

Основы веб-программирования в Go

Последнее обновление: 28.02.2018

Особую сферу разработки на Go занимает веб-программирование, которое представляет создание различных веб-приложений, в том числе различных веб-сайтов, веб-сервисов. Изначально Go не предназначался для веб-программирования и более того даже не рассматривался разработчиками в данной роли, и потребовалось некоторое время, прежде чем Go стал применяться в этой области. В то же время это не значит, что Go подойдет для всех веб-проектов. Многие веб-фреймворки содержат кучу готового функционала из коробки, который облегчает создание приложений. В Go подобного нет. Go предоставляет в основном только самые базовые вещи, на основании которых можно строить более сложные конструкции. Go в плане веб-разработки подойдет прежде всего для таких проектов, которые требуют очень высокой производительности, либо когда важны какие-то определенные возможности Go, например, параллельная обработка запросов.

Первое веб-приложение

Основной функционал для создания веб-приложений в Go расположен в пакете `net/http`. В частности, чтобы запустить веб-приложение, которое могло бы принимать входящие запросы, достаточно вызвать функцию `http.ListenAndServe`:

```
func ListenAndServe(addr string, handler Handler) error
```

Первый параметр указывает, запросы по какому пути будут обслуживаться веб-приложением. Второй параметр определяет обработчик запроса в виде интерфейса `Handler`. Этот интерфейс определяет функцию `ServeHTTP`:

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

Функция `ServeHTTP` принимает два параметра. Первый параметр - объект `ResponseWriter` представляет поток ответа, в который мы можем записать любые данные, которые мы хотим отправить в ответ пользователю. Второй параметр - `Request` инкапсулирует всю информацию о запросе.

Например, определим простейшее веб-приложение. Для этого создадим на жестком диске каталог для хранения файлов с исходным кодом на языке Go, допустим, он будет называться `golang`. И определим в этом каталоге файл `server.go` со следующим кодом:

```
package main  
import (  
    "fmt"  
    "net/http"  
)  
type msg string  
func (m msg) ServeHTTP(resp http.ResponseWriter, req *http.Request) {  
    fmt.Fprint(resp, m)  
}  
func main() {  
    msgHandler := msg("Hello from Web Server in Go")  
    fmt.Println("Server is listening...")  
    http.ListenAndServe("localhost:8181", msgHandler)  
}
```

В данном случае определен кастомный тип `msg` на основе типа `string`, который реализует метод `ServeHTTP` интерфейса `Handler`. В самом методе с помощью вызова `fmt.Fprint(resp, m)` в поток ответа `resp` пишется сообщение, которое хранится в строке `m`. Таким образом, пользователю отправляется ответ.

В функции `main` определяется объект `msgHandler`:

```
msgHandler := msg("Hello from Web Server in Go")
```

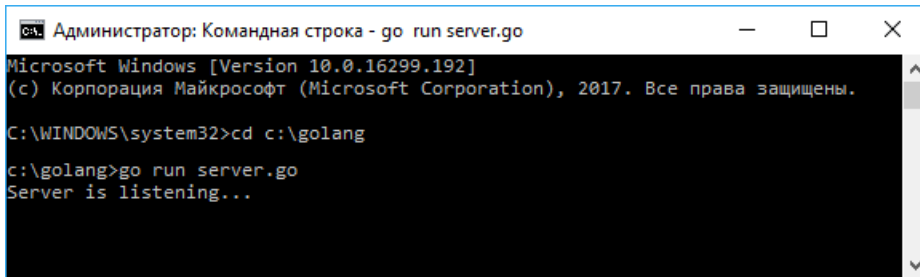
По сути это строка, но тем не менее этот объект реализует интерфейс `Handler`.

Далее для обработки запросов передаем этот объект в качестве второго параметра в функцию `http.ListenAndServe`:

```
http.ListenAndServe("localhost:8181", msgHandler)
```

Первый параметр указывает, что веб-приложение будет запускаться по адресу `localhost:8181`. Номер порта необязательно должен быть 8181. Это может быть любой незанятый порт.

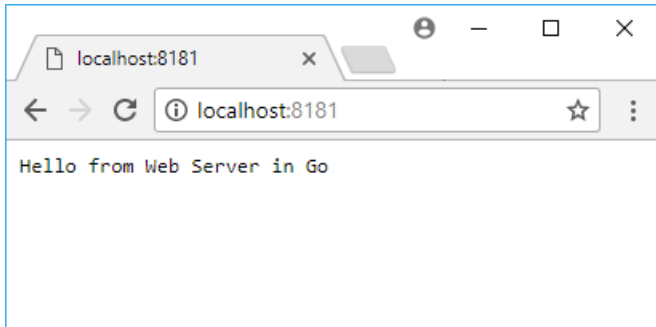
Запустим веб-приложение:



```
Администратор: Командная строка - go run server.go
Microsoft Windows [Version 10.0.16299.192]
(c) Корпорация Майкрософт (Microsoft Corporation), 2017. Все права защищены.

C:\WINDOWS\system32>cd c:\golang
c:\golang>go run server.go
Server is listening...
```

И затем в любом браузере обратимся по адресу "http://localhost:8181/":



Маршрутизация

Последнее обновление: 28.02.2018

Функция HandleFunc

Система маршрутизации позволяет сопоставить определенные запросы с определенными ресурсами внутри веб-приложения. Для создания протейшей системы маршрутизации в приложении может применяться функция `HandleFunc()`.

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

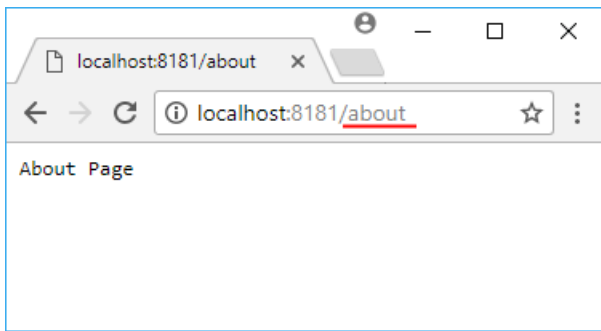
Ее преимущество состоит в том, что она позволяет указать маршруты для обработки. Первый параметр функции представляет маршрут, который будет обрабатываться данной функцией. А второй - функция handler, которая будет обрабатывать запрос. Она также принимает два параметра: `ResponseWriter` - поток ответа и `*Request` - информацию о запросе.

Например, определим следующий код в файле сервера:

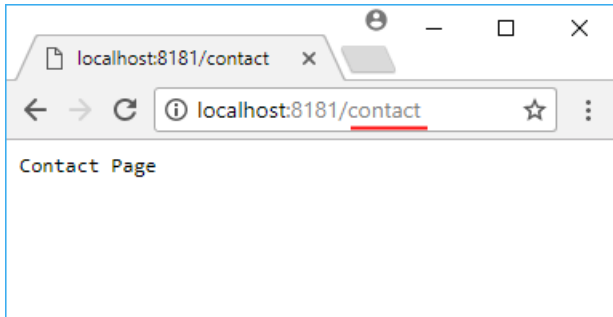
```
package main
import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/about", func(w http.ResponseWriter, r *http.Request){
        fmt.Fprint(w, "About Page")
    })
    http.HandleFunc("/contact", func(w http.ResponseWriter, r *http.Request){
        fmt.Fprint(w, "Contact Page")
    })
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request){
        fmt.Fprint(w, "Index Page")
    })
    fmt.Println("Server is listening...")
    http.ListenAndServe("localhost:8181", nil)
}
```

Первый аргумент функции `HandleFunc` - `"/about"`, указывает, что эта функция будет обрабатывать запросы по пути `"/about"`, то есть по адресу `http://localhost:8181/about`. Второй параметр указывает, что в ответ на запрос по этому пути пользователю будет отправляться строка `"About Page"`.



Соответственно запрос по пути `"/contact"` будет обрабатываться функцией `http.HandleFunc("/contact", ...)`, а запрос к корню веб-сайта будет обрабатываться функцией `http.HandleFunc("/", ...)`.



