

# Golang

## Обработка форм



2020



# Информация

Данная публикация стала возможной благодаря помощи американского народа, оказанной через Агентство США по международному развитию (USAID). Алиф Академия несёт ответственность за содержание публикации, которое не обязательно отражает позицию USAID или Правительства США.



# ПРЕДИСЛОВИЕ



# Предисловие

На прошлых лекциях мы познакомились со стандартной библиотекой, а именно пакетом `net/http`, в котором реализована поддержка как создания клиентов, так и серверов на Go.

Мы построили API, чем-то похожее на Vk, при этом передавали всю информацию в Query. Сегодня мы разберёмся, какие ещё возможности по передаче данных у нас есть и как они работают.



# ФОРМЫ



# Формы

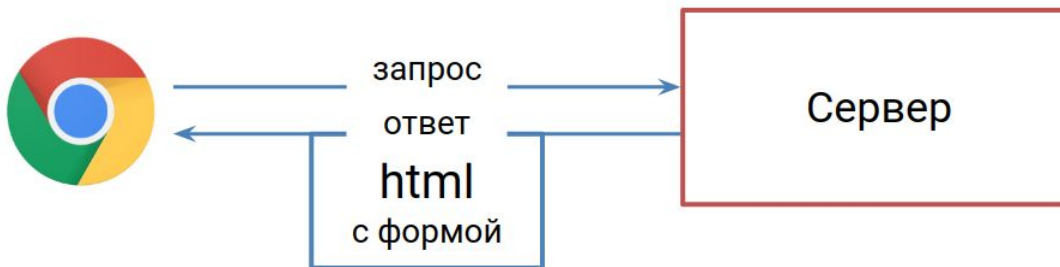
Изначально всё началось с того, какие способы передачи данных предоставлял браузер с помощью HTML.



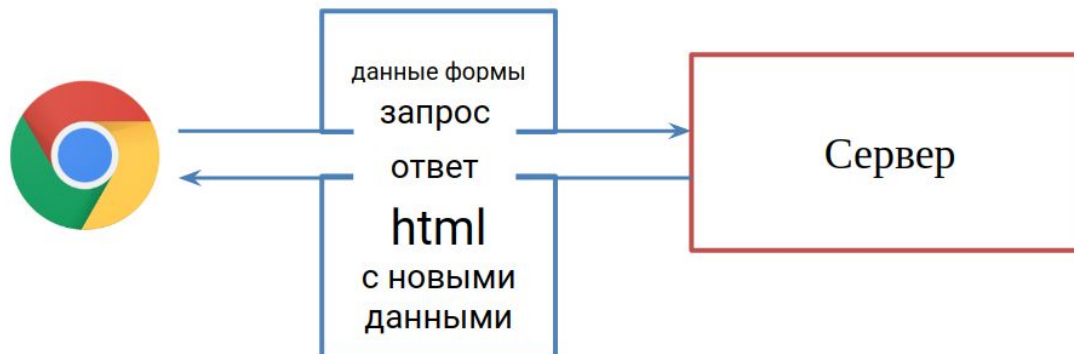
# Формы

Изначально, когда JS мало использовали (у него было достаточно много проблем при работе в разных браузерах), общая схема выглядела следующим образом:

Шаг 1. Браузер отправляет запрос на сервер:



Шаг 2. Пользователь заполняет форму и отправляет на сервер (в ответ приходит новый документ):



# Форма

Такая схема до сих пор широко распространена, поэтому мы рассмотрим вопрос обработки подобных данных.

Чтобы данные отправлялись на сервер, одним из условий является наличие элемента `form` – HTML-форма. Этот элемент (по умолчанию) никак визуально не отображается на странице и служит для объединения полей ввода в одну логическую единицу – форму.





# HTML

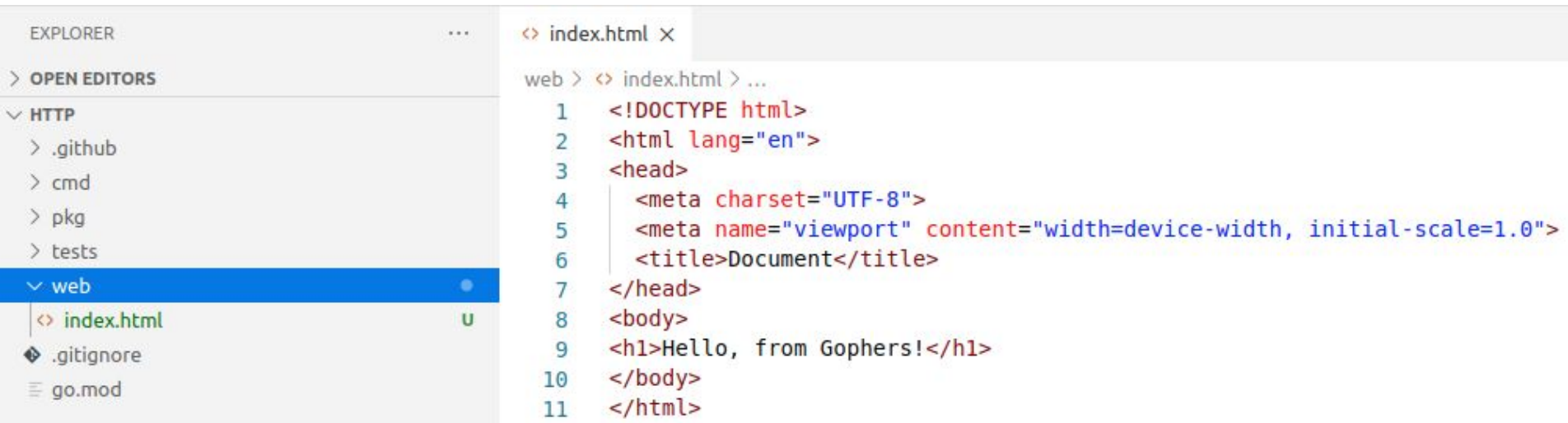
**HyperText Markup Language (HTML)** – язык разметки, определяющий структуру документа. На HTML существует [спецификация](#), которая описывает правила создания подобных документов.

