Введение в библиотеку построения web-приложений Sinatra

Андрей Васильев

2019

Передача документов в сети Интернет

Базовыми протоколами, на основании которых происходит соединение между браузером и сервером являются TCP и IP:

- IP обеспечивает всем уникальные адреса в сети Интернет
- TCP обеспечивает надёжную передачу данных между конкретными узлами и приложениями в сети Интернет

В рамках протокола TCP передача данных происходит кусочками, называемыми пакетам. Для передачи одного документа могут потребоваться сотни пакетов

Поверх протокола TCP реализован протокол HTTP, Hyper Text Transport Protocol - базовый протокол для передачи документов, применяемый в сети Интернет

По протоколу HTTP передаются HTML-документы, CSS-документы, JavaScript-документы, JASON-документы и т.д.

Роли в протоколе HTTP

Данный протокол выделяет две стороны:

- Сервер постоянно работает и ожидает подключения
- Клиенты инициализируют соединение и выполняют различные запросы к серверу
- Прокси-серверы прозрачно пропускают соединения от клиента к серверу для кеширования, фильтрации и т.д.

В качестве клиентов могут выступать любые приложения, которые реализуют клиентскую часть протокола HTTP

Процесс выполнения HTTP-запроса

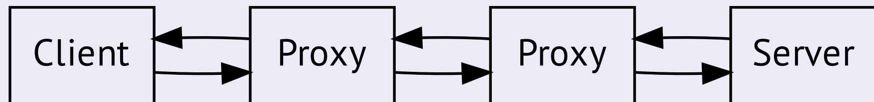

Рис.: Запрос от клиента к серверу

Базовые аспекты протокола HTTP

- Изначально протокол задумывался как простой для понимания и доступный для чтения обычным человеком
- Протокол HTTP поддерживает возможность расширения путём указания произвольного набора заголовков
- Протокол HTTP не включает средств для сохранения состояния, т.е. запросы от клиента обрабатываются на сервере не зависимо друг от друга
- Использование заголовка cookies позволят описывать сессии, которые позволяют сохранять состояние состояния между запросами между сервером и клиентом

Базовый процесс выполнения HTTP-запроса

Рассмотрим шаги, которые необходимо выполнить для выполнения запроса от клиента к серверу

1. Клиент открывает TCP-соединение с сервером, которое используется для отправки сообщения, приёма ответа
2. Клиент посылает HTTP-сообщение, например:

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: ru
```

3. Сервер отвечает документом на сообщение:

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
...
 <!DOCTYPE html...
```

4. Клиент закрывает или переиспользует TCP-соединение

Разбор структуры HTTP-запроса

```
GET / HTTP/1.1
Host: uniyar.ac.ru
Accept-Language: ru
```

- GET - метод (тип запроса)
- / - путь по которому осуществляется запрос
- HTTP/1.1 - версия протокола
- Затем идёт список заголовков, сначала идёт название, а после двоеточия его значение
- После заголовков может идти тело запроса, его данные

Типы запросов

- GET-запрос на получение документа по ссылке
- POST-запрос на передачу данных по ссылке
- PUT-запрос на обновление ресурса по ссылке
- DELETE-запрос на удаление ресурса по ссылке

Разбор HTTP-ответа

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
...
```

- `HTTP/1.1` - версия протокола по которому происходит ответ
- `200` - код ответа сервера, указывающий результат обработки
- `OK` - краткое и неполное описание кода ответа
- Список заголовков ответа
- После заголовка обычно идёт тело ответа, включающее документ

Пути к документам в HTTP

Для определения пути к документу используется специальный формат, называемый URI, Universal Resource Identifier:

