

Mastering Roda

Version: alpha 0.0.1

Federico M. Iachetti Avdi Grimm Jeremy Evans
Denis Borovensky (additions and translation to Russian language)

Содержание

1	Введение	3
1.1	Что такое Roda?	3
1.2	Как читать эту книгу?	5
1.3	Краткое введение в lucid_http	5
2	Ядро Roda	9
2.1	Очень маленький Hello world	9
2.2	Базовая маршрутизация	17
2.3	Методы сопоставления	23
2.3.1	r.on и r.is	23
2.3.2	r.get и r.post	23
2.3.3	r.root	23
2.3.4	Пользовательские методы сопоставления	23
2.3.5	Условные выражения в блоках сопоставления	23
2.3.6	Динамическая маршрутизация	23
2.4	Сопоставители	23
2.4.1	Сопоставители строк	23
2.4.2	Сопоставители классов	23
2.4.3	Логические сопоставители	23
2.4.4	Сопоставители регулярных выражений	23
2.4.5	Сопоставители массивов	23
2.4.6	Сопоставители хэшей	23
2.4.7	Сопоставители символов	23
2.4.8	Сопоставители процедур	23
2.4.9	Сопоставители для чего угодно	23
2.5	Прочие методы RodaRequest	23
2.5.1	r.redirect	23
2.5.2	r.halt	23
2.5.3	r.run	23
2.6	RodaResponse	23
2.7	Область видимости блока route	23
2.8	Класс Roda_	23
2.8.1	app, приложение rack	23

2.8.2	freeze, для предотвращения неожиданных изменений	23
2.8.3	opts, параметры класса и плагина	23
2.8.4	plugin, для загрузки плагинов	23
2.8.5	route, для создания блока маршрута	23
2.8.6	обработка промежуточного программного обеспечения	23

3	Приложение: джем lucid_http	24
----------	------------------------------------	-----------

Раздел 1

Введение

Я являюсь разработчиком **Ruby on Rails** с 2011 года. **Rails** — отличный, но при этом громоздкий и чрезмерно своенравный фреймворк. Он позволяет нам сделать практически все, что угодно, «из коробки» (независимо от того, нужно нам это или нет) без каких-либо дополнительных настроек. Но если мы захотим сделать что-то «нестандартное», то ... окажемся предоставлены сами себе.

Именно поэтому спустя некоторое время разработки на **Rails** я решил, что хочу перейти к более легковесным фреймворкам и использовать более минималистичный подход в разработке. Спустя некоторое время, перепробовав несколько джемов, я остановился на **Roda** — небольшом фреймворке, созданном Джереми Эвансом, который мне очень понравился. Понравился настолько, что я захотел поделиться своими знаниями о том, как его использовать.

1.1 Что такое Roda?

Roda представляет собой набор инструментов для построения древа маршрутизации. В основе философии **Roda** лежат такие принципы, как простота, надежность, расширяемость и производительность. И хотя по умолчанию разработчику доступны лишь самые основные функции, благодаря обширной библиотеке плагинов функционал **Roda** может быть значительно расширен.

Каждый плагин, поставляемый с **Roda**, можно рассматривать как отдельный узкоспециализированный инструмент, который может понадобиться нам в процессе создания веб-приложения, а в зависимости от поставленной задачи, мы можем самостоятельно формировать

необходимый инструментарий. Таким образом **Roda** оказывается скорее ближе к расширяемой библиотеке, нежели к фреймворку, хотя ее и принято называть именно так.

В основе механизма маршрутизации, используемого **Roda**, лежит построение дерева маршрутизации. Как мы увидим в дальнейшем, такой подход обеспечивает **Roda** невероятную гибкость и мощь. Ключевое преимущество дерева маршрутизации заключается в том, что процессы обработки запросов и маршрутизации тесно интегрированы между собой. Возможность обработки запроса непосредственно во время его маршрутизации позволяет устранить дублирование, присущее многим другим веб-фреймворкам, в которых маршрутизация и обработка запросов разделены.