Причем подобным образом мы можем сопоставлять маршруты не только с функциями, которые возвращают некоторое содержимое в виде строки, но также мы можем сопоставлять маршруты со статическими файлами. Например, определим в папке с файлом сервера html-страницу `hello.html` со следующим кодом:

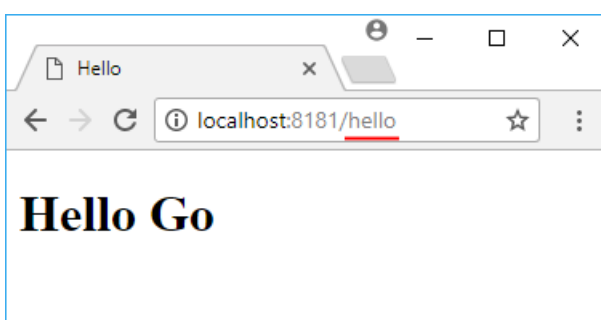
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Index</title>
  </head>
  <body>
    <h1>Index</h1>
  </body>
</html>
```

Изменим файл сервера:

```
package main
import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/hello", func(w http.ResponseWriter, r *http.Request){
        http.ServeFile(w, r, "hello.html")
    })
    http.HandleFunc("/about", func(w http.ResponseWriter, r *http.Request){
        fmt.Fprint(w, "About Page")
    })
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request){
        fmt.Fprint(w, "Index Page")
    })
    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

С помощью функции `http.ServeFile()` при запросе по пути `"/hello"` будет отправляться файл `hello.html`:



Функция Handle

Еще один способ определения маршрутов и сопоставления их с обработчиками представляет функция `http.Handle` :

```
func Handle(pattern string, handler Handler)
```

Например, определим у сервера следующий код:

```
package main
import (
    "fmt"
    "net/http"
)

type httpHandler struct{
    message string
}

func (h httpHandler) ServeHTTP(resp http.ResponseWriter, req *http.Request) {
    fmt.Fprint(resp, h.message)
}

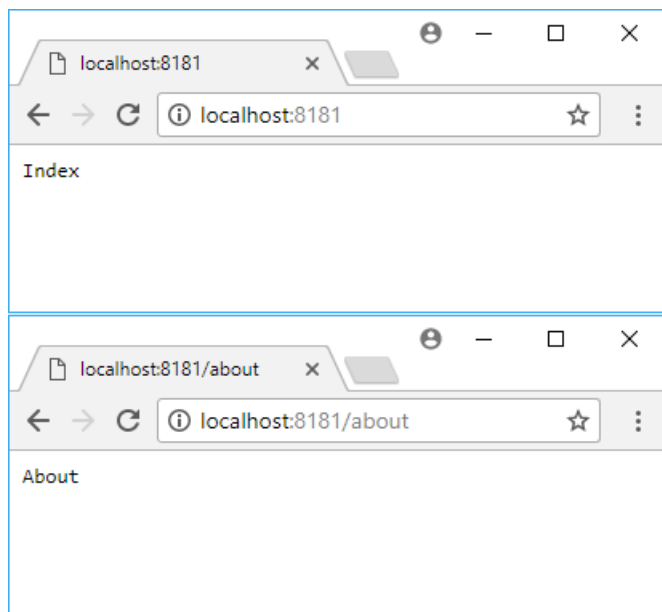
func main() {

    h1 := httpHandler{ message:"Index"}
    h2 := httpHandler{ message:"About"}

    http.Handle("/", h1)
    http.Handle("/about", h2)

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

В данном случае в роли интерфейса Handler, который обрабатывает запрос, выступает структура `httpHandler`:



Статические файлы

Последнее обновление: 04.03.2018

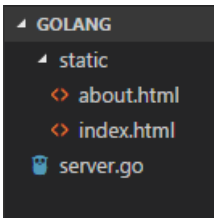
Функция `http.FileServer`

Содержимое веб-приложения или веб-сайта нередко определяется в виде статических html-страниц. Для них не нужен какой-то дополнительный рендеринг на стороне сервера. Для прямой отправки статических файлов в пакете `http` определена функция `FileServer`, которая возвращает объект `Handler`:

```
func FileServer(root FileSystem) Handler
```

В качестве параметра она принимает путь к каталогу со статическими файлами.

Например, определим в каталоге с исходным файлом на Go папку, которую назовем `static`. Создадим в ней два статических файла: `index.html` и `about.html`.



Определим в файле index.html следующий код:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Index</title>
  </head>
  <body>
    <h1>Index</h1>
  </body>
</html>
```

А в файле about.html следующий код:

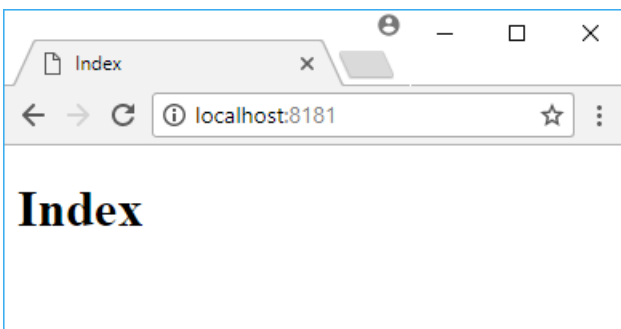
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>About</title>
  </head>
  <body>
    <h1>About</h1>
  </body>
</html>
```

В главном файле server.go определим следующий код:

```
package main
import (
    "fmt"
    "net/http"
)
func main() {
    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", http.FileServer(http.Dir("static")))
}
```

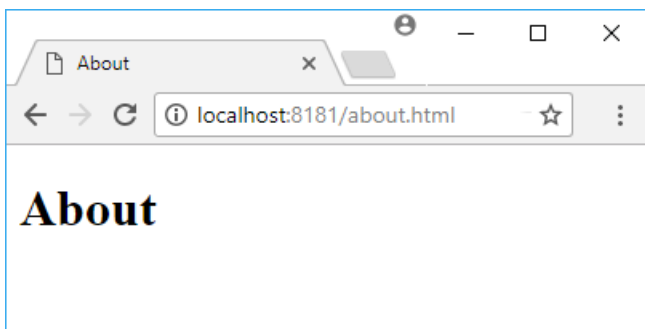
В функцию `http.FileServer()` передается путь к файлам, который определяется функцией `http.Dir()`.

Запустим приложение и обратимся по адресу `http://localhost:8181`



Путь к корню веб-сайта автоматически сопоставляется с файлом index.html - это все равно, если бы мы обратились по адресу `http://localhost:8181/index.html`

Также обратимся по адресу `http://localhost:8181/about.html`. В этом случае мы получим содержимое файла about.html



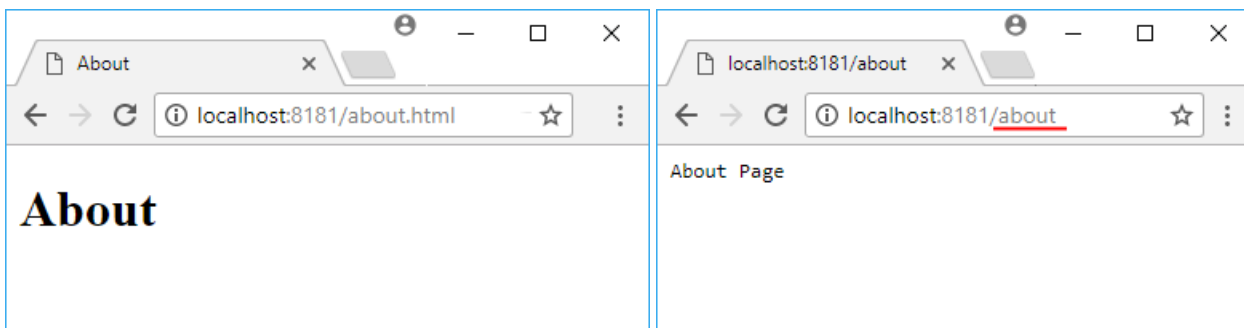
В то же время такой подход довольно ограничен, так как в данном случае сервер отдает только статические файлы. Однако, как правило, возникает необходимость именно в генерации динамического контента. И в этом случае мы можем поступить следующим образом:

```
package main
import (
    "fmt"
    "net/http"
)

func main() {
    fs := http.FileServer(http.Dir("static"))
    http.Handle("/", fs)

    http.HandleFunc("/about", func(w http.ResponseWriter, r *http.Request){
        fmt.Fprint(w, "About Page")
    })
    http.HandleFunc("/contact", func(w http.ResponseWriter, r *http.Request){
        fmt.Fprint(w, "Contact Page")
    })
    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

В данном случае с помощью функции `http.Handle("/", fs)` файловый сервер монтируется к пути `"/"`, то есть к корню сайта. И наряду с этим мы также можем определять обработчики для других маршрутов. Таким образом, будет работать как динамическая генерация контента, так и статические файлы:



Функция `http.ServeFile`

Также для отправки файлов можно использовать функцию `http.ServeFile()`. Она отправляет единичный файл по определенному пути. Например, используем ранее определенные файлы `index.html` и `about.html`:

```
package main
import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/about", func(w http.ResponseWriter, r *http.Request){
        http.ServeFile(w, r, "static/about.html")
    })
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request){
        http.ServeFile(w, r, "static/index.html")
    })
    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

Маршрутизация и `gorilla/mux`

Последнее обновление: 01.03.2018

Go предоставляет базовые возможности по маршрутизации. Однако этих возможностей, как правило, было недостаточно, особенно в тех случаях, когда необходимо выделять сегменты из запрошенного адреса URL и каким-то образом обрабатывать их. В этом случае мы можем воспользоваться рядом существующих инструментов, одним из которых является Gorilla . (Официальный сайт <http://www.gorillatoolkit.org/>) Это пакет разработчика специально для упрощения создания веб-приложений на языке Go, который, в свою очередь, включает ряд пакетов:

- gorilla/context : предназначен для создания глобальных переменных из тела запроса
- gorilla/rpc : представляет реализацию протокола RPC-JSON
- gorilla/websocket : реализует протокол WebSocket
- gorilla/schema : позволяет создавать из значений формы единую структуру
- gorilla/securecookie : позволяет создавать зашифрованные куки, которые применяются при аутентификации
- gorilla/sessions : обеспечивает поддержку сессий
- gorilla/mux : позволяет определять более сложные маршруты, которые могут использовать регулярные выражения
- gorilla/reverse : используется для создания регулярных выражений для маршрутов

В данном случае задействуем возможности по созданию маршрутов с помощью gorilla/mux и для установки данного пакета выполним в командной строке/терминале следующую команду:

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

Определим следующий код сервера:

```
package main
import (
    "fmt"
    "net/http"
    "github.com/gorilla/mux"
)

func productsHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id := vars["id"]
    response := fmt.Sprintf("Product %s", id)
    fmt.Fprint(w, response)
}

func main() {
    router := mux.NewRouter()
    router.HandleFunc("/products/{id:[0-9]+}", productsHandler)
    http.Handle("/", router)

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

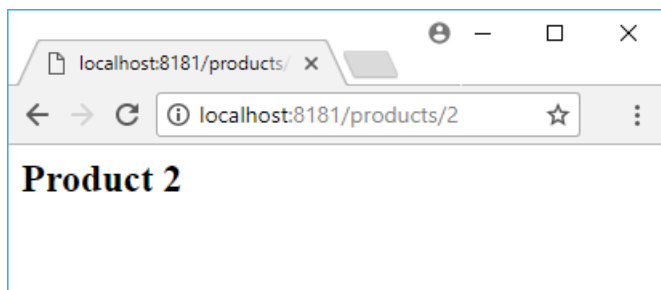
Для определения маршрутов с помощью gorilla/mux применяется функция mux.NewRouter() . У возвращаемого этой функцией объекта мы можем вызвать метод HandleFunc , который сопоставляет маршрут с определенным обработчиком.

Первый параметр представляет шаблон пути запроса. В фигурных скобках мы можем определить параметр в формате имя_параметра:регулярное_выражение . Регулярное выражение определять необязательно, но если оно определено, то параметр должен соответствовать этому выражению. То есть в данном случае параметр id должен представлять числовое значение.

Второй параметр - функция обработчика запросов, по указанному в первом параметре маршруту. Подобная функция должна иметь два параметра: func(w http.ResponseWriter, r *http.Request) .

В данном случае функция обработчика - productsHandler - получает параметры пути запроса через функцию mux.Vars . Затем из полученного объекта можно извлечь название нужного нам параметра: id := vars["id"] . Название параметра здесь то же самое, что в определении маршрута.

В итоге при обращении к приложению по запросу localhost:8181/products/2 мы получим следующий вывод:



Если бы мы захотели бы сделать то же самое, но штатными средствами, которые есть в Go без gorilla/mux, то нам бы пришлось писать дополнительный код для парсинга запрошенного пути.

Подобным образом мы можем использовать несколько параметров:

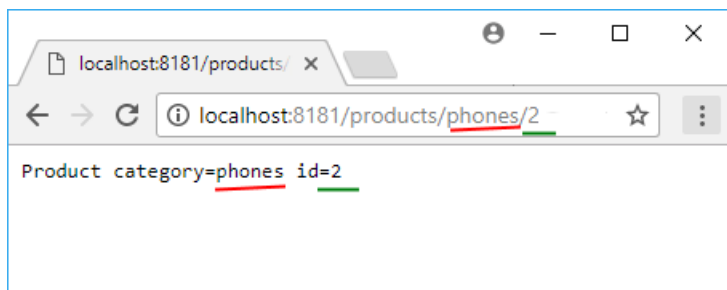
```
package main
import (
    "fmt"
    "net/http"
    "github.com/gorilla/mux"
)

func productsHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id := vars["id"]
    cat := vars["category"]
    response := fmt.Sprintf("Product category=%s id=%s", cat, id)
    fmt.Fprint(w, response)
}

func main() {
    router := mux.NewRouter()
    router.HandleFunc("/products/{category}/{id:[0-9]+}", productsHandler)
    http.Handle("/", router)

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

В данном случае определены два параметра: id и category.



Можно определять несколько маршрутов, которые могут использовать либо различные, либо одни и те же обработчики:

```
package main
import (
    "fmt"
    "net/http"
    "github.com/gorilla/mux"
)

func productsHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id := vars["id"]
    response := fmt.Sprintf("id=%s", id)
    fmt.Fprint(w, response)
}

func indexHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Index Page")
}

func main() {
    router := mux.NewRouter()
    router.HandleFunc("/products/{id:[0-9]+}", productsHandler)
    router.HandleFunc("/articles/{id:[0-9]+}", productsHandler)
    router.HandleFunc("/", indexHandler)
    http.Handle("/", router)
}
```



```

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}

```

Строка запроса и отправка форм

Последнее обновление: 04.03.2018

Строка запроса

Строка запроса представляет набор параметров, которые помещаются в адресе после вопросительного знака. При этом каждый параметр определяет название и значение. Например, в адресе:

localhost:8181/user?name=Sam&age=21

Часть ?name=Sam&age=21 представляет строку запроса, в которой есть два параметра name и age. Для каждого параметра определено имя и значение, которые отделяются знаком равно. Параметр name имеет значение "Sam", а параметр age - значение 21. Друг от друга параметры отделяются знаком амперсанда.

Для получения строки запроса у объекта Request вначале надо получить запрошенный адрес через переменную URL. Далее у адреса вызывается метод Query(), который и возвращает строку запроса.

Например, получим данные строки запроса:

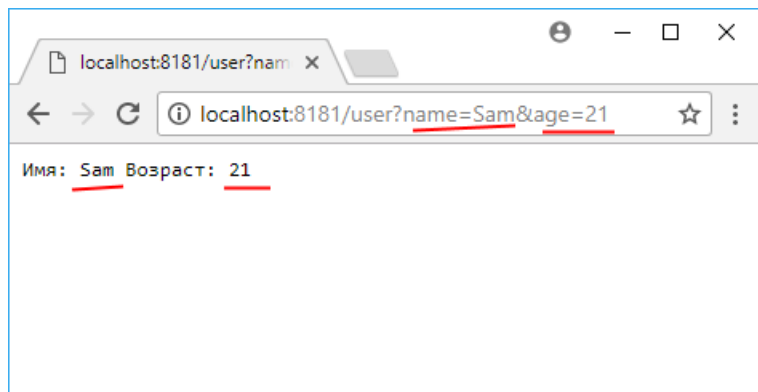
```

package main
import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/user", func(w http.ResponseWriter, r *http.Request){
        name := r.URL.Query().Get("name")
        age := r.URL.Query().Get("age")
        fmt.Fprintf(w, "Имя: %s Возраст: %s", name, age)
    })
    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}

```

Чтобы получить значение отдельного параметра, применяется метод Get(), в который передается имя параметра:



Отправка форм

Рассмотрим, как мы можем получить в Go значения отправленных форм.

Все данные запроса в Go инкапсулируются в объекте http.Request. Через его метод FormValue() можно получить определенные данные, которые отправлены через форму.

```
func (r *Request) FormValue(key string) string
```

Метод FormValue() извлекает данные по ключу из запроса POST и PUT, а также из строки запроса. При этом он всегда возвращает строку.

Например, определим рядом с сервером файл user.html со следующим кодом:

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">

```

```

<title>Ввод данных</title>
</head>
<body>
  <h2>Ввод данных</h2>
  <form method="POST" action="postform">
    <label>Имя</label><br>
    <input type="text" name="username" /><br><br>
    <label>Возраст</label><br>
    <input type="number" name="userage" /><br><br>
    <input type="submit" value="Отправить" />
  </form>
</body>
</html>

```

В данном случае на форме два поля - username и userage. При нажатии на кнопку данные будут отправляться запросом POST по адресу "/postform".

