GO-09 02: Работа с параметрами

Описание:

В прошлом задании мы научились поднимать простейший сервер на go, а также обрабатывать пустые http запросы. В этом же задании мы научимся работать с параметрами запросов, как, например, body, url query, form и т.д. Мы посмотрим, как извлекать из запроса нужную нам информацию и как отправлять более сложные ответы с изменением заголовков или формата ответа.
Давайте сразу смотреть все на примерах.
URL Query параметры
Самым простым способом передачи каких-нибудь данных в запрос является передача URL-параметров. Мы можем передавать их прямо в строку запроса без каких-либо сторонних манипуляций. Например:

```
→ curl http://my-site.com/users?name=ilya&age=25
```

Для того чтобы обработать такие параметры, мы можем обратиться к аргументу http.Request, который находится в наших обработчиках. В нем есть поле URL, оно отвечает за адресную строку, которую мы передали в запросе curl [адресная строка]. У этого поля есть множество разных методов таких, как Hostname(), Port(), EscapePath(), Query(), которые могут отдавать разные части URL-адреса.

```
func main() {
	http.HandleFunc("/order/status", func(w http.ResponseWriter, r
*http.Request) {
		orderId := r.URL.Query().Get("id")
		status := orderStatus(orderId)

		fmt.Fprintf(w, status)
	})

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

Как видно из примера выше, у нас есть обработчик /order/status, который должен отдать статус заказа в зависимости от его идентификатора. Сам идентификатор заказа обработчик забирает из переданного ему в запрос URL-адреса через метод r.URL.Query().Get("id"). Теперь мы можем передавать в адресе запроса нужный идентификатор и видеть статус заказа.

```
→ curl localhost:8080/order/status?id=1
active
```

```
→ curl localhost:8080/order/status?id=2
pending

→ curl localhost:8080/order/status?id=3
cancel
```

Важно помнить, что метод Get("id") возвращает просто строку и он не будет проверять, есть этот параметр в запросе или запрос вообще без параметров. Вообще метод URL.Query() парсит url параметры в отдельную мапу вида map[string][]string, поэтому никто не запретит нам вызвать параметры вот так: r.URL.Query()["id"] - и получить тот же результат. Напротив, в отличие от вызова метода Get("id") мы теперь сможем проверить, а есть ли вообще параметры с ключом ID в запросе или нет.

```
http.HandleFunc("/order/status", func(w http.ResponseWriter, r
*http.Request) {
	orderId, ok := r.URL.Query()["id"]
	if !ok {
		panic("id not found")
	}

	status := orderStatus(orderId)
	fmt.Fprintf(w, status)
})
```

Таким же методом мы можем передавать туда не один параметр, а сразу несколько.

```
http.HandleFunc("/order/status", func(w http.ResponseWriter, r
*http.Request) {
	orderIds, ok := r.URL.Query()["ids[]"]
	if !ok {
		panic("ids not found")
	}

	for _, id := range orderIds {
		status := orderStatus(id)
		fmt.Fprintln(w, status)
	}
})

→ curl "localhost:8080/order/status?ids[]=1&ids[]=2&ids[]=3"
active
```

```
pending
cancel
```

Если же мы захотим как-то по-особенному принимать массивы из данных, например, localhost:8080/order/status?ids[]=1,2,3, в таком случае парсингом мы будем заниматься самостоятельно, например, через пакет json.

FormData

Другим способом передать в запрос какие-нибудь параметры является передача их через форму FormData. Для этого мы можем написать форму на HTML:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Ввод данных</title>
    </head>
    <body>
        <h2>Ввод данных</h2>
        <form method="POST" action="/send/email">
                <label>Address</label><br>
                <input type="text" name="email" /><br><br>

                <label>Message</label><br>
                <input type="text" name="message" /><br><br>

                <label>File</label><br>
                <input type="file" name="file" /><br><br>

                <input type="submit" value="Отправить" />
        </form>
    </body>
</html>
```

Или отправлять их через программные средства (везде обычно по-разному, вот пример для curl):

```
→ curl --request GET 'localhost:8080/send/email' \
      --form 'email=example@example.com' \
      --form 'message=hello world'
      --form 'file=@test.txt'
```

На стороне сервера прием таких параметров мало чем отличается от приема query параметров из URL.

```
http.HandleFunc("/send/email", func(w http.ResponseWriter, r
*http.Request) {
	email := r.FormValue("email")
	message := r.FormValue("message")

	sendMessage(email, message)
})
```

Разве что обращаемся мы теперь к объекту Form, а не к объекту URL, но они оба имеют одинаковый интерфейс map[string][]string. Но есть все же одно важное отличие: формат multipart/form-data, кроме обычных значений, также умеет передавать и файлы, в свою очередь, мы можем легко принимать и файлы при помощи метода FormFile("key").

```
http.HandleFunc("/upload", func(w http.ResponseWriter, r
*http.Request) {
	file, meta, err := r.FormFile("file")
	if err != nil {
		panic(err)
	}
	defer file.Close()
	// ... process file
})
```

Http Body

Еще одним вариантом передачи параметров является передача тела в запрос.

Тело (или Body) - это чистые необработанные данные, которые мы передаем в запрос, а обработчик уже разбирает эти байты информации так, как ему нужно (будь то JSON, файл или обычный текст).

Обработчик в go хранит тело запроса в поле r.Body и для того, чтобы работать с данными, нам нужно сначала считать все байты из этого поля.

```
http.HandleFunc("/bodyparse", func(w http.ResponseWriter, r
*http.Request) {
	data, err := ioutil.ReadAll(r.Body)
	if err != nil {
		panic(err)
	}
	r.Body.Close()
	// ... process data
})
```

Вот, в принципе, и все, дальше можно парсить данные (байты), как нам удобно.