Конечно же, нас не будет интересовать целиком вся спецификация (хотя нам, как backend разработчикам, неплохо бы знать, что умеет современный Web и HTML в частности, поскольку это будет определять то, какой backend мы можем разрабатывать, какие технологии использовать и т.д.).



# HTML Document

Базовая структура HTML-документа выглядит следующим образом (пока этот код набирать не нужно):



The screenshot shows a code editor interface. On the left is the 'EXPLORER' sidebar with a tree view containing folders like '.github', 'cmd', 'pkg', 'tests', and 'web'. The 'web' folder is selected, and inside it, the file 'index.html' is highlighted. On the right, the 'index.html' file is open in the editor, showing the following HTML code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Document</title>
7 </head>
8 <body>
9   <h1>Hello, from Gophers!</h1>
10 </body>
11 </html>
```

Согласно [Standard Go Project Layout](#) мы поместили файл в каталог web. Файл мы называли index.html.

Давайте разберёмся с основными правилами.



# HTML Element

HTML документ представляет из себя дерево из HTML Element'ов. Каждый HTML Element представляет из себя специальную запись:



Пишется всё именно в таком порядке: сначала открывающий, затем содержимое (в данном случае – текст) и только потом закрывающий.

Закрывающий тег от открывающего отличается наличием символа /.



# HTML Element

Все допустимые элементы (те, которые понимает браузер) описаны в спецификации. Элементы – это те "кирпичики", из которых строится страница.

Например, есть элементы для отображения картинок, аудио, видео. Элемент **h1** предназначен для отображения заголовка первого уровня (самого главного заголовка на странице).



# Вложенность

Внутри элементов можно помещать текст и другие элементы:

```
<body>  
<h1>Hello, from Gophers!</h1>  
</body>
```

Здесь внутри элемента **body** расположен элемент **h1**. Это позволяет организовывать дерево элементов: тот элемент, в который вкладывают другие, называется родительским элементом (**body** родительский элемент для **h1**), а те, которые вкладывают, называют дочерними элементами (**h1** дочерний элемент для **body**).

**Важно правило:** у каждого элемента есть только один родительский элемент.



# Вложенность

Пример правильной вложенности:

```
<body>  
<h1>Hello, from <span>Gophers</span>!</h1>  
</body>
```

Элемент не выходит за "границы" родительского

Примеры неправильной вложенности (внимательно следите за вложенностью!):

```
<body>  
<h1>Hello, from <span>Gophers!</h1></span>  
</body>
```

Элемент **выходит** за "границы" родительского

```
<body>  
<h1>Hello, from <span>Gophers!</h1>  
</body>
```

Отсутствует закрывающий тег



# Закрывающий тег

Для некоторых элементов работает следующее правило: если у элемента нет содержимого (текста внутри или вложенных элементов), то можно использовать сокращённую запись:

```
<meta charset="utf-8" />
```

Либо вообще опускать:

```
<meta charset="utf-8">
```

Мы разберём чуть позже для каких элементов это правило работает.



# Атрибуты

Элемент может содержать атрибуты (располагаются только в открывающем теге):

```
<meta charset="utf-8">
```

имя атрибута

значение



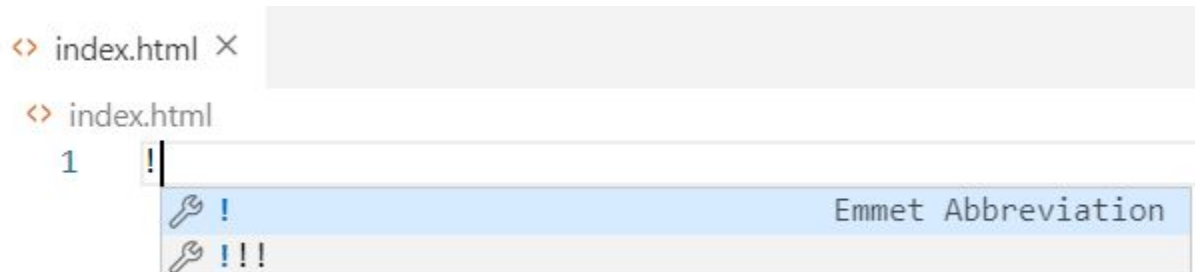


# Emmet

Мы с вами разобрали ключевые моменты приведённого фрагмента HTML-кода.

Самый главный вопрос: "Как эту базовую структуру создать"? Ведь запомнить её с ходу достаточно сложно.

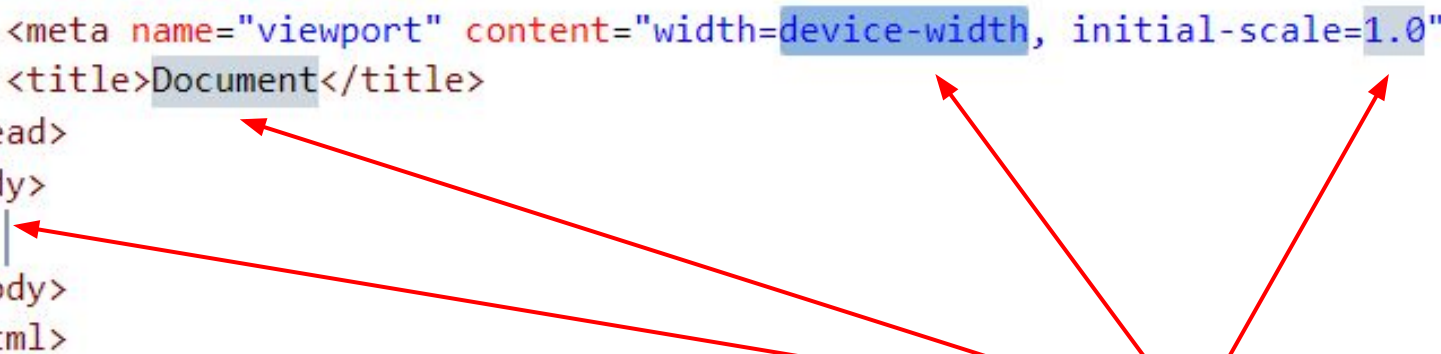
На самом деле, запоминать её и не нужно, потому что вам будет помогать специальный инструмент - Emmet (который уже встроен в VS Code). Наберите в редакторе символ **!** и подождите, пока появится подсказка, после чего нажмите клавишу Tab:



# Emmet

Emmet за вас напишет типовую структуру:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  |
</body>
</html>
```



Нажимая на клавишу Tab вы можете перемещаться между подсвеченными областями, пока не попадёте внутрь элемента **body**.