В коде сервера определим получение данных:

```

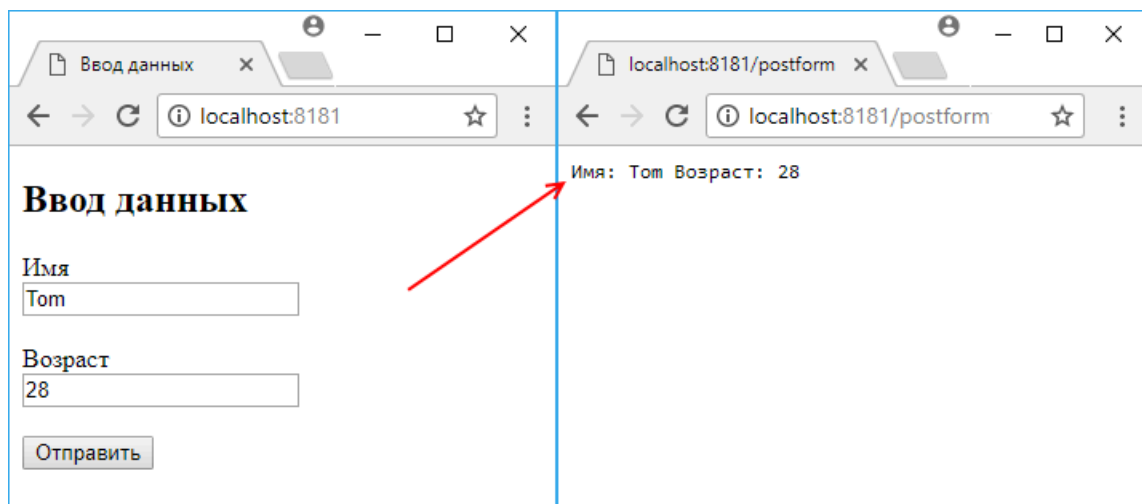
package main
import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request){
        http.ServeFile(w, r, "user.html")
    })
    http.HandleFunc("/postform", func(w http.ResponseWriter, r *http.Request){
        name := r.FormValue("username")
        age := r.FormValue("userage")

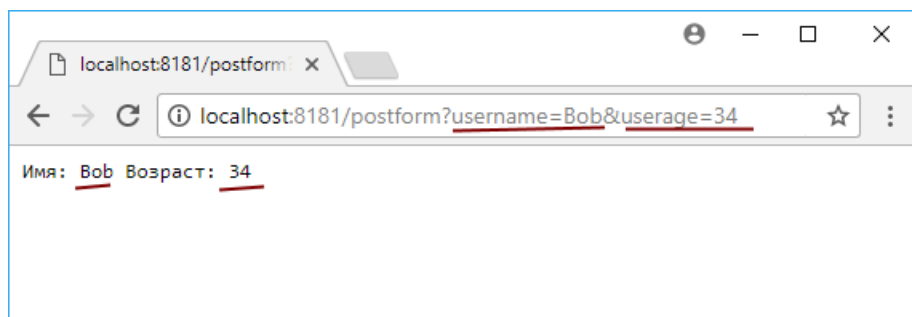
        fmt.Fprintf(w, "Имя: %s Возраст: %s", name, age)
    })
    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}

```

По запросу к корню сайта приложение будет отправлять пользователю файл user.html для ввода данных. При отправке форму на адрес "/postform" приложение будет получать данные. Так как поля на форме называются username и userage, то чтобы получить эти данные, нужно использовать эти имена, типа `name := r.FormValue("username")`.



При этом также FormValue позволяет получить данные из строки запроса, то есть мы, например, можем передать значения для username и userage через строку запроса:



Шаблоны

Определение и использование шаблонов

Последнее обновление: 02.03.2018

Ранее рассматривалось, как в Go отправлять статические файлы, в частности, html-страницы. Определение контента в виде html-страниц довольно удобно: мы используем преимущества html+css+javascript, отделяем представление от основной логики, которая пишется на Go. Однако статические страницы малополезны, когда нам необходимо динамически генерировать некоторый контент на основании различных факторов, например, параметров, переданных через строку запроса. И в этом случае мы можем воспользоваться шаблонами.

Язык Go предоставляет функциональность шаблонов по умолчанию в виде пакета `html/template`.

Используем протейший шаблон:

```
package main
import (
    "fmt"
    "net/http"
    "html/template"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        data := "Go Template"
        tmpl, _ := template.New("data").Parse("<h1>{{ . }}</h1>")
        tmpl.Execute(w, data)
    })

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

С помощью функции `template.New("data")` определяется имя шаблона. Затем для установки самого шаблона используется функция `Parse("<h1>{{ . }}</h1>")`. В данном случае шаблон фактически представляет заголовок `h1`. Но ключевым элементом здесь является двойная пара фигурных скобок `{{ . }}`. Они позволяют вводить в разметку html различные данные. Здесь в качестве данных указана точка. Точка указывает на контекст шаблона - то есть все данные, которые переданы шаблону.

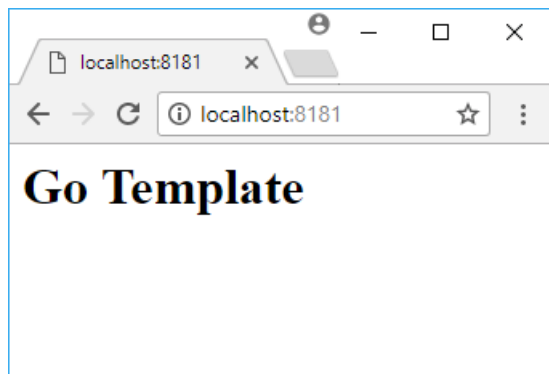
Стоит отметить, что функция `Parse` возвращает два значения: собственно шаблон (в данном случае переменная `tmpl`) и объект ошибки (при ее возникновении). В данном случае объект ошибки не используется, поэтому вместо него идет прочерк.

Чтобы передать шаблону данные, сгенерировать итоговую html-разметку и отправить ее в ответ на запрос, применяется функция `Execute`:

```
tmpl.Execute(w, data)
```

В данном случае переменная `data` представляет строку, и это как раз те данные, которые будут вставляться в шаблон вместо точки `{{ . }}`. Ну а первый параметр - это объект `http.ResponseWriter`, через который отправляются данные.

В итоге при обращении к приложению мы увидим следующий результат:



Шаблон может принимать более сложные данные, которые описываются структурой. Например:

```
package main
import (
    "fmt"
    "net/http"
    "html/template"
)

type ViewData struct{
```

```

    Title string
    Message string
}
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        data := ViewData{
            Title: "World Cup",
            Message: "FIFA will never regret it",
        }
        tmpl := template.Must(template.New("data").Parse(`

<h1>{{ .Title }}</h1>
            <p>{{ .Message }}</p>
        </div>`))
        tmpl.Execute(w, data)
    })

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}

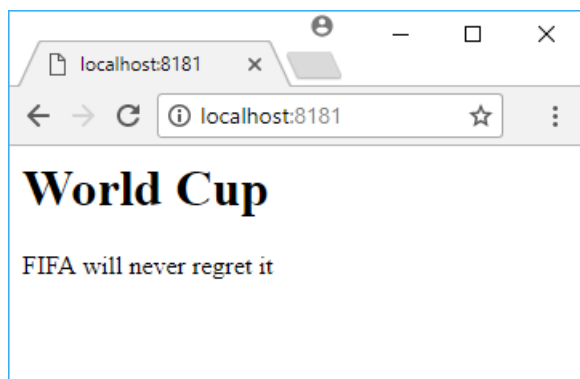

```

Здесь данные, передаваемые в шаблон, описываются структурой `ViewData`, и данная структура будет представлять контекст шаблона. Поэтому чтобы обратиться к отдельным ее переменным, надо после точки указать название переменной: `{{ .Title }}`.

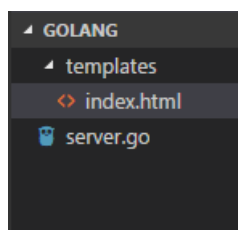
Стоит отметить, что названия переменных следует определять с большой буквы.

Так как в данном случае используются сложные данные, то их надо обернуть в функцию `template.Must()`. Сам код шаблона можно переносить на несколько строк, в этом случае код помещается в косые кавычки. Если код шаблона размещается на одной строке, то можно использовать обычные кавычки.

Результат работы программы:



Однако определение шаблона внутри кода на Go - не лучший вариант, особенно когда шаблон содержит много сложной html-разметки, вкрапления стилей и скриптов javascript. Поэтому более оптимально определять шаблоны в виде отдельных файлов. Например, определим в проекте папку `templates`, а в ней создадим файл `index.html`.



Определим в `index.html` следующий код:

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>{{ .Title }}</title>
  </head>
  <body>
    <h1>{{ .Title }}</h1>
    <p>{{ .Message }}</p>
  </body>
</html>

```

Используем этот шаблон в коде сервера:

```

package main
import (

```

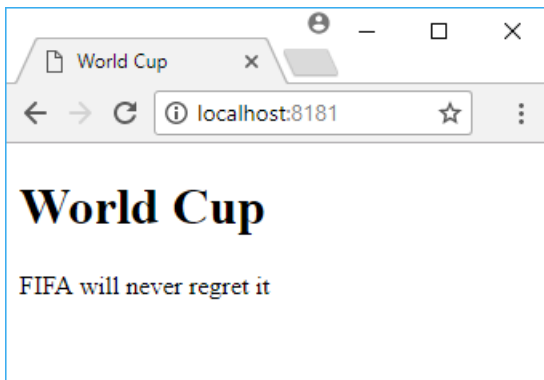
```

    "fmt"
    "net/http"
    "html/template"
)
type ViewData struct{
    Title string
    Message string
}
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        data := ViewData{
            Title: "World Cup",
            Message: "FIFA will never regret it",
        }
        tpl, _ := template.ParseFiles("templates/index.html")
        tpl.Execute(w, data)
    })

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}

```

Для получения кода из файла применяется функция `template.ParseFiles()`, которой передается путь к файлу. Итоговый результат будет почти таким же, как и в предыдущим случае:



Синтаксис шаблонов

Последнее обновление: 02.03.2018

Рассмотрим некоторые базовые элементы синтаксиса шаблонов, например, условные конструкции и циклы.

