с `net/http` пакетом и помогает нам с некоторыми вещями при создании api.

Использование Gorilla Mux

Чтобы установить этот пакет будем использовать `get`. Под капотом `go get` использует `git`.

В папке где лежит `go.mod` и `main.go`, запустите:

```
go get github.com/gorilla/mux
```

Изменим наш код следующим образом.

```
package main

import (
    "log"
    "net/http"

    "github.com/gorilla/mux"
)

func home(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    switch r.Method {
    case "GET":
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(`{"message": "get called"}`))
    case "POST":
        w.WriteHeader(http.StatusCreated)
        w.Write([]byte(`{"message": "post called"}`))
    case "PUT":
        w.WriteHeader(http.StatusAccepted)
        w.Write([]byte(`{"message": "put called"}`))
    case "DELETE":
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(`{"message": "delete called"}`))
    default:
        w.WriteHeader(http.StatusNotFound)
        w.Write([]byte(`{"message": "not found"}`))
    }
}
```

```
func main() {
    r := mux.NewRouter()
    r.HandleFunc("/", home)
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Выглядит, так как-будто ничего не изменилось за исключением строки с импортом и строки под номером 32.

HandleFunc HTTP Методы

Но теперь мы можем делать немного больше с помощью `HandleFunc`, к примеру создание каждого обработчика функции для определенного метода HTTP. Выглядить это будет следующим образом.

```
package main

import (
    "log"
    "net/http"

    "github.com/gorilla/mux"
)

func get(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "get called"}`))
}

func post(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    w.Write([]byte(`{"message": "post called"}`))
}

func put(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusAccepted)
```

```
    w.Write([]byte(`{"message": "put called"}`))
}

func delete(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "delete called"}`))
}

func notFound(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusNotFound)
    w.Write([]byte(`{"message": "not found"}`))
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/", get).Methods(http.MethodGet)
    r.HandleFunc("/", post).Methods(http.MethodPost)
    r.HandleFunc("/", put).Methods(http.MethodPut)
    r.HandleFunc("/", delete).Methods(http.MethodDelete)
    r.HandleFunc("/", notFound)
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Если вы запустите, то программа должна всё еще делать всё тоже самое. Теперь вы гадаете, что почему этот вариант лучше, елси строчек кода получилось больше? Но подумайте так: Наш код стал гораздо чище, и еще большее понятен.

```
Лучше чище, чем умнее.
```

Подмаршруты

```
func main() {
    r := mux.NewRouter()
    api := r.PathPrefix("/api/v1").Subrouter()
    api.HandleFunc("", get).Methods(http.MethodGet)
    api.HandleFunc("", post).Methods(http.MethodPost)
```

```
    api.HandleFunc("", put).Methods(http.MethodPut)
    api.HandleFunc("", delete).Methods(http.MethodDelete)
    api.HandleFunc("", notFound)
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Всё остается тем же самым за исплючением создания `sub-router`(подмаршрут). Подмаршрут очень полезен, когда нам нужно поддерживать большое количество ресурсов. Он помогает нам группировать содержание, а так же защищает нас от перенабора одного и того же префикса пути.

Пенесем наше `api` в `api/va`. Таким образом мы можем создавать v2 и так далее.

Параметры пути и запроса

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "strconv"

    "github.com/gorilla/mux"
)

func get(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "get called"}`))
}

func post(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    w.Write([]byte(`{"message": "post called"}`))
}

func put(w http.ResponseWriter, r *http.Request) {
```

```
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusAccepted)
    w.Write([]byte(`{"message": "put called"}`))
}

func delete(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "delete called"}`))
}

func params(w http.ResponseWriter, r *http.Request) {
    pathParams := mux.Vars(r)
    w.Header().Set("Content-Type", "application/json")

    userID := -1
    var err error
    if val, ok := pathParams["userID"]; ok {
        userID, err = strconv.Atoi(val)
        if err != nil {
            w.WriteHeader(http.StatusInternalServerError)
            w.Write([]byte(`{"message": "need a number"}`))
            return
        }
    }

    commentID := -1
    if val, ok := pathParams["commentID"]; ok {
        commentID, err = strconv.Atoi(val)
        if err != nil {
            w.WriteHeader(http.StatusInternalServerError)
            w.Write([]byte(`{"message": "need a number"}`))
            return
        }
    }

    query := r.URL.Query()
    location := query.Get("location")

    w.Write([]byte(fmt.Sprintf(`{"userID": %d, "commentID": %d, "location": "%s" }`, userID,
```

```
commentID, location)))
}

func main() {
    r := mux.NewRouter()

    api := r.PathPrefix("/api/v1").Subrouter()
    api.HandleFunc("", get).Methods(http.MethodGet)
    api.HandleFunc("", post).Methods(http.MethodPost)
    api.HandleFunc("", put).Methods(http.MethodPut)
    api.HandleFunc("", delete).Methods(http.MethodDelete)

    api.HandleFunc("/user/{userID}/comment/{commentID}", params).Methods(http.MethodGet)

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Давайте посмотрим на параметры функции в строке 36. Мы обрабатываем оба параметра: путь и запрос.

Теперь вы знаете достаточно, чтобы быть "опасным".

Книжное API

В Kaggle есть набор данных по книгам. Это csv файл с 13000 книгами. Мы будем использовать его для создания нешго api.

books.csv

Файл можно найти выше.

Склонируем репозиторий

В отдельной папке.

```
git clone https://github.com/moficodes/bookdata-api.git
```

Пройдемся по коду

Есть ва пакета внутри кода. Один называется `datastore`, другой - `loader`.

- `lader` - конвертирует csv данные в массив объектов с данными о книге.
- `datastore` работает с доступом к масиву. Обычно это интерфейс который имеет метод.

un app

Из корня репозитория запустите:

```
go run .
```

Точки доступа

У приложения есть несколько точек доступа. Все точки доступа api имеют префикс `/api/v1`.

Чтобы достичь какую либо из них, нужно использовать базовый адрес: `baseurl:8080/api/v1/{endpoint}`

Получить книги для автора:

```
"/books/authors/{author}"
```

Можно указать параметр запроса для пределов `ratingAbove` и `ratingBelow`

Получить книги по названию.

```
"/books/book-name/{bookName}"
```

Можно указать параметр запроса для пределов `ratingAbove` и `ratingBelow`

Получить книгу по ISBN

```
"/book/isbn/{isbn}"
```

Удалить книгу по ISBN

```
"/book/isbn/{isbn}"
```

Создать новую книгу

```
"/book"
```

Revision #6

Created 2021-07-28 12:14:07 UTC by gasick

Updated 2023-04-16 19:30:04 UTC by gasick