Лекция 7

План

- Web-приложения
- Sinatra
- Инструменты

Web-приложения

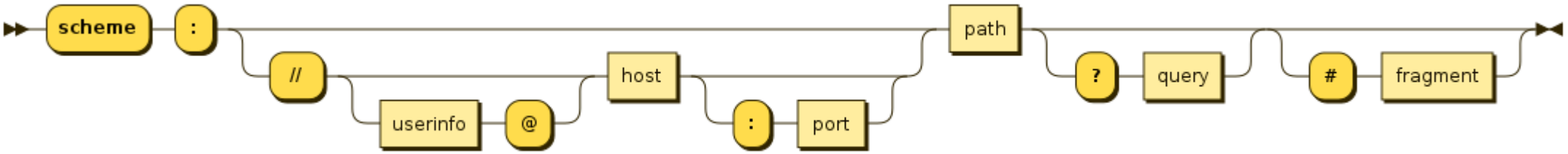
Что отличает web-приложения?

1. Архитектура
2. Uptime
3. Многопользовательское приложение
4. Многопоточное приложение
5. Набор шаблонных действий

Базовые действия web-приложения

1. Динамический рендеринг страниц
2. Хранение данных и доступ к ним
3. Механизмы авторизации
4. Интеграции с другими web приложениями

URL и URI



<схема>://[//[<логин>[:<пароль>]@]<хост>[:<порт>]]
[/<URL-путь>][?<параметры>][#<якорь>]|

Зарезервированные домены:

example, localhost, invlaid, test

Коды HTTP

- 1XX - информационные
- 2XX - успех
- 3XX - перенаправление
- 4XX - ошибка клиента
- 5XX - ошибка сервера

Некоторые полезные коды

- 200 - OK
- 307,308 - временное и постоянное перенаправление
- 400 - Ошибка клиента
- 403 - Запрещено
- 404 - Не найдено
- 418 - "Я чайник"
- 500 - Ошибка сервера
- 502 - Bad Gateway
- 503 - Сервер недоступен
- 504 - Тайм-аут