# Emmet: элементы

Помимо базовой структуры, мы можем с помощью Emmet создавать и элементы.

Для этого нужно написать тег элемента и нажать на Tab:

```
<body>  
  h1  
</body>  h1 Emmet Abbreviation  
</html>
```

Нажатие на Tab создаёт открывающий и закрывающий тег + помещает курсор между ними

```
<body>  
  <h1></h1>  
</body>  
</html>
```

Теперь внутри элемента мы можем написать Hello, from Gophers!



# Emmet

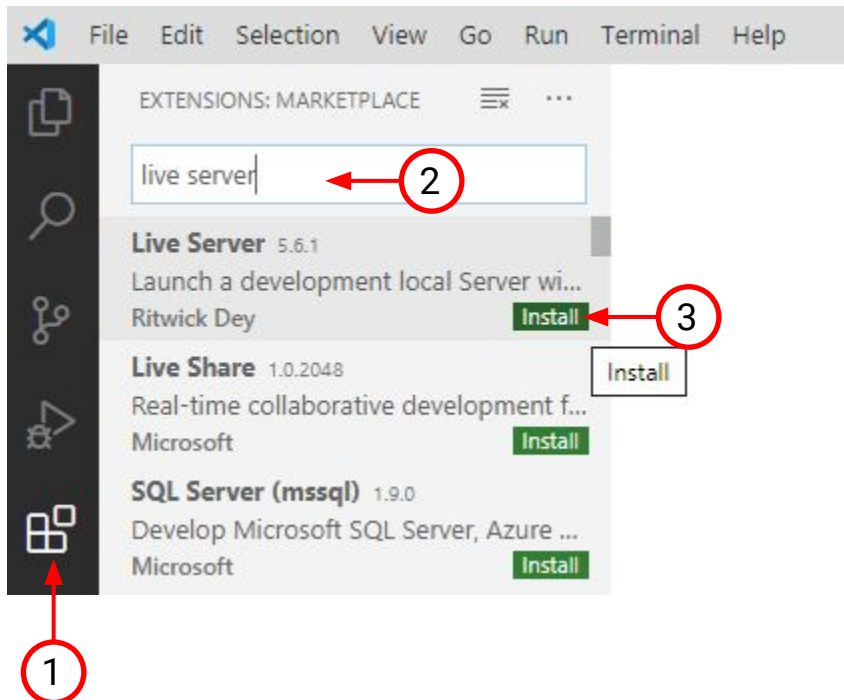
Emmet имеет ещё кучу других возможностей, с которыми мы познакомимся в рамках курса.

Например, он знает, для каких элементов закрывающий тег не обязателен и не будет его писать (а вам не придётся это запоминать, если вы пользуетесь Emmet).



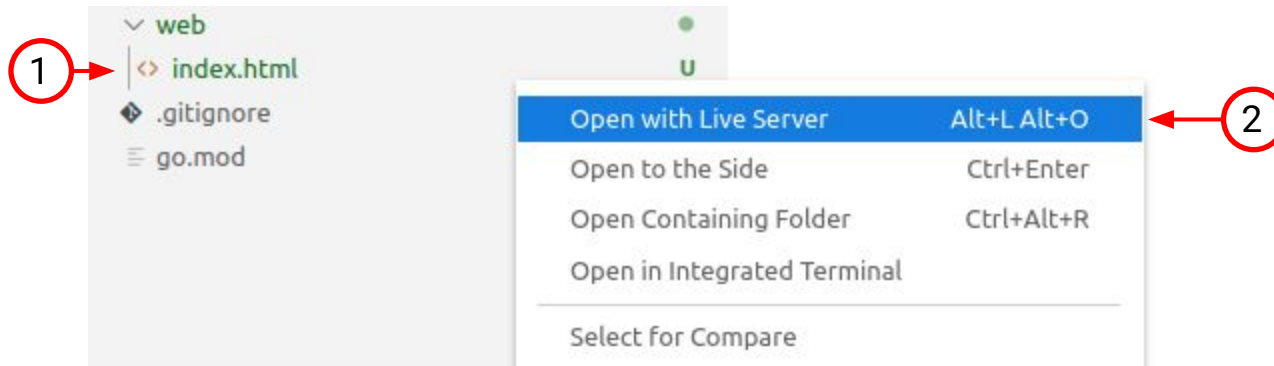
# VS Code Live Server

Для удобного просмотра документа нам потребуется соответствующее расширение VS Code: зайдите в панельку Extensions (1), наберите Live Server (2) и нажмите на кнопку Install (3):



