GO-09 05: Middleware

Описание:

Теперь давайте поговорим о том, что мы затронули уже ранее в наших заданиях, - это middleware. Иногда нам бывает необходимо выполнять какую-нибудь логику непосредственно перед обработкой наших запросов. Например, проставить каждому реквесту идентификатор (fingerprint), или провалидировать запрос по IP-адресу клиента (если IP не входит в наш whitelist, тогда не пускать его дальше в обработчики), или какое-нибудь другое действие, которое будет вызвано перед обработкой запроса. В этом нам и помогут middlewares!
Middleware - в переводе означает промежуточный, что крайне доходчиво объясняет принцип его работы. Middleware можно назвать что угодно, это что-то, что стоит посередине выполнения. В рамках http middleware будет являться то, что стоит после обработки пути маршрута вашего вызова, но перед самим обработчиком этого маршрута.

```
Router => Middleware Handler => Application Handler
```

Создание middleware обработчика
Создавать такие обработчики в go достаточно просто, главное придерживаться определенного интерфейса.

```
type MiddlewareHandler = func(next http.Handler) http.Handler
```

Нам нужно сделать функцию, которая принимает в себя обработчик http.Handler и в ответе отдает новый обработчик http.Handler. Таким образом мы как будто оборачиваем наш обработчик в другую функцию со своей логикой выполнения (тем, кто знаком с ООП, это может напомнить декораторы). Теперь давайте посмотрим на реализацию.

```
func SimpleMiddleware(next http.Handler) http.Handler {
	return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
		// ... you code
		fmt.Println("Hello from middleware!")
		next.ServeHTTP(w, r)
	})
}
```

Здесь мы оборачиваем любой входящий через параметр next обработчик в функцию, которая что-то печатает в консоль, а затем вызывает next.ServeHTTP(w, r), тем самым отдавая процесс выполнения дальше по цепочке. Важно понимать, что это работает и в обратную сторону, то есть после вызова next.ServeHTTP(w, r) ход выполнения опять вернется к функции SimpleMiddleware, и дальше мы тоже можем выполнить какую-нибудь работу уже после выполнения запроса.

```
func SimpleMiddleware(next http.Handler) http.Handler {
	return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
		fmt.Fprintln(w, "before handler")
		next.ServeHTTP(w, r)
		fmt.Fprintln(w, "after handler")
	})
}

func main() {
	http.Handle("/middleware",
SimpleMiddleware(http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
		fmt.Fprintln(w, "in handler")
	})))

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

После отправки запроса мы должны увидеть следующую картину.

```
→ curl localhost:8080/middleware
Hello, middleware
in handler
Hello from middleware!
```

Middleware chain

Мы также можем оборачивать middleware в другие middleware, тем самым создавая цепочки вызовов и расширяя функционал еще больше.

```
func FirstMiddleware(next http.Handler) http.Handler {
	return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
		fmt.Fprintln(w, "First middleware execute")
		next.ServeHTTP(w, r)
	})
}

func SecondMiddleware(next http.Handler) http.Handler {
	return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
		fmt.Fprintln(w, "Second middleware execute")
```

```
            next.ServeHTTP(w, r)
        })
}

func Handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "handler execute")
}

func main() {
    hanler :=
FirstMiddleware(SecondMiddleware(http.HandlerFunc(Handler)))
        http.Handle("/middleware", hanler)
        http.ListenAndServe(":8080", nil)
}
```

Теперь при запросе мы увидим следующее:

```
→ curl localhost:8080/middleware
First middleware execute
Second middleware execute
handler execute
```

Другим вариантом вызова большого количества middleware является создание executor, который итеративно или в нужном порядке вызовет у себя все middleware, которые мы ему передадим. Реализовать такой executor можно следующим образом:

```
func MiddlewareExecutor(h http.Handler, middleware
...func(http.Handler) http.Handler) http.Handler {
    for _, mw := range middleware {
        h = mw(h)
    }

    return h
}

func FirstMiddleware(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
            fmt.Fprintln(w, "First middleware execute")
            next.ServeHTTP(w, r)
        })
}
```

```
func SecondMiddleware(next http.Handler) http.Handler {
     return http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
          fmt.Fprintln(w, "Second middleware execute")
          next.ServeHTTP(w, r)
     })
}

func main() {
    http.Handle("/middleware", Middleware(
        http.HandlerFunc(Handler),
        FirstMiddleware,
        SecondMiddleware,
    ))

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

Такой вариант выглядит намного красивее, чем бесконечно оборачивать функции в функции.

Роутеры

Во многих роутерах есть реализации middleware executor и чаще всего не надо будет писать их самостоятельно или оборачивать функции в функции. Например, в chi или gorilla/mux достаточно будет передать свой middleware обработчик в метод r.Use(handler).

Пример с chi:

```
// Ко всем обработчикам

func main() {
    r := chi.NewMux()
     r.Use(FirstMiddleware)
     r.Use(SecondMiddleware)
     r.Get("/middleware", Handler)
     http.ListenAndServe(":8080", r)
}


// Или к конкретному обработчику

func main() {
```

```
    r := chi.NewMux()
     r.With(FirstMiddleware, SecondMiddleware).Get("/middleware",
Handler)
     http.ListenAndServe(":8080", r)
}
```

Пример с gorilla/mux

```
// Ко всем обработчикам

func main() {
     r := mux.NewRouter()
     r.Use(FirstMiddleware)
     r.Use(SecondMiddleware)
     r.HandleFunc("/middleware", Handler)

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

// Или к конкретному обработчику

func main() {
     r := mux.NewRouter()
     r.HandleFunc("/middleware",
FirstMiddleware(SecondMiddleware(Handler)))
     http.Handle("/", r)
     http.ListenAndServe(":8080", nil)
}
```

Third-Party Middleware

Мы можем использовать в качестве middleware что угодно, главное, чтобы это удовлетворяло нужному интерфейсу, а это, в свою очередь, значит, что мы можем использовать не только middleware, который написали самостоятельно, но и различные пакеты от сторонних разработчиков. Например, в задании по роутеру Chi мы говорили про его модули, так вот эти модули все являются middleware и поэтому могут быть использованы нами в качестве промежуточной логики.

```
package main

import (
     "net/http"
     "fmt"
```

```
	"github.com/go-chi/chi"
	"github.com/go-chi/chi/middleware"
)

func Handler(w http.ResponseWriter, r *http.Request) {
	fmt.Fprintln(w, "handler execute")
}

func main() {
    r := chi.NewMux()
	r.Use(middleware.Logger)
	r.Use(middleware.RequestID)
	r.Get("/middleware", Handler)
	http.ListenAndServe(":8080", r)
}
```

Теперь мы умеем логировать все наши запросы и проставлять каждому запросу уникальный идентификатор.

Полезные ссылки:

- Middleware (Basic)
- Chi github more examples for middlewares
- Apply Middleware to Your Route Handlers
- Making and Using HTTP Middleware

Задание:

В этом задании мы предлагаем вам на основе middleware построить базовую аутентификацию.
Условия:

- Аутентификация должна быть построена на базе Basic auth.
- Для проверок используйте креды администратора из module09/internal/constants.
- Middleware должна быть активирована не на все обработчики, а только на create и delete.
- Если аутентификация не прошла, то обработчик должен отдавать статус 401 StatusUnauthorized и не пускать пользователя дальше.

Порядок действий:

1. В вашем проекте module09 сделайте новую ветку 05_task.

2. В пакете module09/internal/routers/chi или module09/internal/routers/gorilla создайте middleware и подключите к роутеру.
3. Проверьте работоспособность.
4. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки 05_task.