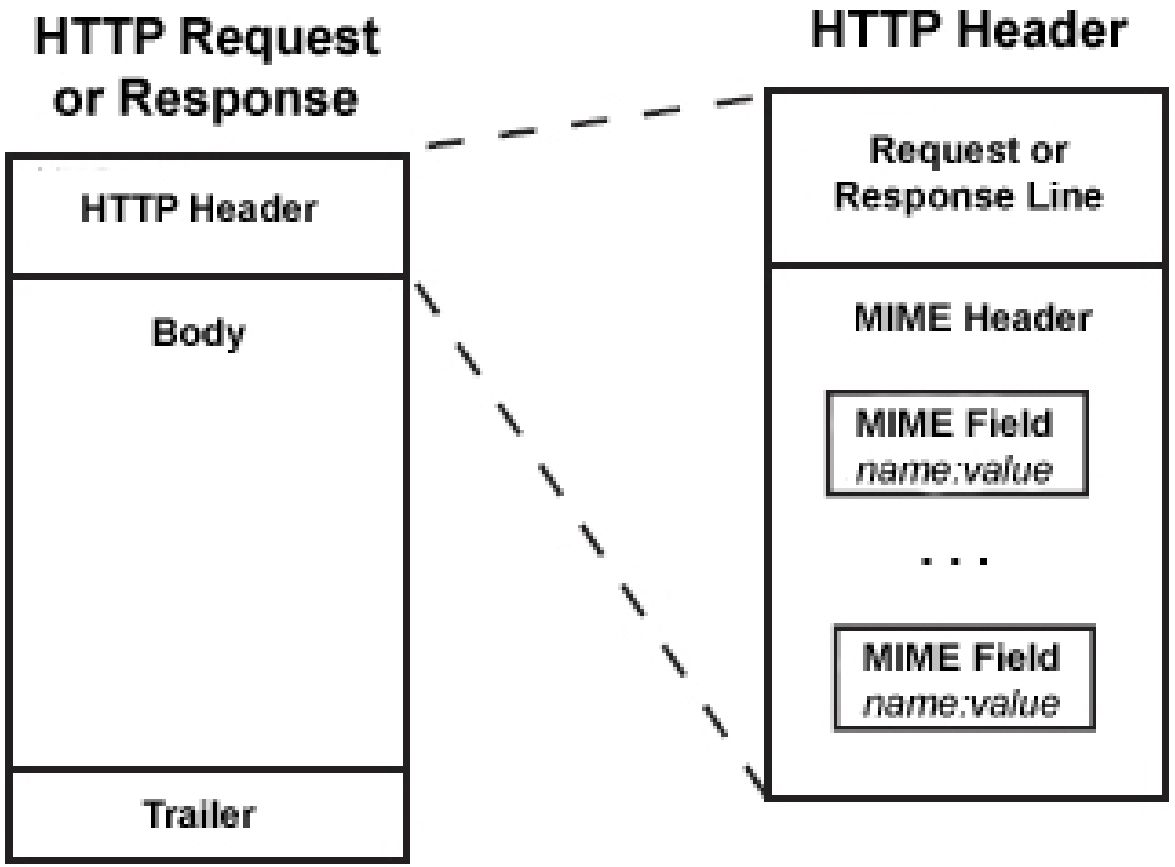
Глаголы HTTP

- **GET** Запрашивает представление ресурса
- **HEAD** Запрашивает ресурс так же, как и метод GET, но без тела ответа
- **POST** Используется для отправки сущностей к определённому ресурсу
- **PUT** Заменяет все текущие представления ресурса данными запроса
- **DELETE** Удаляет указанный ресурс.
- **CONNECT** Устанавливает "туннель" к серверу, определённому по ресурсу
- **OPTIONS** Используется для описания параметров соединения с ресурсом.
- **TRACE** Выполняет вызов возвращаемого тестового сообщения с ресурса
- **PATCH** Используется для частичного изменения ресурса.

Заголовки запросов

- General Headers — используются в запросах и ответах.
- Request Headers — используются только в запросах.
- Response Headers — используются только в ответах.
- Entity Headers — сопровождают каждую сущность сообщения.

Структура заголовка



Важные заголовки

- Content-Type
- Cookie
- Expect
- Forwarded
- From
- User-Agent
- If-Modified-Since

Структура запроса

```
GET https://httpbin.org/ip
Accept: application/json

POST https://httpbin.org/post
Content-Type: application/json

{
    "id": 999,
    "value": "content"
}
```

Структура запроса

```
POST https://httpbin.org/post
Content-Type: application/x-www-form-urlencoded

id=999&value=content

POST https://httpbin.org/post
Content-Type: multipart/form-data; boundary=WebAppBoundary
```

JSON – что может быть внутри?

- Объект — это неупорядоченное множество пар ключ:значение, заключённое в фигурные скобки «{ }». Ключ описывается строкой, между ним и значением стоит символ «:». Пары ключ-значение отделяются друг от друга запятыми.

- Массив — это упорядоченное множество значений. Массив заключается в квадратные скобки «[ ]». Значения разделяются запятыми.

- Число.

- Литералы true, false и null.

- Строка — это упорядоченное множество из нуля или более символов юникода, заключённое в двойные кавычки.

Как организовать запросы?

- REST vs SOAP

SOAP

- Simple Object Access Protocol

Envelope – Корневой элемент, который определяет сообщение и пространство имен, использованное в документе.

Header – Содержит атрибуты сообщения, например: информация о безопасности или о сетевой маршрутизации.

Body – Содержит сообщение, которым обмениваются приложения.

Fault – Необязательный элемент, который предоставляет информацию об ошибках, которые произошли при обработке сообщений.