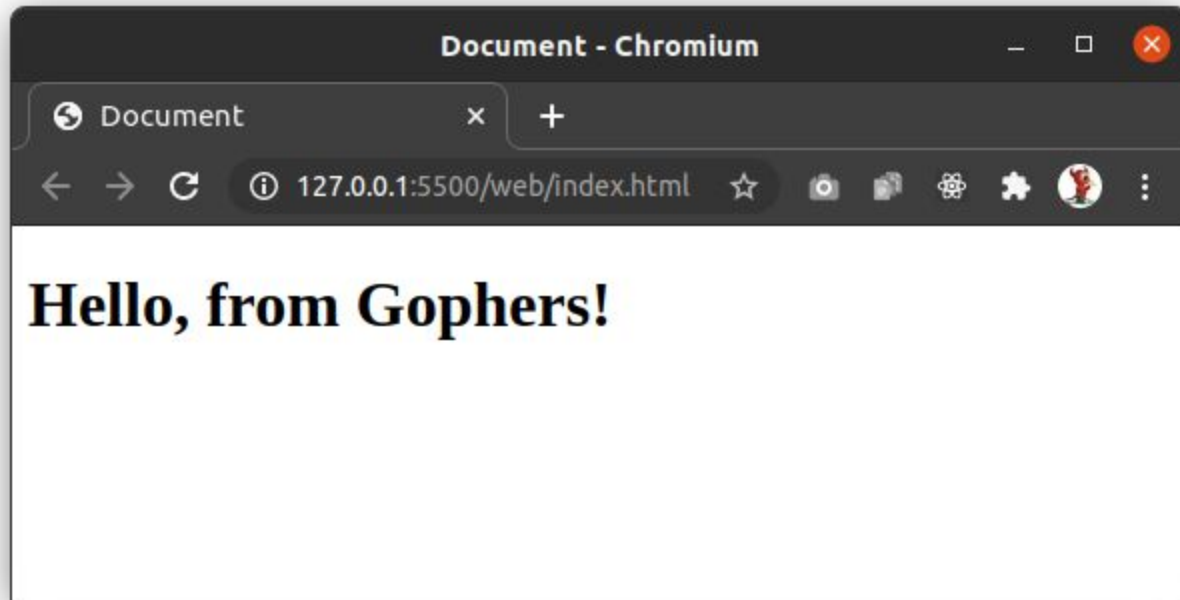
# Live Server

Щёлкните правой кнопкой на index.html (1) и выберите "Open with Live Server" (2):



# Live Server

В установленном у вас браузере откроется следующая страница:



# HTML

В HTML документе есть две большие "части": **head** и **body**.

**head** отвечает за мета-данные страницы (т.е. специальные служебные данные для браузера) – например, название вкладки или масштаб.

**body** же отвечает за те данные, которые будут отображены на самой странице, именно поэтому мы видим содержимое **h1**.





# Формы

А теперь давайте разбираться с тем, как делать формы и отправлять с помощью них данные на наш сервер (и самое главное, как их там принимать).

Согласно спецификации, форма задаётся элементом `form`, внутри которого располагаются поля ввода. У самой формы есть ряд ключевых атрибутов:

- `action` - URI, на который будет отправлена форма (если не указано, то тот, с которого страница загружена: <http://127.0.0.1:5500/web/index.html> у нас)
- `method` - HTTP-метод для отправки данных (допустимые значения - GET или POST, по умолчанию используется GET)
- `enctype` - как кодируются данные формы



# Формы

Начнём с простой формы:

<> index.html ×

web > <> index.html > ...

```
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5      <meta charset="UTF-8">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Document</title>
8  </head>
9
10 <body>
11     <form action="http://localhost:9999/process">
12         <input type="number" name="count">
13         <input type="text">
14         <input type="text" name="tags">
15         <input type="text" name="tags">
16         <input type="file" name="image">
17         <button>Отправить</button>
18     </form>
19 </body>
20
21 </html>
```



# Сервер

Сервер будет супер-простым, предназначенным только для демонстрации того, как получать значения формы (см. следующий слайд).



```

func main() {
    host := "0.0.0.0"
    port := "9999"

    if err := execute(host, port); err != nil {
        os.Exit(1)
    }
}

func execute(host string, port string) (err error) {
    srv := &http.Server{
        Addr:    net.JoinHostPort(host, port),
        Handler: http.HandlerFunc(func(writer http.ResponseWriter, request *http.Request) {
            log.Print(request.RequestURI) // полный URI
            log.Print(request.Method)    // метод
            log.Print(request.Header)    // все заголовки
            log.Print(request.Header.Get("Content-Type")) // конкретный заголовок

            log.Print(request.FormValue("tags")) // только первое значение Query + POST
            log.Print(request.PostFormValue("tags")) // только первое значение POST

            body, err := ioutil.ReadAll(request.Body) // тело запроса
            if err != nil {
                log.Print(err)
            }
            log.Printf("%s", body)

            err = request.ParseForm()
            if err != nil {
                log.Print(err)
            }

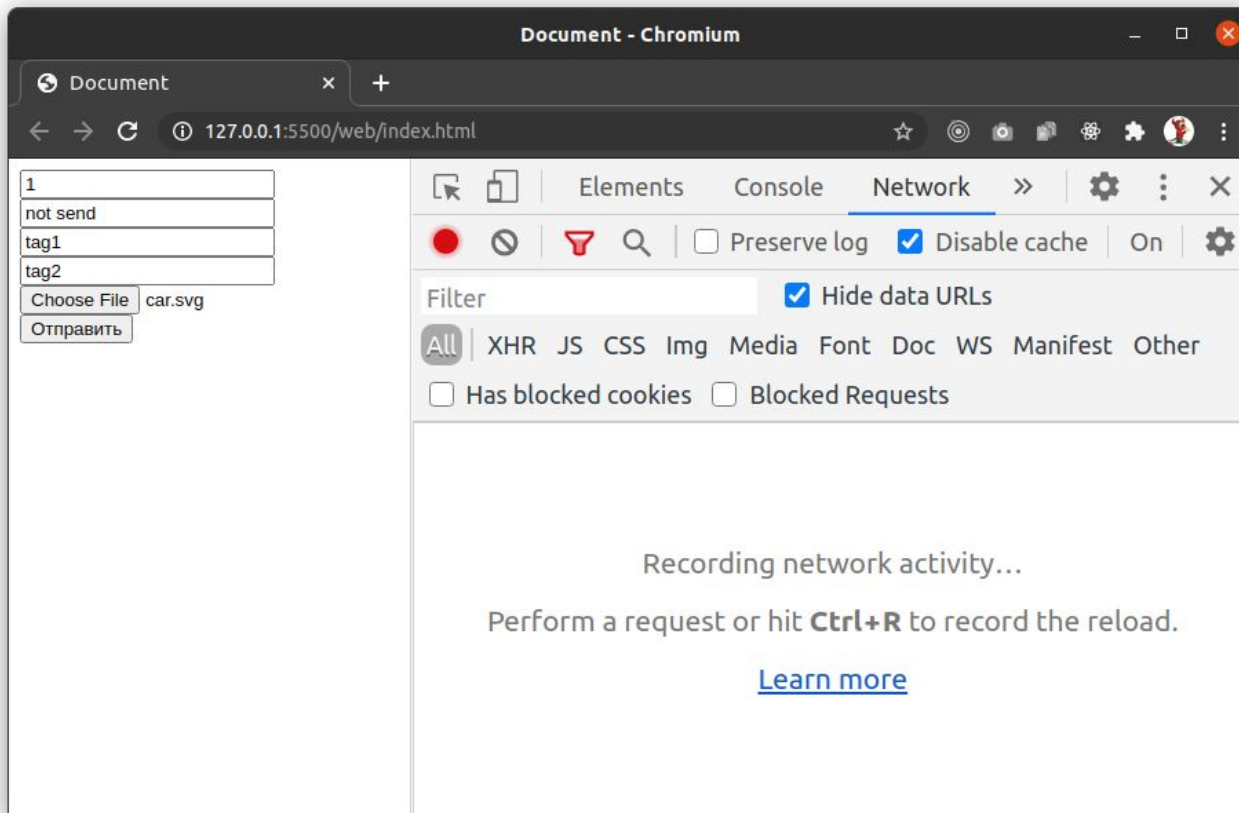
            // доступно только после ParseForm (либо FormValue, PostFormValue)
            log.Print(request.Form) // все значения формы
            log.Print(request.PostForm) // все значения формы
        })),
    }
    return srv.ListenAndServe()
}