```
[схема ":"] путь ["?" запрос] ["#" фрагмент]
```

схема описывает способ связи

- `file` - файл на локальной файловой системе
- `http` - досту по протоколу HTTP

путь путь к файлу, зависит от схемы

- Абсолютный - путь от корня ресурса
- Относительный - путь от текущего документа

запрос дополнительные данные, передаваемые в запросе
фрагмент выделенный компонент в документе

Реализация серверного HTTP-стека в Ruby

В основе реализации всего современного серверного HTTP-стека для Ruby приложений лежит библиотека Rack

- Предлагает подход для обработки входящих запросов и формирования ответов на них
- Описывает стандартные интерфейсы как для веб-серверов, так и для приложений

Существует множество веб-серверов: `puma`, `unicorn`, `thin` и т. д., обладающими различными свойствами

На основе библиотеки Rack созданы библиотеки и фреймворки: Ruby on Rails, Sinatra, Hanami, RODA, Cuba

Такой подход позволяет

- серверам улучшать производительность и эффективность
- разработчикам библиотек позволяет сосредоточиться на создании подходящего API для пользователей

Простейшее веб-приложение на Sinatra

Сначала необходимо установить джем `sinatra`:

```
bundle add sinatra
```

Затем необходимо создать файл приложения:

```ruby
require 'sinatra'

get '/' do
  'Hello world!'
end
```

Осталось запустить данный файл: `ruby app.rb`

Описание обработчиков маршрутов

При обращении клиента к серверу он указывает путь к ресурсу, с которым хочет взаимодействовать

На стороне сервера нам необходимо указать список путей, которые сервер сможет обработать

Для описания списка поддерживаемых путей используются специальные методы: `get`, `post`, `put`, `patch`, `delete`, `options`, `link`, `unlink`; они определяют тип запроса

```
get '/abc' do
  # ...
end
```

Каждому методу в качестве аргумента передаётся строка-шаблон, описывающая набор подходящих путей

Ассоциированный блок должен обработать запрос и вернуть ответ на данный запрос

Описание шаблонов путей

При поступлении запроса его путь сравнивается с шаблонами в порядке их определения в приложении до первого совпадения

В описании шаблона пути можно использовать параметры

- Именованные параметры, доступные в хеше `params`

```
get '/hello/:name' do
  info = params['name']
```

- Именованные параметры, доступные в блоке

```
get 'hello/:name' do |name|
```

- Произвольные последовательности символов

```
get '/say/*/to/*' do
  params['splat'] # => [значение №1, значение
```

- Также доступны через параметры блока

```
get '/download/*.*/' do |path, ext|
```

Возвращаемое значение блока

Основная задача тела блока, ассоциированного с методом, - возвратить содержимое, которое ожидает клиент

Результатом работы блока могут быть:

- Строка, содержащая документ для клиента. Код ответа в этом случае будет 200
- Массив из трёх элементов: код ответа, хеш из заголовков, тело ответа
- Массив из двух элементов: код ответа, тело ответа
- Объект, имеющий итератор `#each`, который возвращает только строки
- Целое число, обозначающее код ответа

Статические файлы

Любое веб-приложение состоит из динамических и статических документов. Чем больше у вас вторых, тем легче обслужить большое количество клиентов.

По умолчанию статические файлы располагаются в каталоге `public`, находящемся рядом с корневым файлом приложения

Для установки альтернативного пути к файлам необходимо установить значение переменной `:public_folder`:

```
set :public_folder, File.join(__dir__, 'static')
```

Статические файлы доступны по пути без каталога `public`. Например файл `./public/css/style.css` будет доступен по `http://example.com/css/style.css`

Использование шаблонов

Для формирования содержимого страниц желательно использовать файлы-шаблоны

- Шаблоны располагаются в подкаталоге `viewes`
- Sinatra использует библиотеку `tilt` для отображения шаблонов, что позволяет воспользоваться более 20 различными шаблонизаторами

Для отображения шаблона `data.erb` из папки `views` используйте `erb :data`:

```
get '/data' do
  erb :data
end
```

Можно переопределить местоположение каталога с видами, установив значение переменной `:views`

Параметры шаблонов

При формировании шаблона учитывается ряд параметров их можно указать

- либо глобально, с помощью `set :option, value`
- либо локально при вызове генератора шаблона в качестве опций, `erb :info, option: value`

Доступные параметры

- `:locals` - список локально доступных переменных
- `:layout` - использовать указанную раскладку для формирования страницы или не использовать макет совсем
- `:content_type` - тип документа, который будет создан после обработки шаблона; также зависит от шаблонизатора
- `:views` - каталог, откуда следует брать шаблоны
- `:layout_options` - опции, которые будут использованы только для отображения макета

Доступ к переменным внутри шаблонов

Шаблоны выполняются в рамках того же контекста, что и обработчики маршрутов. Переменные экземпляра, определённые в обработчике доступны в шаблоне

```
get '/data/:id' do
  @info = Info.find(params['id'])
  erb 'Info: <%= @info %>'
end
```

Также можно явно указать список локальных переменных через указание опции `:locals`

```
get '/page/:number' do
  data = Info.find(params['number'])
  erb 'Info: <%= data %>', locals: { data: data }
end
```

Шаблоны с yield и встроенные шаблоны

Раскладкой называют шаблон, внутри которого используется `yield`. Вместо `yield` будет вставлен другой шаблон

Их можно использовать явно:

```
erb :post, :layout => false do
  erb :index
end
```

Эта конструкция в основном эквивалентна

```
erb :index, layout: :post
```

Если вы назовёте файл-шаблон с вызовом `yield` именем `layout.erb`, то он будет использоваться как раскладка по умолчанию

Настройка Sinatra

Блоки конфигураций лучше всего помещать в вызов метода `configure`, который будет выполняться 1 раз при старте

```
configure do
  # Задание одной опции
  set :option, 'value',
  # Задание нескольких опций
  set a: 1, b: 2
  # Установка `true` значением опции
  enable :option
  # Установка `false` значением опции
  disable :option
end
```

В настройках можно хранить объекты, описывающие глобальное состояние веб-приложения

Получение ввода от пользователя

При работе с формами возможны 2 сценария:

- Получение существующих данных, фильтрация
- Изменение или добавление новых данных

В первом случае изменение состояния данных на сервере не происходит, а во втором подразумевается изменение состояния

Для первого случая следует использовать `GET`-запросы, а для второго `POST`-запросы

Сценарий обработки POST-запросов

1. Получить POST-запрос от клиента
2. Обработать данные POST-запроса
3. Если данные верны, тогда изменить состояние сервера и ответить клиенту перенаправлением к виду, где можно посмотреть на изменение: redirect to('/view')

Если нет, тогда вернуть заполненную форму

Описание формы

```
<form action="/search" method="get">
  <label for="travel-date">Когда?</label>
  <input type="date" id="travel-date" placeholder
  <label for="destination">Куда?</label>
  <input type="text" id="destination" placeholder
  <label for="luxurious">Богато?</label>
  <input type="checkbox" id="luxurious">
  <button type="submit">Искать</button>
</form>
```

- `action` указывает путь, по которому будут отправлены данные
- `method` указывает тип запроса, который будет отправлен
- атрибут `id` у полей ввода, `input` описывает имена полей, которые будут переданы веб-приложению

Получение данных с формы

Вне зависимости от типа запроса, который отправляется веб-приложению (GET или POST), данные с формы будут доступны в хеше `params`

```
get '/search' do
  when = params['travel-date']
  where = params['destination']
  rich = params['luxurious']
  @data = find_destination(when, where, rich)
  erb :search
end
```

Обработка POST-запросов

POST-запросы предназначены для изменения состояния веб-приложения, поэтому они сами выводят текст только в случае ошибочного ввода. В случае успеха надо перенаправить пользователя на страницу для просмотра результата

```
post '/students/new' do
  @errors = []
  first_name = params['first_name']
  @errors.concat(Student.check_data(first_name))
  if @errors.empty?
    settings.students.append(
        Student.new(first_name))
    redirect to('/students')
  else
    erb :stunent_new
  end
end
```