SOAP-запрос

```
<?xml version="1.0" encoding="utf-8"?>
    <soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
          <getProductDetails xmlns="http://warehouse.example.com/ws">
                    <productID>12345</productID>
          </getProductDetails>
</soap:Body>
</soap:Envelope>
```

Почему не SOAP?

- Размер
- Еще один способ передать параметры
- Можно передавать названия действий

REST

- Representational State Transfer (передача состояния представления)

- Клиент-серверная архитектура
- Отсутствие состояния
- Возможность кэширования
- Единообразность интерфейсов
- Многослойность
- Передача кода по требованию

Единообразность интерфейсов

- Идентификация ресурсов
- Манипуляция ресурсами через представление
- «Самоописываемые» сообщения
- Гипермедиа как средство изменения состояния приложения

Реализуем CRUD по REST

Глагол	Путь	C#A	Использование
GET	/photos	photos#index	Список всех фото
GET	/photos/new	photos#new	Форма для нового фото
POST	/photos	photos#create	Создает новое фото
GET	/photos/:id	photos#show	Отображает отдельное фото
GET	/photos/:id/edit	photos#edit	Форма редактирования фото
PATCH/PUT	/photos/:id	photos#update	Обновляет фото
DELETE	/photos/:id	photos#destroy	Удаляет фото

Как отличить REST от всего остального?

- `GET /api/users/do_something`
- `GET /users/1?action=delete,`

Как быть в сложных случаях?

- Ресурс может быть в единственном числе или во множественном
- У ресурса могут вложенные ресурсы
- Дробить ли ресурсы?
- О чем думает клиентское приложение?

Пытаемся спроектировать: подход 1: Крупные ресурсы

- Делаем сложные ресурсы со вложенными полями
- Разрешаем обновлять ресурс целиком
- Запрещаем обновлять отдельные поля
- Экономим на реализации
- Теряем часть информации
- Обновление структуры ресурса может вызвать проблемы у клиентов

Подход 2: Ресурс – это действие

- Делаем ресурс ChangeOfAddress
- Клевая модель на сообщениях
- Инкапсуляция
- Сложно хорошо спроектировать (но легко дополнить)

CQRS

- command-query responsibility segregation

- Метод должен быть либо *командой*, выполняющей какое-то действие, либо *запросом*, возвращающим данные, но не одновременно.

REST без PUT

- Запрещаем частичное обновление ресурсов
- Выполняем POST на команды (C)
- Выполняем GET на запросы (Q)

Хорошие примеры

```
https://api.github.com/repos/vmg/redcarpet/issues?state=closed
```

```
https://api.github.com/authorizations
                           {"scopes":["public_repo"]}
```

Плохие примеры

- POST lists/create
- POST lists/destroy
- POST lists/members/create
- POST lists/members/create_all
- POST lists/members/destroy
- POST lists/members/destroy_all
- POST lists/subscribers/create
- POST lists/subscribers/destroy
- POST lists/update

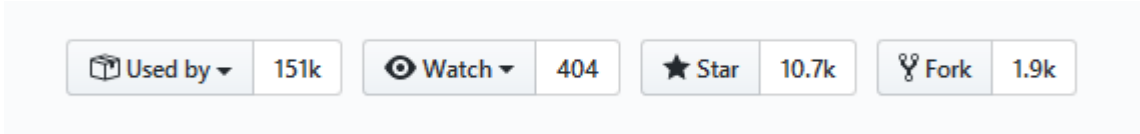
Проектируем сервис футбольного агента

- Есть база игроков
- Игрока можно продать, сдать в аренду, отправить на осмотр или расторгнуть контракт.
- Можно обновлять детали

Базовый URL

- GET /players/:id
- DELETE /players/:id
- POST /players/:id/transfer {team: “Rostov”}
- POST /players/:id/rent {team: “Rostov”}

