GO-09 03: Роутер Gorilla

Описание:

Прошлые задания были посвящены разработке http-серверов исключительно на базе стандартной библиотеки golang. В таком подходе есть свои плюсы, например:

- быстрый и легкий старт (подключили пакет net/http, зарегистрировали обработчик и готово);
- нет необходимости подключать сторонние зависимости;
- достаточно хороший функционал, который может удовлетворить все базовые хотелки.

Но все же многие вещи приходится писать руками, например, парсинг/валидацию параметров запроса или группировку запросов по их методам [GET POST PUT DELETE]. В таких случаях мы прибегаем к помощи сторонних библиотек, у которых функционал гораздо богаче и удобнее. И одной из таких библиотек является gorilla web toolkit (mux).
Ее особенности:

- Реализует интерфейс http.Handler, поэтому является полностью совместимой со стандартным роутером http.ServeMux.
- Умеет группировать запросы.
- Пути в маршрутах могут иметь переменные вида {var} или {var:regexp}, что позволяет очень гибко работать с url параметрами запросов.

Установка

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

Простейший пример

```
package main

import (
    "net/http"
    "github.com/gorilla/mux"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Hello, world!")
}

func main() {
     r := mux.NewRouter()
     r.HandleFunc("/", handler)
     http.Handle("/", r)
```

```
	http.ListenAndServe(":8080", nil)
}
```

Как можно увидеть, из-за того что mux имплементирует интерфейс http.Handler, мы можем использовать его в методе http.Handle.

Переменные

Давайте теперь посмотрим вот на такой пример:

```
func handler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    fmt.Fprinln(w, vars["category"], vars["id"])
}

func main() {
	r := mux.NewRouter()
	r.HandleFunc("/order/{category}/{id:[0-9]+}", handler)
	http.Handle("/", r)
	http.ListenAndServe(":8080", nil)
}
```

gorilla/mux, в отличие от стандартного net/http, умеет явно задавать параметры в URL, а также валидировать их при помощи регулярных выражений. Таким образом, если мы в параметре пути напишем конструкцию вида {any}, тогда в обработчике мы будем иметь ключ с именем any, в котором будет лежать какое-нибудь значение, например, для /order/cars - это будет cars, а для /order/books - books. Если же нам нужно провалидировать какой-нибудь параметр (например, идентификатор ордера может быть только числовым), то мы можем воспользоваться конструкцией {var:reqexp}, где regexp - это регулярное выражение, по которому будет проведена валидация. И если передать в пути /order/cars/12, то все будет хорошо и обработчик займется своим делом, но если передать ему что-то, чего он не ждет, например, /order/cars/bmw, тогда он сразу обработает это как 404 ошибку.

Группировка запросов

Нередко бывает, что в нашем API за каким-нибудь путем скрывается еще целый список разных путей. Самый простой пример - это версионирование API. Давайте представим, что у нас есть API со следующими маршрутами:

- /v1 - это первая версия API, в ней есть такие методы, как /orderCreate, /orderUpdate, /saleStart, /saleStop и другие;
- /v2 - это новая версия уже с обновленными маршрутами (приведенными в порядок или в другой вид) - /order, /sale/start, /sale/stop и другие.

Для того чтобы сделать такое версионирование со стандартной библиотекой, нам нужно было бы прописывать каждый метод до конца.

```
http.HandleFunc("/v1/orderCreate", orderCreate)
http.HandleFunc("/v1/orderUpdate", orderUpdate)
```

```
http.HandleFunc("/v1/saleStart", saleStart)
http.HandleFunc("/v1/saleStop", saleStop)
http.HandleFunc("/v2/order/create", orderCreate)
http.HandleFunc("/v2/order/update", orderUpdate)
http.HandleFunc("/v2/sale/start", saleStart)
http.HandleFunc("/v2/sale/stop", saleStop)
```