Циклы

Пусть на стороне сервера в шаблон передается массив:

```

package main
import (
    "fmt"
    "net/http"
    "html/template"
)
type ViewData struct{
    Title string
    Users []string
}
func main() {
    data := ViewData{
        Title : "Users List",
        Users : []string{ "Tom", "Bob", "Sam", },
    }
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        tpl, _ := template.ParseFiles("templates/index.html")
        tpl.Execute(w, data)
    })

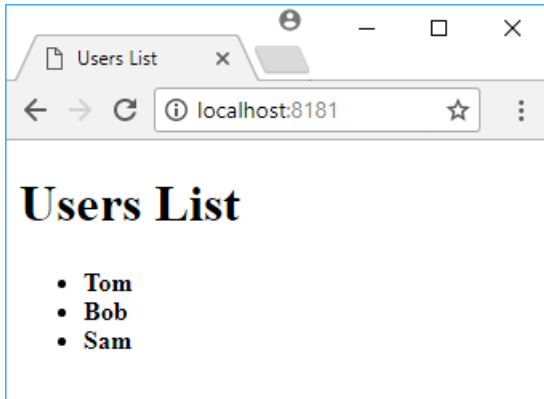
    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}

```

Для вывода массива в шаблоне используется конструкция `{{range массив}} {{end}}`. После слова `range` указывается перебираемый массив:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>{{ .Title }}</title>
  </head>
  <body>
    <h1>{{ .Title }}</h1>
    <ul>
      {{range .Users}}
        <li><b>{{ . }}</b></li>
      {{end}}
    </ul>
  </body>
</html>
```

Внутри конструкции `range` мы можем обращаться к текущему перебираемому объекту с помощью точки `{{ . }}`.



Массив может не иметь данных. Если нам надо определить поведение на этот случай, то можно использовать подконструкцию `{{else}}`:

```
<ul>
  {{range .Users}}
    <li><b>{{ . }}</b></li>
  {{else}}
    <li><b>no rows</b></li>
  {{end}}
</ul>
```

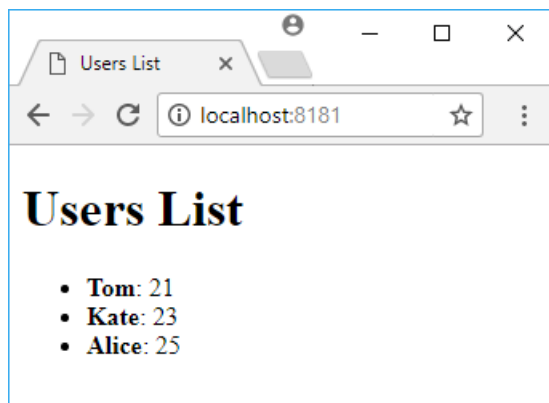
Данные в массиве могут представлять сложные данные:

```
package main
import (
    "fmt"
    "net/http"
    "html/template"
)
type ViewData struct{
    Title string
    Users []User
}
type User struct{
    Name string
    Age int
}
func main() {
    data := ViewData{
        Title : "Users List",
        Users : []User{
            User{Name: "Tom", Age: 21},
            User{Name: "Kate", Age: 23},
            User{Name: "Alice", Age: 25},
        },
    }
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        tmpl, _ := template.ParseFiles("templates/index.html")
        tmpl.Execute(w, data)
    })

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

Вывод этих данных в шаблоне:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>{{ .Title }}</title>
  </head>
  <body>
    <h1>{{ .Title }}</h1>
    <ul>
      {{range .Users}}
        <li>
          <div><b>{{ .Name }}</b>: {{ .Age }}</div>
        </li>
      {{end}}
    </ul>
  </body>
</html>
```



Условные конструкции

Если нам надо вывести в шаблоне некоторую разметку в зависимости от определенного условия, то можно использовать конструкцию `{{if условие}}{{end}}`. После ключевого слова `if` идет условие, которое должно возвращать значение типа `bool`: `true` или `false`.

Например, передадим из сервера в шаблон данные, которые содержат логическое выражение:

```
package main
import (
    "fmt"
    "net/http"
    "html/template"
)
type ViewData struct{
    Available bool
}
func main() {
    data := ViewData{
        Available: true,
    }
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        tmpl, _ := template.ParseFiles("templates/index.html")
        tmpl.Execute(w, data)
    })

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

Определим в шаблоне следующий код:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Available</title>
  </head>
  <body>
    <div>
      {{if .Available}}
        <p>Available</p>
      {{end}}
    </div>
```

```
</body>
</html>
```

То есть если переменная Available равна true, то будет выводиться разметка `<p>Available</p>`. Если переменная равна false, то ничего не будет выводиться.

С помощью конструкции `{{else}}` можно определить разметку html, которая выводится, если условие в if равно false:

```
<div>
    {{if .Available}}
    <p>Available</p>
    {{else}}
    <p>Not Available</p>
    {{end}}
</div>
```

Также мы можем в if сравнивать значения. Например, пусть сервер передает в шаблон текущий час:

```
package main
import (
    "fmt"
    "net/http"
    "html/template"
    "time"
)
type ViewData struct{
    Hour int
}
func main() {
    data := ViewData{
        Hour: time.Now().Hour(),
    }
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        tmpl, _ := template.ParseFiles("templates/index.html")
        tmpl.Execute(w, data)
    })

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

С помощью метода `time.Now().Hour()` здесь получаем текущий час.

В шаблоне определим следующую конструкцию:

```
<div>
    {{if lt .Hour 12 }}
    <p>Доброе утро</p>
    {{else}}
    <p>Добрый день</p>
    {{end}}
</div>
```

В данном случае сравнивается значение текущего часа с числом 12, и в зависимости от значения выводим тот или иной текст.

Оператор `lt` можно расшифровать как "less than", то есть меньше чем. То есть фактически это аналог операции `<`. Он сравнивает два значения и возвращает true, если первое значение меньше второго. Иначе возвращается значение false.

Подобным образом мы можем использовать еще ряд операторов, которые аналогичны стандартным операторам сравнения:

- `eq` : возвращает true, если два значения равны
- `ne` : возвращает true, если два значения НЕ равны
- `le` : возвращает true, если первое значение меньше или равно второму
- `gt` : возвращает true, если первое значение больше второго
- `ge` : возвращает true, если первое значение больше или равно второму

Кроме того, есть ряд операторов, которые аналогичны логическим операторам:

- `and` : возвращает true, если два выражения равны true
- `or` : возвращает true, если хотя бы одно из двух выражений равно true
- `not` : возвращает true, если выражение возвращает false

Применение некоторых операторов:


```
<div>
    {{if or (gt 2 1) (lt 5 7)}}
    <p>Первый вариант</p>
    {{else}}
    <p>Второй вариант</p>
    {{end}}
</div>
```

Работа с базой данных

Подключение к БД и получение данных

Последнее обновление: 03.03.2018

Рассмотрим, как мы можем взаимодействовать с базой данных в веб-приложении. Основные моменты работы с бд с помощью языка программирования Go были рассмотрены в материале [Go и базы данных](#). В данном же случае мы рассмотрим только непосредственно применение этих моментов в рамках веб-приложения.

В качестве системы управления базами данных возьмем MySQL. Вначале создадим на сервере MySQL базу данных productdb и в ней таблицу products. Для этого можно использовать следующие выражений SQL

```
create database productdb;
use productdb;
create table products (
    id int auto_increment primary key,
    model varchar(30) not null,
    company varchar(30) not null,
    price int not null
)
```

То есть база данных productdb, в ней есть таблица products, которая будет хранить информацию о товарах, будет 4 столбца: id - идентификатор каждой записи, model - название товара, company - производитель товара и price - цена товара.