Sinatra



- Gem для создания простейших web-приложений

```ruby
require 'sinatra'
get '/frank-says' do
  'Put this in your pipe & smoke it!'
end
```

Полный цикл разработки

- `Gem install sinatra`
- `app.rb`
- `ruby app.rb`

Результат

Put this in your pipe & smoke it!

```
Puma starting in single mode...
* Version 3.12.1 (ruby 2.6.3-p62), codename: Llamas in Pajamas
* Min threads: 0, max threads: 16
* Environment: development
* Listening on tcp://localhost:4567
Use Ctrl-C to stop
::1 - - [17/Oct/2019:22:58:02 +0300] "GET / HTTP/1.1" 404 458 0.0187
::1 - - [17/Oct/2019:22:58:02 +0300] "GET /__sinatra__/404.png HTTP/1.1" 200 18893 0.0048
::1 - - [17/Oct/2019:22:58:02 +0300] "GET /favicon.ico HTTP/1.1" 404 469 0.0007
::1 - - [17/Oct/2019:22:58:40 +0300] "GET / HTTP/1.1" 404 458 0.0006
::1 - - [17/Oct/2019:22:58:40 +0300] "GET /__sinatra__/404.png HTTP/1.1" 304 - 0.0016
::1 - - [17/Oct/2019:22:59:24 +0300] "GET / HTTP/1.1" 404 458 0.0008
::1 - - [17/Oct/2019:22:59:37 +0300] "GET /frank-says HTTP/1.1" 200 33 0.0005
```

Пишем приложение

```
get '/' do
  .. show something ..
end

post '/' do
  .. create something ..
end

put '/' do
  .. replace something ..
end

patch '/' do
  .. modify something ..
end
```

```
delete '/' do
  .. annihilate something ..
end

options '/' do
  .. appease something ..
end

link '/' do
  .. affiliate something ..
end

unlink '/' do
  .. separate something ..
end
```

Используем шаблоны для параметров

```ruby
get '/hello/:name' do
  # matches "GET /hello/foo" and "GET /hello/bar"
  # params['name'] is 'foo' or 'bar'
  "Hello #{params['name']}!"
end

get '/hello/:name' do |n|
  # matches "GET /hello/foo" and "GET /hello/bar"
  # params['name'] is 'foo' or 'bar'
  # n stores params['name']
  "Hello #{n}!"
end
```

Wildcards

```ruby
get '/say/*/to/*' do
  # matches /say/hello/to/world
  params['splat'] # => ["hello", "world"]
end

get '/download/*.*' do
  # matches /download/path/to/file.xml
  params['splat'] # => ["path/to/file", "xml"]
end
```

Splat-block

```
get '/download/*.*' do |path, ext|
  [path, ext] # => ["path/to/file", "xml"]
end
```

Регулярные выражения

```ruby
get /\/hello\/([\w]+)/ do
  "Hello, #{params['captures'].first}!"
end


get %r{/hello/([\w]+)} do |c|
  # Matches "GET /meta/hello/world", "GET /hello/world/1234" etc.
  "Hello, #{c}!"
end
```

Необязательные параметры

```
get '/posts/:format?' do
  # matches "GET /posts/" and any extension "GET /posts/json", "GET /posts/xml" etc
end
```

Параметры из запроса

```
get '/posts' do
  # matches "GET /posts?title=foo&author=bar"
  title = params['title']
  author = params['author']
  # uses title and author variables; query is optional to the /posts route
end
```

Условия в URL

```
get '/foo', :agent => /Chrome (\d\.\d)[\d\/]*?/ do
  "You're using Chrome version #{params['agent'][0]}"
end

get '/foo' do
  # Matches non-songbird browsers
end
```

Тип данных

```
get '/', :provides => 'html' do
  haml :index
end

get '/', :provides => ['rss', 'atom', 'xml'] do
  builder :feed
end
```