```



# Браузер

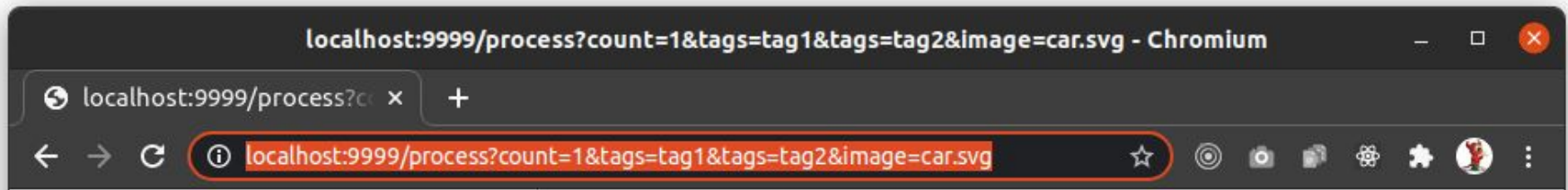
Откройте браузер, нажмите кнопку F12 (инструменты разработчика) и перейдите на вкладку Network. Заполните поля так же, как на скриншоте (включая выбор файла - файл можете использовать любой) и нажмите на кнопку "Отправить":



# Браузер

Произойдёт несколько вещей:

1. У вас в браузере откроется страница из action:



2. В боковой панельке появится информация о двух запросах:

выберите этот



Elements Console Network >> [Settings] [More]				
[Status Icons] [Filter] [Preserve log] [Disable cache] [Online] [Settings]				
Filter [x] [Hide data URLs]				
[All] XHR JS CSS Img Media Font Doc WS Manifest Other				
<input type="checkbox"/> Has blocked cookies <input type="checkbox"/> Blocked Requests				
Name	Status	Type	Size	Waterfall
[Icon] process?cou...	200	document	75 B	[Timeline]
[Icon] favicon.ico	200	text/plain	75 B	[Timeline]



# Браузер

После выбора запроса вы увидите следующее:

## ▼ General

**Request URL:** `http://localhost:9999/process?count=1&tags=tag1&tags=tag2&image=car.svg`

**Request Method:** GET

**Status Code:**  200 OK

## ▼ Query String Parameters

view source

view decoded

**count:** 1

**tags:** tag1

**tags:** tag2

**image:** car.svg

## ▼ Query String Parameters

view parsed

`count=1&tags=tag1&tags=tag2&image=car.svg`



# Браузер

Т.е. при такой отправке все поля формы (за исключением тех, у которых нет атрибута name) отправились в виде **Query** со всеми вытекающими.

Go их аккуратно обработал и положил в **Form** (который наполняется данными после **ParseForm** или **FormValue**).

RequestURI	/process?count=1&tags=tag1&tags=tag2&image=car.svg
Method	GET
Header	map[Accept: [text/html,application/xhtml+xml,application/xml;q=0.9,image/avif, image/webp,*/*;q=0.8] Accept-Language: [en-US,en;q=0.9] Cache-Control: [no-cache] Connection: [keep-alive] Content-Type: [text/html] DNT: [1] Host: [localhost:8080] Sec-Fetch-Mode: [navigate] Sec-Fetch-Site: [cross-site] Sec-Fetch-User: [?] Upgrade-Insecure-Requests: [1] User-Agent: [Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7; rv:81.0) Gecko/20100101 Firefox/81.0] X-Requested-With: [XMLHttpRequest]]
Header.Get("Content-Type")	text/html
FormValue("tags")	tag1
PostFormValue("tags")	tag2
Body	
request.Form	map[count:[1] image:[car.svg] tags:[tag1 tag2]]
request.PostForm	map[]





# Браузер

Причём самое интересное - изображение не отправилось (отправилось только имя файла). Всё просто потому, что файлы GET-запросом не отправляются (у GET нет тела).