Добавим в нее какие-нибудь начальные данные, например, с помощью следующего скрипта:

```
insert into productdb.Products (model, company, price)
values ('iPhone X', 'Apple', 74000),
('Pixel 2', 'Google', 62000),
('Galaxy S9', 'Samsung', 65000)
```

Прежде чем начать работать с MySQL, надо добавить драйвер для Go к переменной \$GOPATH (если он ранее не был добавлен). Для этого нужно выполнить в командной строке/терминале следующую команду:

```
go get github.com/go-sql-driver/mysql
```

После этого определим на сервере следующий код:

```
package main
import (
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
    "net/http"
    "html/template"
    "log"
)
type Product struct{
    Id int
    Model string
    Company string
    Price int
}
var database *sql.DB

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

    rows, err := database.Query("select * from productdb.Products")
    if err != nil {
        log.Println(err)
    }
    defer rows.Close()
    products := []Product{}

    for rows.Next(){
        p := Product{}
        err := rows.Scan(&p.Id, &p.Model, &p.Company, &p.Price)
        if err != nil{
            fmt.Println(err)
            continue
        }
    }
}
```

```

        products = append(products, p)
    }

    tmpl, _ := template.ParseFiles("templates/index.html")
    tmpl.Execute(w, products)
}

func main() {
    db, err := sql.Open("mysql", "root:password@/productdb")

    if err != nil {
        log.Println(err)
    }
    database = db
    defer db.Close()
    http.HandleFunc("/", IndexHandler)

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}

```

Прежде всего здесь определена структура Product, которая соответствует определению таблицы products в базе данных. А за взаимодействие с базой данных отвечает переменная database.

Для отправки пользователю списка объектов из БД определена функция IndexHandler. В ней с помощью метода database.Query выполняется запрос "select * from productdb.Products", который извлекает все объекты из таблицы. Затем из полученного набора создается массив структур Product, который затем передается в шаблон index.html (код шаблона приведен ниже).

В функции main открываем подключение с базой данных с помощью функции sql.Open :

```
db, err := sql.Open("mysql", "root:password@/productdb")
```

Этой функции в качестве первого параметра передается название драйвера - "mysql". Вторым параметром является настройки подключения, где root - название пользователя в MySQL, password - пароль этого пользователя (как правило тот, который устанавливается при установке MySQL), и productdb - название базы данных. Соответственно в каждом конкретном случае пароль может отличаться.

После открытия подключения устанавливается значение переменной database.

```
database = db
```

И далее функция IndexHandler устанавливается в качестве обработчика запросов по корневому адресу:

```
http.HandleFunc("/", IndexHandler)
```

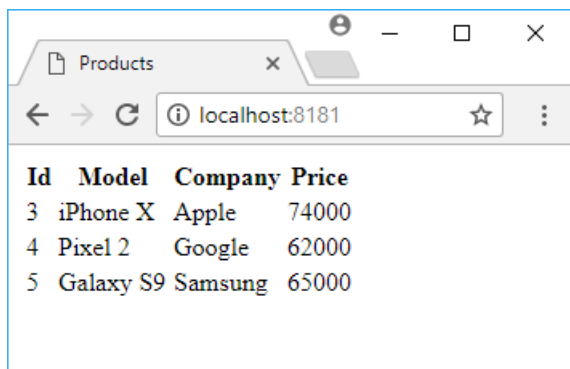
Теперь определим в проекте папку templates , а в ней создадим новый файл index.html , который будет представлять шаблон для вывода массива объектов и будет иметь следующий код:

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Products</title>
  </head>
  <body>
    <table>
      <thead><th>Id</th><th>Model</th><th>Company</th><th>Price</th></thead>
      {{range . }}
      <tr>
        <td>{{.Id}}</td>
        <td>{{.Model}}</td>
        <td>{{.Company}}</td>
        <td>{{.Price}}</td>
      </tr>
      {{end}}
    </table>
  </body>
</html>

```

В итоге после запуска проекта и обращения к корню сайта будет открыто подключение к базе данных, приложение получит все необходимые данные из бд и передаст их в шаблон:



Id	Model	Company	Price
3	iPhone X	Apple	74000
4	Pixel 2	Google	62000
5	Galaxy S9	Samsung	65000

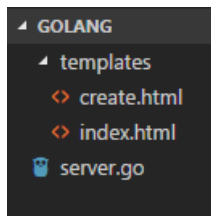
Добавление данных

Последнее обновление: 03.03.2018

Процесс добавления данных разбивается на две части. Вначале нам надо показать пользователю форму для ввода данных. Затем, когда пользователь отправит введенные данные, нам надо их получить и добавить в базу данных.

Возьмем проект из прошлой темы. И прежде всего добавим в папку templates новый файл create.html, который будет содержать форму для добавления:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>New Product</title>
  </head>
  <body>
    <h3>Add Product</h3>
    <form method="POST">
      <label>Model</label><br>
      <input type="text" name="model" /><br><br>
      <label>Company</label><br>
      <input type="text" name="company" /><br><br>
      <label>Price</label><br>
      <input type="number" name="price" /><br><br>
      <input type="submit" value="Send" />
    </form>
  </body>
</html>
```



Изменим файл сервера следующим образом:

```
package main
import (
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
    "net/http"
    "html/template"
    "log"
)
type Product struct{
    Id int
    Model string
    Company string
    Price int
}
var database *sql.DB

// функция добавления данных
func CreateHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method == "POST" {

        err := r.ParseForm()
        if err != nil {
            log.Println(err)
        }
        model := r.FormValue("model")
```

```

    company := r.FormValue("company")
    price := r.FormValue("price")

    _, err = database.Exec("insert into productdb.Products (model, company, price) values (?, ?, ?)",
        model, company, price)

    if err != nil {
        log.Println(err)
    }
    http.Redirect(w, r, "/", 301)
} else {
    http.ServeFile(w, r, "templates/create.html")
}
}

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

    rows, err := database.Query("select * from productdb.Products")
    if err != nil {
        log.Println(err)
    }
    defer rows.Close()
    products := []Product{}

    for rows.Next(){
        p := Product{}
        err := rows.Scan(&p.Id, &p.Model, &p.Company, &p.Price)
        if err != nil{
            fmt.Println(err)
            continue
        }
        products = append(products, p)
    }

    tpl, _ := template.ParseFiles("templates/index.html")
    tpl.Execute(w, products)
}

func main() {

    db, err := sql.Open("mysql", "root:password@/productdb")

    if err != nil {
        log.Println(err)
    }
    database = db
    defer db.Close()
    http.HandleFunc("/", IndexHandler)
    http.HandleFunc("/create", CreateHandler)

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}

```

Для добавления данных определена функция `CreateHandler()`. Поскольку нам надо, с одной стороны, отображать пользователю форму для добавления, а, с другой стороны, получать и добавлять данные в БД, то данная функция условно разделена на две части. В ней мы проверяем тип запроса. Если запрос представляет тип "GET", то мы будем возвращать форму для добавления. Если запрос имеет тип "POST", то парсим данные полученной формы и извлекаем из них нужные элементы.

Для извлечения нужных данных из полученных форм применяется метод `r.FormValue()`. В качестве параметра этому методу передается название данных. То есть, например, на форме есть следующее поле:

```
<input type="text" name="model" />
```

Атрибут `name` указывает, что название этого поля - "model". Следовательно, чтобы получить введенные в него данные, необходимо использовать выражение `model := r.FormValue("model")`.

После получения всех данных они добавляются в БД с помощью метода `database.Exec`:

```
database.Exec("insert into productdb.Products (model, company, price) values (?, ?, ?)", model, company, price)
```

После этого выполняется переадресация на корень сайта с помощью функции `http.Redirect`:

```
http.Redirect(w, r, "/", 301)
```

Третий параметр указывает путь переадресации. В данном случае "/", по которому выводится список объектов из БД. Четвертый параметр представляет статусный код переадресации. В данном случае код 301 указывает, что переадресация временная.

Если же запрос к серверу представляет тип GET, то просто возвращаем пользователю веб-страницу `create.html`:

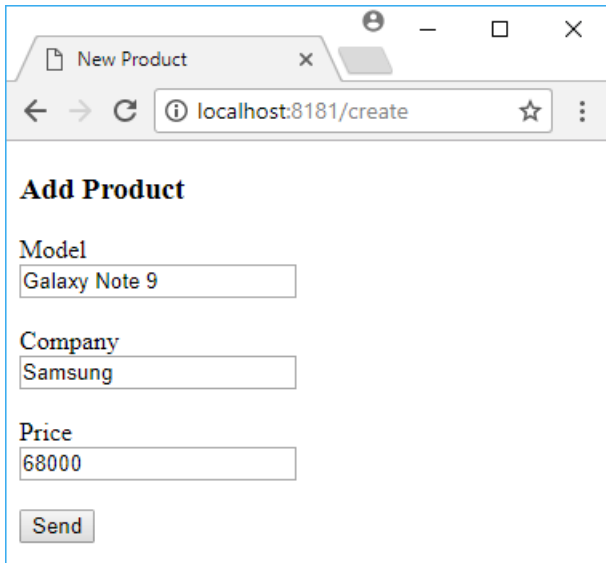
```
http.ServeFile(w, r, "templates/create.html")
```

Функция `IndexHandler`, которая возвращает список объектов, остается той же, что и в прошлой теме.