Пишем свои условия

```
set(:probability) { |value| condition { rand <= value } }

get '/win_a_car', :probability => 0.1 do
  "You won!"
end

get '/win_a_car' do
  "Sorry, you lost."
end
```

Условия с несколькими значениями

```
set(:auth) do |*roles|   # <- notice the splat here
  condition do
    unless logged_in? && roles.any? {|role| current_user.in_role? role }
      redirect "/login/", 303
    end
  end
end

get "/my/account/", :auth => [:user, :admin] do
  "Your Account Details"
end

get "/only/admin/", :auth => :admin do
  "Only admins are allowed here!"
end
```

Rack

```ruby
class HelloWorld
  def call(env)
    [200, {"Content-Type" => "text/plain"}, ["Hello world!"]]
  end
end
```

Middleware

```
class RackMiddleware
  def initialize(app)
    @app = app
  end

  def call(env)
#…before
    status, headers, body = @app.call(env)
#…after
 end
end
```

Что может вернуть Синатра?

1. [status (Fixnum), headers (Hash),
   response body (responds to #each)]

2. [status (Fixnum), response body (responds to #each)]

3. An object that responds to #each and
   passes nothing but strings to the given block

4. A Fixnum representing the status code

Стриминг!

```ruby
class Stream
  def each
    100.times { |i| yield "#{i}\n" }
  end
end

get('/') { Stream.new }
```

Статические файлы

```
set :public_folder, File.dirname(__FILE__) + '/static'
```

Почему мы должны тратить на это время???

Reverse Proxy

- Обратный прокси-сервер может скрывать существование опрашиваемых им серверов и их характеристики.
- Применение программного файрвола (брандмауэр) в обратном прокси-сервере может защитить от наиболее распространенных веб-атак, таких как DOS или DDOS. Без обратного прокси-сервера удаление вредоносного ПО может оказаться непростой задачей.
- Основной веб-сайт может не поддерживать подключение по SSL, однако это можно реализовать с помощью обратного прокси-сервера, который может быть оборудован аппаратным SSL-ускорителем.
- Выполнение функций балансировщика нагрузки между несколькими серверами, подменяя URL таким образом, чтобы использовался наиболее уместный сервер.
- Уменьшение нагрузки на основные серверы благодаря кэшированию статического и динамического контента. Эта возможность известна как акселерация веб-сайтов. Сервер может отсортировать свой кэш по частоте запросов к контенту, что значительно уменьшит нагрузку на основные серверы.
- Сжатие содержимого для уменьшения времени его загрузки.

Генерация динамического контента

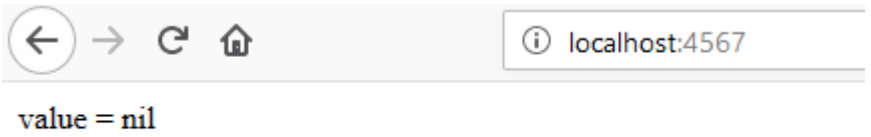
```ruby
get '/' do
  erb :index
end

get '/' do
  code = "<%= Time.now %>"
  erb code
end
```

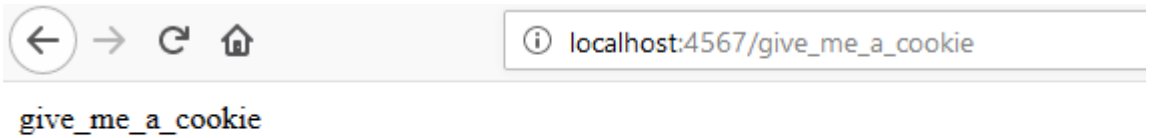
Фильтры

```
before do
  @note = 'Hi!'
  request.path_info = '/foo/bar/baz'
end

after do
  puts response.status
end
```

Сессии

```
enable :sessions

get '/' do
  "value = " << session[:value].inspect
end

get '/:value' do
  session['value'] = params['value']
end
```