Например, в JSON:

```
type MyObject struct {
	Field1 string
	Field2 int
	Field3 bool
}

http.HandleFunc("/json", func(w http.ResponseWriter, r *http.Request)
{
	data, err := ioutil.ReadAll(r.Body)
	if err != nil {
		panic(err)
	}
	r.Body.Close()

	var obj MyObject
	if err := json.Unmarshal(data, &obj); err != nil {
		panic(err)
	}

	fmt.Println(obj.Field1)
})
```

Метаданные

В объекте http.Request также хранятся вспомогательные данные о запросе, такие как:

- Заголовки r.Header - через метод r.Header.Get("key") возвращают строку, а само поле имеет интерфейс вида map[string][]string.
- Куки (Cookie) - через методы r.Cookie("key") и r.Cookies() возвращают объект http.Cookie, в котором есть вся информация о куке (имя, значение, время, после которого кука считается устаревшей, и другие).
- Метод запроса через r.Method (GET, POST, PUT или другие).
- Вся информация об адресе (хост r.URL.Hostname(), порт r.URL.Port(), путь r.URL.EscapePath()).
- Версия протокола. При помощи метода ProtoAtLeast(majorVersion int, minorVersion int) можно проверить версию http протокола в запросе.
- Информация о защищенном соединении через поле r.TLS.

Basic auth

Одним из простейших способов аутентификации (проверки на то, что тот, кто посылает вам запрос, имеет для этого достаточно прав) является механизм Basic Auth. На самом деле, механизм представляет из себя передачу в заголовке Authorization: Basic [creds] информации о пользователе username и password. На стороне обработчика мы можем принять эту информацию двумя способами.

1. Честно взять информацию из заголовка. Здесь проблема состоит в том, что данные в заголовке не всегда приходят в открытом виде и парсить строку придется самостоятельно.

```
http.HandleFunc("/secure", func(w http.ResponseWriter, r
*http.Request) {
	auth := r.Header.Get("Authorization")
	fmt.Println(auth)
})
```

Если посмотреть, в каком виде нам прилетает заголовок:

```
→ curl 'localhost:8080/secure' --header 'Authorization: Basic
YWE6YWFh'
Basic YWE6YWFh
```

то можно сделать вывод, что парсить это будет не совсем удобно.

2. Использовать стандартный метод r.BasicAuth(), который вернет нам непосредственно username и password в открытом виде:

```
http.HandleFunc("/secure", func(w http.ResponseWriter, r
*http.Request) {
	name, pass, ok := r.BasicAuth()
	if !ok {
		panic("auth header not found")
	}

	fmt.Println(name, pass)
})
```

Response

Все это время мы либо писали что-то в консоль, либо отдавали простые строковые ответы через fmt.Fprint. Теперь давайте попробуем вернуть что-нибудь посложнее:

```
type User struct {
	Name string
	Age  int
}

http.HandleFunc("/user", func(w http.ResponseWriter, r *http.Request)
{
	user := User{Name: "Ivan", Age: 25}
	jsonData, err := json.Marshal(user)
	if err != nil {
```

```
            http.Error(w, "ivalid json marshal",
http.StatusInternalServerError)
            return
      }

      w.Header().Set("Content-Type", "application/json")
      w.WriteHeader(http.StatusOK)
      _, _ = w.Write(jsonData)
})
```

На самом деле, отправка даже сложных ответов типа json или html в go не является сложной задачей. Главное - привести данные в нужный для отправки вид (мы это сделали, конвертировав нашу структуру в json и поставив правильный контент-заголовок w.Header().Set("Content-Type", "application/json"), после чего просто пишем данные в соединение через w.Write(data):

```
→ curl 'localhost:8080/user'
{"Name":"Ivan","Age":25}
```

Полезные ссылки:

- GO Строка запроса и отправка форм
- Serializing Array Values In URLs For Golang
- Upload file go
- HTTP Doc go

Задание:

В этом задании нам надо реализовать ручку (API endpoint / точка подключения) для получения списка кошельков пользователей от имени администратора сервиса.
Логика ручки:

- Должна присутствовать базовая аутентификация Basic Auth, в случае ее отсутствия ручка должна отдавать статус 401 Unauthorized.
- Username и Password должны совпадать с эталонными, в противном случае ручка должна отдавать статус 401 Unauthorized.
- Должен присутствовать Query параметр ids (может быть массивом), в случае его отсутствия ручка должна отдавать статус 400 BadRequest.
- В ответ на реквест должен вернуться массив из информации по всем кошелькам, по которым были переданы идентификаторы в запросе. Информация должна быть в JSON формате.

- В случае возникновения посторонних ошибок (например, не распарсил JSON) ручка должна отдавать статус 500 InternalServerError.
- Пустышка для логики ручки находится в пакете module09/internal/handlers/wallet.
- Репозиторий с данными и методы для их получения находятся в пакете module09/internal/repositories/wallet.
- Эталонные AdminName и AdminPass находятся в пакете module09/internal/constants.
- Сущность Wallet находится в пакете module09/internal/entities.

Порядок действий:

1. В вашем проекте module09 сделайте новую ветку 02_task.
2. В пакете module09/cmd/app зарегистрируйте обработчик /wallet, функция обработчика находится в пакете module09/internal/handlers/wallet.
3. Заполните функцию Handle в пакете module09/internal/handlers/wallet логикой.
4. Проверьте работоспособность.
5. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки 02_task.