Ну и в функции main CreateHandler устанавливается в качестве обработчика по пути "/create":

```
http.HandleFunc("/create", CreateHandler)
```

Запустим приложение. Перейдем по пути "http://localhost:8181/create", и нам отобразится форма. Введем в нее какие-нибудь данные:



Add Product

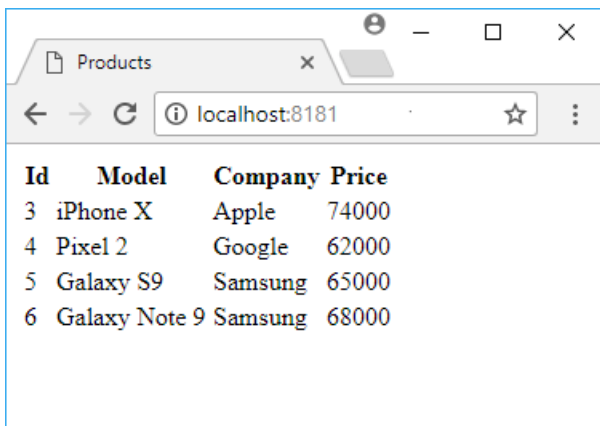
Model
Galaxy Note 9

Company
Samsung

Price
68000

Send

И после отправки формы приложение получит данные, добавит их в БД и переадресует на главную страницу:



Id	Model	Company	Price
3	iPhone X	Apple	74000
4	Pixel 2	Google	62000
5	Galaxy S9	Samsung	65000
6	Galaxy Note 9	Samsung	68000

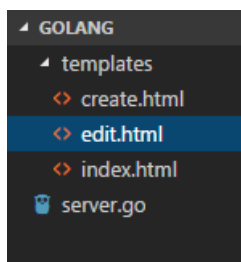
Редактирование данных

Последнее обновление: 03.03.2018

Продолжим работу с проектом из прошлой темы и добавим в него возможность редактирования данных.

Редактирование данных, как и добавление, разбивается на две части. Вначале нам надо отобразить пользователю форму для изменения выбранного объекта. Потом нам надо получить отправленные данные и сохранить их в базу данных.

Прежде всего определим форму для редактирования. Для этого в папке templates создадим файл edit.html .



Определим в файле edit.html следующий код:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Edt Product</title>
  </head>
```

```

<body>
  <h3>Edit Product</h3>
  <form method="POST">
    <input type="hidden" name="id" value="{{.Id}}" />
    <label>Model</label><br>
    <input type="text" name="model" value="{{.Model}}" /><br><br>
    <label>Company</label><br>
    <input type="text" name="company" value="{{.Company}}" /><br><br>
    <label>Price</label><br>
    <input type="number" name="price" value="{{.Price}}" /><br><br>
    <input type="submit" value="Send" />
  </form>
</body>
</html>

```

Данный файл представляет шаблон, в который из кода сервера будут передаваться редактируемые данные.

Теперь изменим код сервера, добавив в него возможность редактирования:

```

package main
import (
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
    "net/http"
    "html/template"
    "log"
    "github.com/gorilla/mux"
)
type Product struct{
    Id int
    Model string
    Company string
    Price int
}
var database *sql.DB

// возвращаем пользователю страницу для редактирования объекта
func EditPage(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id := vars["id"]

    row := database.QueryRow("select * from productdb.Products where id = ?", id)
    prod := Product{}
    err := row.Scan(&prod.Id, &prod.Model, &prod.Company, &prod.Price)
    if err != nil{
        log.Println(err)
        http.Error(w, http.StatusText(404), http.StatusNotFound)
    }else{
        tpl, _ := template.ParseFiles("templates/edit.html")
        tpl.Execute(w, prod)
    }
}

// получаем измененные данные и сохраняем их в БД
func EditHandler(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        log.Println(err)
    }
    id := r.FormValue("id")
    model := r.FormValue("model")
    company := r.FormValue("company")
    price := r.FormValue("price")

    _, err = database.Exec("update productdb.Products set model=?, company=?, price = ? where id = ?",
        model, company, price, id)

    if err != nil {
        log.Println(err)
    }
    http.Redirect(w, r, "/", 301)
}

func CreateHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method == "POST" {
        err := r.ParseForm()
        if err != nil {
            log.Println(err)
        }
        model := r.FormValue("model")
        company := r.FormValue("company")
        price := r.FormValue("price")

        _, err = database.Exec("insert into productdb.Products (model, company, price) values (?, ?, ?)",
            model, company, price)
    }
}

```

```

        if err != nil {
            log.Println(err)
        }
        http.Redirect(w, r, "/", 301)
    }else{
        http.ServeFile(w, r, "templates/create.html")
    }
}

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

    rows, err := database.Query("select * from productdb.Products")
    if err != nil {
        log.Println(err)
    }
    defer rows.Close()
    products := []Product{}

    for rows.Next(){
        p := Product{}
        err := rows.Scan(&p.Id, &p.Model, &p.Company, &p.Price)
        if err != nil{
            fmt.Println(err)
            continue
        }
        products = append(products, p)
    }

    tmpl, _ := template.ParseFiles("templates/index.html")
    tmpl.Execute(w, products)
}

func main() {

    db, err := sql.Open("mysql", "root:password@/productdb")

    if err != nil {
        log.Println(err)
    }
    database = db
    defer db.Close()

    router := mux.NewRouter()
    router.HandleFunc("/", IndexHandler)
    router.HandleFunc("/create", CreateHandler)
    router.HandleFunc("/edit/{id:[0-9]+}", EditPage).Methods("GET")
    router.HandleFunc("/edit/{id:[0-9]+}", EditHandler).Methods("POST")

    http.Handle("/", router)

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}

```

По сравнению с прошлой темой здесь добавлены функции EditPage и EditHandler и изменена функция main.

Чтобы указать, какой объект будет редактироваться, мы будем передавать через адрес id этого объекта. И для упрощения маршрутизации в данном случае мы будем использовать пакет gorilla/mux .

В функции IndexPage мы получаем id объекта, который надо изменить, извлекаем из БД данные этого объекта и передаем их в шаблон edit.html:

```

func EditPage(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id := vars["id"]

    row := database.QueryRow("select * from productdb.Products where id = ?", id)
    prod := Product{}
    err := row.Scan(&prod.Id, &prod.Model, &prod.Company, &prod.Price)
    if err != nil{
        log.Println(err)
        http.Error(w, http.StatusText(404), http.StatusNotFound)
    }else{
        tmpl, _ := template.ParseFiles("templates/edit.html")
        tmpl.Execute(w, prod)
    }
}

```

На случай, если в базе данных не окажется объекта с подобным id, с помощью функции http.Error() возвращаем статусный код 404, который указывает, что объект не найден.

В функции EditHandler получаем данные из отправленной формы и с их помощью изменяем объект в базе данных по определенному id.

```
func EditHandler(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        log.Println(err)
    }
    id := r.FormValue("id")
    model := r.FormValue("model")
    company := r.FormValue("company")
    price := r.FormValue("price")

    _, err = database.Exec("update productdb.Products set model=?, company=?, price = ? where id = ?",
        model, company, price, id)

    if err != nil {
        log.Println(err)
    }
    http.Redirect(w, r, "/", 301)
}
```

После обновления БД выполняется редирект на главную страницу.

В функции main эти функции EditPage и EditHandler связываются с определенными маршрутами. По сути они привязаны к одному и тому же маршруту, однако для разного типа запросов: EditPage для запросов GET, а EditHandler - для запросов POST.