Управление потоком выполнения

```
halt 401, 'go away!'
halt 402, {'Content-Type' => 'text/plain'}, 'revenge'
halt erb(:error)
```

```
get '/guess/:who' do
  pass unless params['who'] == 'Frank'
  'You got me!'
end

get '/guess/*' do
  'You missed!'
end
```

Вызов другого обработчика

```ruby
get '/foo' do
  status, headers, body = call env.merge("PATH_INFO" => '/bar')
  [status, headers, body.map(&:upcase)]
end

get '/bar' do
  "bar"
end
```

Настоящий стриминг

```
get '/' do
  stream do |out|
    out << "It's gonna be legen -\n"
    sleep 0.5
    out << " (wait for it) \n"
    sleep 1
    out << "- dary!\n"
  end
end
```

Логирование

```
class MyApp < Sinatra::Base
  configure :production, :development do
    enable :logging
  end
end
```

Конфигурирование

```ruby
configure do
  # setting one option
  set :option, 'value'

  # setting multiple options
  set :a => 1, :b => 2

  # same as `set :option, true`
  enable :option

  # same as `set :option, false`
  disable :option

  # you can also have dynamic settings with blocks
  set(:css_dir) { File.join(views, 'css') }
end
```

Rack::Protection

- Включена по умолчанию (если есть сессии!)

```
disable :protection

set :protection, :except => [:path_traversal, :session_hijacking]
```

Обработка ошибок

```
error do
  'Sorry there was a nasty error - ' + env['sinatra.error'].message
end

error MyCustomError do
  'So what happened was...' + env['sinatra.error'].message
end


get '/' do
  raise MyCustomError, 'something bad'
end
```

Тестирование

```ruby
class MyAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_my_default
    get '/'
    assert_equal 'Hello World!', last_response.body
  end

  def test_with_params
    get '/meet', :name => 'Frank'
    assert_equal 'Hello Frank!', last_response.body
  end

  def test_with_user_agent
    get '/', {}, 'HTTP_USER_AGENT' => 'Songbird'
    assert_equal "You're using Songbird!", last_response.body
  end
end
```

```ruby
require 'my_sinatra_app'
require
'minitest/autorun'
require 'rack/test'
```

Mocks

```
get '/path', params={}, rack_env={}
```

```
app
last_request
last_response
```

Первые шаги к интеграционному тестированию

```ruby
class HelloWorldTest < Test::Unit::TestCase
  include Capybara::DSL
  # Capybara.default_driver = :selenium # <-- use Selenium driver

  def setup
    Capybara.app = Sinatra::Application.new
  end

  def test_it_works
    visit '/'
    assert page.has_content?('Hello World')
  end
end
```

```ruby
ENV['APP_ENV'] = 'test'

require 'hello_world'  # <-- your sinatra app
require 'capybara'
require 'capybara/dsl'
require 'test/unit'
```

Модульная или классическая форма

```ruby
require 'sinatra/base'

class MyApp < Sinatra::Application
  get '/' do
    'Hello world!'
  end
end
```

```ruby
require 'sinatra/base'
my_app = Sinatra.new { get('/') { "hi" } }
my_app.run!
```

Что делать, если просят еще?

```
require 'sinatra'
require 'sinatra/content_for'

require 'sinatra'
require 'sinatra/contrib'

require 'sinatra'
require 'sinatra/contrib/all'
```

O Silicium

1. Краткие доклады о запланированных фичах
2. Документация!
3. Первые PR в основной репозиторий

Ссылки

https://developer.mozilla.org/ru/docs/Web/HTTP/%D0%97%D0%B0%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BA%D0%B8

https://www.thoughtworks.com/insights/blog/rest-api-design-resource-modeling

https://habr.com/ru/post/251193/

http://sinatrarb.com/documentation.html

https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D1%8B%D0%B9_%D0%BF%D1%80%D0%BE%D0%BA%D1%81%D0%B8

https://www.getpostman.com/use-cases/api-first-development