# enctype

Мы, конечно, можем помимо action, попробовать ещё выставить enctype у формы (поскольку method и так GET), но это ни к чему не приведёт, поэтому перейдём к рассмотрению метода POST:

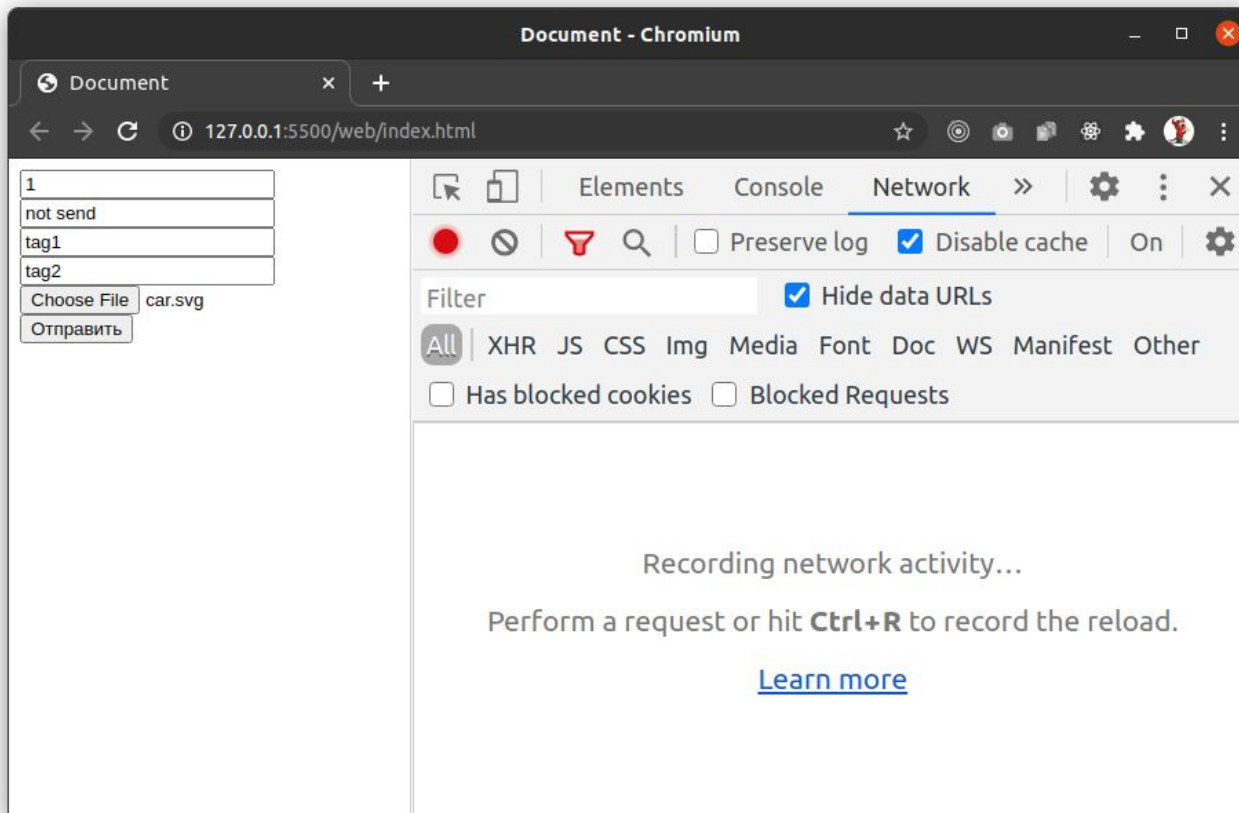
```
<form action="http://localhost:9999/process" method="POST">  
  <input type="number" name="count">  
  <input type="text">  
  <input type="text" name="tags">  
  <input type="text" name="tags">  
  <input type="file" name="image">  
  <button>Отправить</button>  
</form>
```

Нажмите кнопку назад в браузере и обновите страницу, чтобы у вас была загружена чистая форма по адресу <http://127.0.0.1:5500/web/index.html>



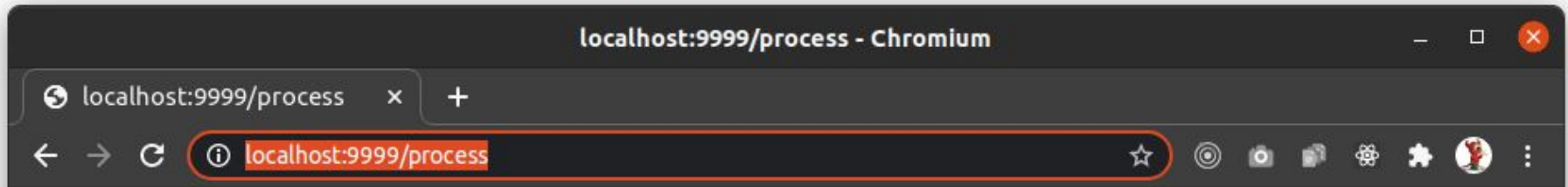
# Браузер

Снова заполните поля так же, как на скриншоте (включая выбор файла) и нажмите на кнопку "Отправить":



# Браузер

Теперь браузер перейдёт на страницу (никаких данных в URL'e не будет):



А все данные будут отправлены в теле запроса с выставлением заголовков Content-Length и Content-Type:

**Content-Length:** 41

**Content-Type:** application/x-www-form-urlencoded

**▼ Form Data**   view source   view URL encoded

count: 1  
tags: tag1  
tags: tag2  
image: car.svg

**▼ Form Data**   view parsed

count=1&tags=tag1&tags=tag2&image=car.svg



# Браузер

Т.е. при такой отправке все поля формы (за исключением тех, у которых нет атрибута name) отправились в теле запроса.

Go их аккуратно обработал и положил в `Form` (который наполняется данными после `ParseForm` или `FormValue`) и `PostForm`.

```
RequestURI      /process
Method          POST
Header          map[Accept: [text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image
ng: [gzip, deflate, br] Accept-Language: [en-US,en;q=0.9] Cache-Control: [no-cache] Connection: [keep-alive] Content-
http://127.0.0.1:5500] Pragma: [no-cache] Referer: [http://127.0.0.1:5500/web/index.html] Sec-Fetch-Dest: [document]
: [?1] Upgrade-Insecure-Requests: [1] User-Agent: [Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like G
Header.Get("Content-Type") application/x-www-form-urlencoded
FormValue("tags")      tag1
PostFormValue("tags")  tag1
Body
request.Form           map[count: [1] image: [car.svg] tags: [tag1 tag2]]
request.PostForm       map[count: [1] image: [car.svg] tags: [tag1 tag2]]
```

Обратите внимание: `Body` всегда пустой потому, что уже был вычитан в процессе обработки формы.