```
router.HandleFunc("/edit/{id:[0-9]+}", EditPage).Methods("GET")
router.HandleFunc("/edit/{id:[0-9]+}", EditHandler).Methods("POST")
```

Стоит отметить, что добавление данных, которое представлено в данном случае функцией CreateHandler, также фактически выполняет два действия в зависимости от типа запроса: отображает страницу для добавления и собственно добавляет данные. И в принципе организацию добавления можно сделать также, как и редактирование, разделив на две функции для каждого типа запросов.

Для упрощения управлением объектами изменим файл index.html, добавив в него ссылки на редактирование:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Products</title>
  </head>
  <body>
    <h2>Список товаров</h2>
    <p><a href="/create">Добавить</a></p>
    <table>
      <thead><th>Id</th><th>Model</th><th>Company</th><th>Price</th><th></th></thead>
      {{range . }}
      <tr>
        <td>{{.Id}}</td>
        <td>{{.Model}}</td>
        <td>{{.Company}}</td>
        <td>{{.Price}}</td>
        <td><a href="/edit/{{.Id}}">Изменить</a>
      </tr>
      {{end}}
    </table>
  </body>
</html>
```

И после запуска приложения мы сможем отредактировать нужные объекты:

The screenshot shows two browser windows side-by-side. The left window, titled 'Products', displays a list of products in a table. The right window, titled 'Edt Product', shows a form for editing a product. A red arrow points from the 'Изменить' (Edit) link in the table to the 'Edit Product' form.

Id	Model	Company	Price	
3	iPhone X	Apple	74000	Изменить
4	Pixel 2	Google	62000	Изменить
5	Galaxy S9	Samsung	65000	Изменить
6	Galaxy Note 9	Samsung	68000	Изменить
7	Galaxy S 8	Samsung	42000	Изменить

The 'Edit Product' form contains the following fields:

- Model: Galaxy S 8
- Company: Samsung
- Price: 42000
- Send button

Удаление данных

Последнее обновление: 03.03.2018

Продолжим работу с проектом из прошлой темы и добавим в него удаление объектов. Удаление можно сделать различными способами. В данном же случае мы оставимся на самом простом варианте, когда в GET-запросе передается id объекта, и в базе данных происходит удаление по этому id.

Итак, изменим код сервера следующим образом:

```
package main
import (
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
    "net/http"
    "html/template"
    "log"
    "github.com/gorilla/mux"
)
type Product struct{
    Id int
    Model string
    Company string
    Price int
}
var database *sql.DB

func DeleteHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id := vars["id"]

    _, err := database.Exec("delete from productdb.Products where id = ?", id)
    if err != nil{
        log.Println(err)
    }

    http.Redirect(w, r, "/", 301)
}

func EditPage(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id := vars["id"]

    row := database.QueryRow("select * from productdb.Products where id = ?", id)
    prod := Product{}
    err := row.Scan(&prod.Id, &prod.Model, &prod.Company, &prod.Price)
    if err != nil{
        log.Println(err)
        http.Error(w, http.StatusText(404), http.StatusNotFound)
    }else{
        tpl, _ := template.ParseFiles("templates/edit.html")
        tpl.Execute(w, prod)
    }
}

func EditHandler(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        log.Println(err)
    }
    id := r.FormValue("id")
    model := r.FormValue("model")
    company := r.FormValue("company")
    price := r.FormValue("price")

    _, err = database.Exec("update productdb.Products set model=?, company=?, price = ? where id = ?",
        model, company, price, id)

    if err != nil {
        log.Println(err)
    }
    http.Redirect(w, r, "/", 301)
}

func CreateHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method == "POST" {

        err := r.ParseForm()
        if err != nil {
            log.Println(err)
        }
        model := r.FormValue("model")
        company := r.FormValue("company")
        price := r.FormValue("price")
    }
}
```

```

_, err = database.Exec("insert into productdb.Products (model, company, price) values (?, ?, ?)",
    model, company, price)

if err != nil {
    log.Println(err)
}
http.Redirect(w, r, "/", 301)
}else{
    http.ServeFile(w, r, "templates/create.html")
}
}

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

    rows, err := database.Query("select * from productdb.Products")
    if err != nil {
        log.Println(err)
    }
    defer rows.Close()
    products := []Product{}

    for rows.Next(){
        p := Product{}
        err := rows.Scan(&p.Id, &p.Model, &p.Company, &p.Price)
        if err != nil{
            fmt.Println(err)
            continue
        }
        products = append(products, p)
    }

    tpl, _ := template.ParseFiles("templates/index.html")
    tpl.Execute(w, products)
}

func main() {

    db, err := sql.Open("mysql", "root:password@/productdb")

    if err != nil {
        log.Println(err)
    }
    database = db
    defer db.Close()

    router := mux.NewRouter()
    router.HandleFunc("/", IndexHandler)
    router.HandleFunc("/create", CreateHandler)
    router.HandleFunc("/edit/{id:[0-9]+}", EditPage).Methods("GET")
    router.HandleFunc("/edit/{id:[0-9]+}", EditHandler).Methods("POST")
    router.HandleFunc("/delete/{id:[0-9]+}", DeleteHandler)

    http.Handle("/", router)

    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}

```

По сравнению с прошлой темой здесь добавлена функция DeleteHandler, которая получает id удаляемого объекта и выполняет DELETE-запрос к базе данных. И после удаления происходит переадресация на главную страницу.

Для упрощения удаления определим в файле index.html ссылку на удаление рядом с каждым объектом:

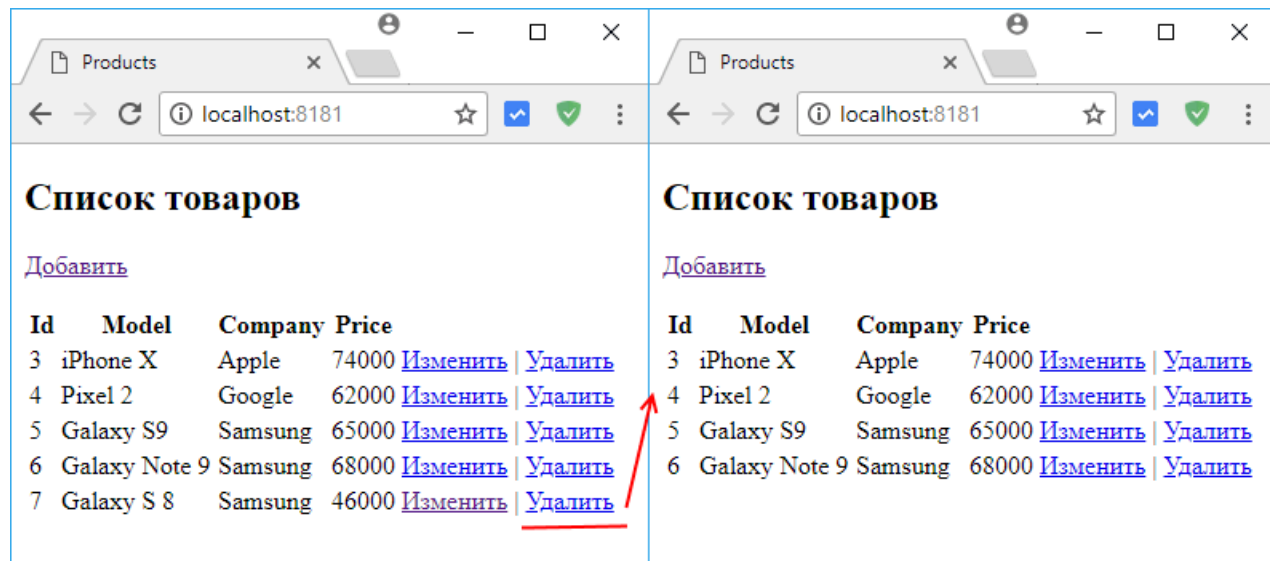
```

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Products</title>
</head>
<body>
<h2>Список товаров</h2>
<p><a href="/create">Добавить</a></p>
<table>
<thead><th>Id</th><th>Model</th><th>Company</th><th>Price</th><th></th></thead>
{{range . }}
<tr>
<td>{{.Id}}</td>
<td>{{.Model}}</td>
<td>{{.Company}}</td>
<td>{{.Price}}</td>
<td><a href="/edit/{{.Id}}">Изменить</a> |
<a href="/delete/{{.Id}}">Удалить</a>
</td>
</tr>
{{end}}

```

```
</table>
</body>
</html>
```

Протестируем удаление, нажав на соответствующую ссылку рядом с каким-нибудь объектом:



Скриншоты веб-интерфейса, демонстрирующие список товаров и возможность удаления объектов.

Оба скриншота отображают страницу с заголовком "Список товаров" и ссылкой "Добавить".

Id	Model	Company	Price	Измeнить	Удалить
3	iPhone X	Apple	74000	Измeнить	Удалить
4	Pixel 2	Google	62000	Измeнить	Удалить
5	Galaxy S9	Samsung	65000	Измeнить	Удалить
6	Galaxy Note 9	Samsung	68000	Измeнить	Удалить
7	Galaxy S 8	Samsung	46000	Измeнить	Удалить

В левом скриншоте ссылка "Удалить" для товара с Id 7 выделена красной линией. В правом скриншоте красная стрелка указывает на ссылку "Удалить" для товара с Id 4.