Согласимся, что выглядит это не самым удобным образом при том, что API может быть не маленьким, как в примере, а огромным. В gorilla/mux это можно привести в порядок при помощи Subrouter():

```
r := mux.NewRouter()
v1router := r.PathPrefix("/v1").Subrouter()
v1router.HandleFunc("/orderCreate", orderCreate)
v1router.HandleFunc("/orderUpdate/{id:[0-9]+}", orderUpdate)
v1router.HandleFunc("/saleStart/{id:[0-9]+}", saleStart)
v1router.HandleFunc("/saleStop/{id:[0-9]+}", saleStart)

v2router := r.PathPrefix("/v2").Subrouter()

orderRouter := v2router.PathPrefix("/order").Subrouter()
orderRouter.HandleFunc("/", orderCreate)
orderRouter.HandleFunc("/{id:[0-9]+}", orderUpdate)

saleRouter := v2router.PathPrefix("/sale").Subrouter()
saleRouter.HandleFunc("/start/{id:[0-9]}", saleStart)
saleRouter.HandleFunc("/stop/{id:[0-9]+}", saleStart)

http.Handle("/", r)
http.ListenAndServe(":8080", nil)
```

Ограничения обработчиков

Еще одной интересной фишкой gorilla/mux роутера является то, что на каждый обработчик можно накинуть некоторые ограничения.

- Ограничения по HTTP методу:

```
r.HandleFunc("/order", orderCreate).Methods("POST")
r.HandleFunc("/order/{id:[0-9]+}", getOrder).Methods("GET")
r.HandleFunc("/order/{id:[0-9]+}", orderUpdate).Methods("PUT")
r.HandleFunc("/order/{id:[0-9]+}", orderDelete).Methods("DELETE")
```

- Ограничения на схему (протокол http/https):

```
r.HandleFunc("/secure", handler).Schemes("https")
r.HandleFunc("/notSecure", handler).Schemes("http")
```

- Ограничения по хосту (с какого хоста могут быть приняты запросы по конкретному маршруту, по умолчанию - это *, то есть с любого):

```
r.HandleFunc("/onlygoogle", handler).Host("https://google.com")
```

- И даже кастомные:

```
r.MatcherFunc(func(r *http.Request, rm *RouteMatch) bool {
    return r.ProtoMajor == 0
})
```

Полезные ссылки:

- Gorilla web toolkit official website
- Github repo mux
- Маршрутизация gorilla/mux
- Rest API with gorilla/mux
- Routing using gorilla/mux

Задание:

В этом задании нам нужно реализовать CRUD для сущности User в нашем тестовом сервисе.
Условия:

1. Нужно реализовать 4 ручки:

- createUser

Пример запроса по ручке:

```
url --request POST -d
'{"name":"1","email":"example@mail.ru","age":25}' --header
'Content-Type: application/json' http://localhost:8080/user/
``
 Ручка должна возвращать статус `400 BadRequest`, если ей не удастся распарсить тело запроса.
 Ручка должна вернуть статус `500 InternalServerError`, если во время обработки появились какие-нибудь ошибки, не связанные с параметрами.
 Ручка должна вернуть статус `201 Created` и пустое тело.
```

- getUsersList

Пример запроса по ручке:

```
url --request GET http://localhost:8080/user/
```

Ручка должна вернуть статус `200 OK` и массив пользователей. Если пользователей нет, вернуть пустой массив.

- getUserById

    Пример запроса по ручке:

```
url --request GET http://localhost:8080/user/2
```

Ручка должна вернуть статус `400 BadRequest`, если не удалось найти пользователя.
Ручка должна вернуть статус `500 InternalServerError`, если во время обработки появились какие-нибудь ошибки, не связанные с параметрами.
Ручка должна вернуть статус `200 OK` и массив пользователей с элементом в нем.

- deleteUser

    Пример запроса по ручке:

```
url --request DELETE http://localhost:8080/user/2
```

Ручка должна вернуть статус `400 BadRequest`, если не удалось найти пользователя.
Ручка должна вернуть статус `500 InternalServerError`, если во время обработки появились какие-нибудь ошибки, не связанные с параметрами.
Ручка должна вернуть статус `200 OK` и пустое тело.

2. Маршруты должны быть сгруппированы, то есть в коде не должно быть маршрутов вида:

```
router.HandleFunc("/v1/order")
router.HandleFunc("/v1/sale")
router.HandleFunc("/v1/article")

// ...etc
```

3. Нужно воспользоватся валидацией параметров там, где это необходимо.
4. Нужно воспользоватся репозиториями из пакета module09/internal/repositories/user.

Порядок действий:

1. В вашем проекте module09 сделайте новую ветку 03_task.
2. Реализуйте логику обработчиков в пакете module09/internal/handlers/user.
3. В пакете module09/internal/routers/gorilla создайте роутер и зарегистрируйте нужные маршруты.
4. В пакете module09/cmd/app подключите роутер и запустите сервер.
5. Проверьте работоспособность.

6. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки 03_task.