# Form vs PostForm

Но зачем нам `Form` и `PostForm`, если данные одинаковы? Давайте модифицируем `action` и посмотрим, что произойдёт:

```
<form action="http://localhost:9999/process?tags=query" method="POST">
  <input type="number" name="count">
  <input type="text">
  <input type="text" name="tags">
  <input type="text" name="tags">
  <input type="file" name="image">
  <button>Отправить</button>
</form>
```

## ▼ Query String Parameters

[view source](#)[view URL encoded](#)

**tags:** query

## ▼ Form Data

[view source](#)[view URL encoded](#)

**count:** 1

**tags:** tag1

**tags:** tag2

**image:** car.svg



# Браузер

Т.е. при такой отправке приоритет имеет форма, при этом в **Form** будут все данные Query + Form (Query в конце), а в **PostForm** только данные формы:

```
RequestURI      /process?tags=query
Method          POST
Header          map[Accept: [text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp
ng:[gzip, deflate, br] Accept-Language: [en-US,en;q=0.9] Cache-Control: [no-cache] Connection: [keep-alive] Co
http://127.0.0.1:5500] Pragma: [no-cache] Referer: [http://127.0.0.1:5500/web/index.html] Sec-Fetch-Dest: [doc
: [?1] Upgrade-Insecure-Requests: [1] User-Agent: [Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML,
Header.Get("Content-Type") application/x-www-form-urlencoded
FormValue("tags")      tag1
PostFormValue("tags")  tag1
Body
request.Form           map[count:[1] image:[car.svg] tags:[tag1 tag2 query]]
request.PostForm       map[count:[1] image:[car.svg] tags:[tag1 tag2]]
```

Конечно, возникает вопрос, а что делать, если нужно вытащить данные только из Query? Для этого есть **request.URL.Query()** - ничего, из того, что мы проходили - не отменяется (см. прошлую лекцию).



# Файлы

Несмотря на то, что данные теперь передаются в теле запроса, сам файл всё равно не передался, а передалось только его имя, что не очень-то полезно. Давайте посмотрим, что можно поменять.





# enctype

Итак, у атрибута enctype могут быть следующие значения:

- `application/x-www-form-urlencoded` (по умолчанию)
- `text/plain`
- `multipart/form-data`



# enctype

Давайте для интереса посмотрим на text/plain:

```
<form action="http://localhost:9999/process?tags=query" method="POST" enctype="text/plain">
  <input type="number" name="count">
  <input type="text">
  <input type="text" name="tags">
  <input type="text" name="tags">
  <input type="file" name="image">
  <button>Отправить</button>
</form>
```

**Content-Length:** 46

**Content-Type:** text/plain

## ▼ Query String Parameters

[view source](#)[view decoded](#)

**tags:** query

## ▼ Request Payload

count=1

tags=tag1

tags=tag2

image=car.svg



# Браузер

Т.е. при такой отправке данные формы отправляются в теле в формате key=value и разделены \r\n:

RequestURI	/process?tags=query
Method	POST
Header	map[Accept: [text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp;q=0.8] Accept-Language: [en-US,en;q=0.9] Cache-Control: [no-cache] Connection: [keep-alive] Content-Type: [text/plain] Pragma: [no-cache] Referer: [http://127.0.0.1:5500/web/index.html] Sec-Fetch-Dest: [document] Sec-Fetch-Mode: [cors] Requests: [1] User-Agent: [Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.0.0 Safari/537.36]
Header.Get("Content-Type")	text/plain
FormValue("tags")	query
PostFormValue("tags")	count=1
Body	tags=tag1 tags=tag2 image=car.svg
request.Form	map[tags: [query]]
request.PostForm	map[]

Go не парсит данный формат и поэтому он попадает прямиком Body. Стоит отметить, что формат достаточно редко используется.



# multipart/form-data

И остался последний enctype, который как раз-таки позволяет передавать файлы (только для метода POST):

```
<form action="http://localhost:9999/process?tags=query" method="POST" enctype="multipart/form-data">
  <input type="number" name="count">
  <input type="text">
  <input type="text" name="tags">
  <input type="text" name="tags">
  <input type="file" name="image">
  <button>Отправить</button>
</form>
```

Но для его работы нам потребуется чуть больше работы со стороны сервера (см. следующий слайд).



```

func execute(host string, port string) (err error) {
    srv := &http.Server{
        Addr:    net.JoinHostPort(host, port),
        Handler: http.HandlerFunc(func(writer http.ResponseWriter, request *http.Request) {
            log.Print(request.RequestURI) // полный URI
            log.Print(request.Method)    // метод
            log.Print(request.Header)    // все заголовки
            log.Print(request.Header.Get("Content-Type")) // конкретный заголовок

            log.Print(request.FormValue("tags")) // только первое значение Query + POST
            log.Print(request.PostFormValue("tags")) // только первое значение POST

            body, err := ioutil.ReadAll(request.Body) // тело запроса
            if err != nil {
                log.Print(err)
            }
            log.Printf("%s", body)

            err = request.ParseMultipartForm(10 * 1024 * 1024) // 10 MB
            if err != nil {
                log.Print(err)
            }

            // доступно только после ParseForm (либо FormValue, PostFormValue)
            log.Print(request.Form) // все значения формы (кроме файлов)
            log.Print(request.PostForm) // все значения формы (кроме файлов)
            // доступно только после ParseMultipart (FormValue, PostFormValue автоматически вызывают ParseMultipartForm)
            log.Print(request.FormFile("image"))
            // request.MultipartForm.Value - только "обычные поля"
            // request.MultipartForm.File - только файлы
        })),
    }
    return srv.ListenAndServe()
}

```

# multipart/form-data

```

/process?tags=query
POST
map[Accept:[text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/sig
ng:[gzip, deflate, br] Accept-Language:[en-US,en;q=0.9] Cache-Control:[no-cache] Connection:[keep-alive] Content-Length:[2998] Content-Type:[mul
ormBoundaryNCG3N33cMSakjLEX] Origin:[http://127.0.0.1:5500] Pragma:[no-cache] Referer:[http://127.0.0.1:5500/web/index.html] Sec-Fetch-Dest:[doc
etch-Site:[cross-site] Sec-Fetch-User:[?1] Upgrade-Insecure-Requests:[1] User-Agent:[Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML,
i/537.36)]

multipart/form-data; boundary=----WebKitFormBoundaryNCG3N33cMSakjLEX
query
tag1

map[count:[1] tags:[query tag1 tag2]]
map[count:[1] tags:[tag1 tag2]]
{0xc0000906f0} &{car.svg map[Content-Disposition:[form-data; name="image"; filename="car.svg"] Content-Type:[image/svg+xml]]
4 115 105 111 110 61 34 49 46 48 34 32 101 110 99 111 100 105 110 103 61 34 105 115 111 45 56 56 53 57 45 49 34 63 62 13 10 60 33 45 45 32 71 10
100 111 98 101 32 73 108 108 117 115 116 114 97 116 111 114 32 49 57 46 48 46 48 44 32 83 86 71 32 69 120 112 111 114 116 32 80 108 117 103 45 7
115 105 111 110 58 32 54 46 48 48 32 66 117 105 108 100 32 48 41 32 32 45 45 62 13 10 60 115 118 103 32 118 101 114 115 105 111 110 61 34 49 46

```

↑  
сами данные





# multipart/form-data

Но как же реально отправляются данные? Почему называется multipart/form-data?

Давайте не будем "парсить" (т.е. разбирать форму):

```
func execute(host string, port string) (err error) {
    srv := &http.Server{
        Addr: net.JoinHostPort(host, port),
        Handler: http.HandlerFunc(func(writer http.ResponseWriter, request *http.Request) {
            log.Print(request.RequestURI)           // полный URI
            log.Print(request.Method)               // метод

            body, err := ioutil.ReadAll(request.Body) // тело запроса
            if err != nil {
                log.Print(err)
            }
            log.Printf("%s", body)
        }),
    }
    return srv.ListenAndServe()
}
```



# multipart/form-data

Тогда мы увидим полное тело запроса:

```
/process?tags=query
POST
-----WebKitFormBoundaryNCG3N33cMSakjLEX
Content-Disposition: form-data; name="count"

1
-----WebKitFormBoundaryNCG3N33cMSakjLEX
Content-Disposition: form-data; name="tags"

tag1
-----WebKitFormBoundaryNCG3N33cMSakjLEX
Content-Disposition: form-data; name="tags"

tag2
-----WebKitFormBoundaryNCG3N33cMSakjLEX
Content-Disposition: form-data; name="image"; filename="car.svg"
Content-Type: image/svg+xml

<?xml version="1.0" encoding="iso-8859-1"?>
```

те самые части, разделённые  
спец.набором символов  
(их генерирует сам браузер)

**Content-Length:** 2998

**Content-Type:** multipart/form-data; boundary=----WebKitFormBoundaryNCG3N33cMSakjLEX





# MultipartFile

FormFile возвращает интерфейс, который выглядит следующим образом:

```
// File is an interface to access the file part of a multipart message.  
// Its contents may be either stored in memory or on disk.  
// If stored on disk, the File's underlying concrete type will be an *os.File.  
type File interface {  
    io.Reader  
    io.ReaderAt  
    io.Seeker  
    io.Closer  
}
```

Это embedding интерфейсов, который означает, что File содержит все методы из интерфейсов `io.Reader`, `io.ReaderAt`, `io.Seeker`, `io.Closer`.

Как работать с `Reader` и `Closer` вы уже знаете.



# MultipartFile

**FormFile** возвращает также информацию об имени файла и т.д.:

```
// A FileHeader describes a file part of a multipart request.
type FileHeader struct {
    Filename string
    Header   textproto.MIMEHeader
    Size     int64

    content []byte
    tmpfile string
}
```

Соответственно, пользуясь этими данными, а также пакетом **io** и сопутствующими, вы можете легко реализовать загрузку файла (т.к. у вас есть **Reader** содержимого).



# ИТОГИ



# ИТОГИ

Сегодня мы познакомились с обработкой форм. Большая часть из изученного нами (особенно Query и Multipart) будут активно использоваться в дальнейшей разработке.



# ДОМАШНЕЕ ЗАДАНИЕ



# Орг.моменты

Курс состоит из 33 обязательных занятий. Мы выкладываем видео-уроки каждый четверг и субботу к 20:00 (по Душанбе). PDF-материал выкладывается в 20:00 каждую среду и пятницу.

В течении 24 часов мы выкладываем запись вебинара. **Каждый вторник 12:00 (по Душанбе) дедлайн** сдачи домашнего задания.

Если не успеете сдать в срок домашнее задания, тогда этот курс будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.

Все вопросы вы сможете задавать в [телеграм-чате](#).



# ДЗ: Save

ДЗ у вас будет всего одно, но достаточно интересное: нужно переделать сервер так, чтобы теперь данные для save приходили не в Query, а в теле запроса и среди них содержалась картинка:

## Депозиты с доходом 60 на 40

Вкладывайте депозит в национальной  
валюте и получайте больше половины дохода

Рассчитать доход



< 1/3 >

Бот сам будет присылать вам эту картинку со всеми полями в POST-запросе.



# Д3: Save

Вы должны будете обработать весь запрос, а картинку сохранить в каталог web/banners:



Обратите внимание: Git не сохраняет пустые каталоги. Поэтому обычно в них кладут пустой файл с именем .gitkeep.





## Д3: Save

После сохранения вы записываете имя картинки (не полный путь, а только имя) в поле Image:

```
22 // Banner представляет собой баннер.  
23 type Banner struct {  
24     ID        int64  
25     Title     string  
26     Content   string  
27     Button    string  
28     Link      string  
29     Image     string  
30 }
```

Имя картинки формируется как: ID + расширение. Например, если баннер сохраняется под ID = 1 и присылается картинка с расширением .png, то вы сохраняете её под именем "1.png" (чем это плохо и к каким последствиям может привести, мы поговорим на следующей лекции).



Спасибо за внимание

alif academy

2020