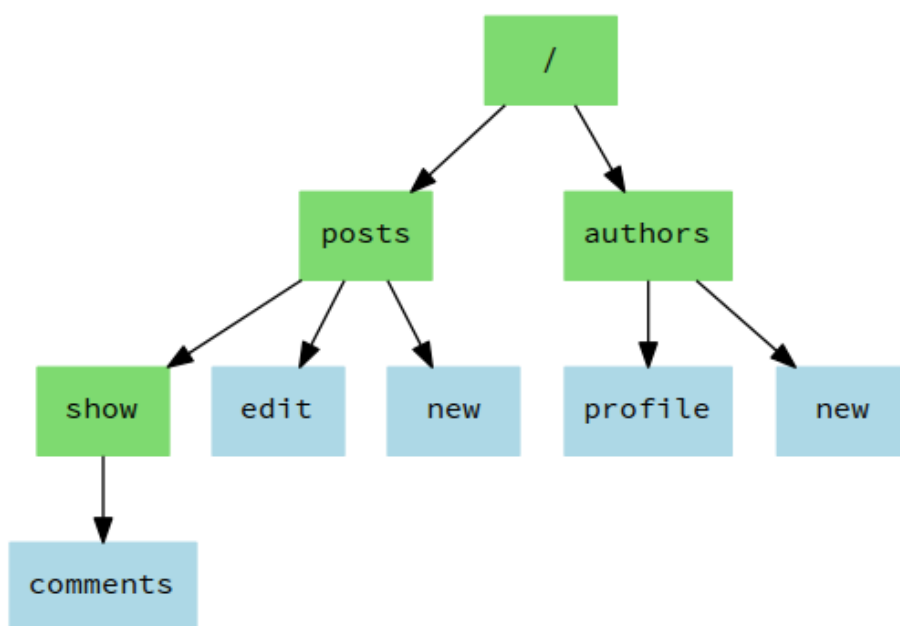


Figure 1.1: Дерево маршрутизации Roda

Roda — чрезвычайно легкая библиотека. Основные ее функции, доступные по умолчанию, реализованы менее, чем в 800 строках кода. Тем не менее, для **Roda** создано свыше 100 готовых плагинов, способных удовлетворить потребности большинства веб-разработчиков.

Roda разрабатывалась с упором на производительность и по праву считается самым быстрым веб-фреймворком **Ruby**. Хотя некоторые оптимизации усложняют понимание кода, большая часть кодовой базы фреймворка остается наглядной и простой для анализа. Благодаря этому любое приложение, созданное с помощью **Roda**, также легко

понять, поскольку вы сможете без труда проследить логику его работы и увидеть, как будет маршрутизироваться и обрабатываться тот или иной запрос.

Roda спроектирована таким образом, чтобы область видимости приложения не засорялась множеством переменных экземпляров, констант и методов, что помогает избежать неожиданных конфликтов имен. Все внутренние переменные экземпляра, которые использует **Roda**, имеют префикс в виде знака подчеркивания, а все константы — префикс **Roda**. При этом в области видимости приложения определены лишь несколько методов.

В этой книге мы рассмотрим базовые концепции и инструменты, предоставляемые **Roda**, а также соглашения и передовой опыт, которые помогут начать работу с этой удивительной библиотекой.

1.2 Как читать эту книгу?

Данная книга целиком и полностью основана на примерах. Каждое новое понятие раскрывается через решение конкретной проблемы.

Вы можете просто прочитать эту книгу от корки до корки. Каждый пример написан так, чтобы вы могли воспроизвести его самостоятельно, и я настоятельно советую вам это сделать. Но я также предлагаю вам пойти дальше и начать экспериментировать с кодом, ведь это — самый лучший способ закрепить представленные концепции в уме.

1.3 Краткое введение в `lucid_http`

Прежде, чем приступить к работе с самой **Roda**, я хотел бы представить вам **`lucid_http`** — джем, который я создал специально для демонстрации взаимодействий по протоколу **HTTP**. Данная библиотека используется во всех примерах для отправки запросов и получения ответов от веб-приложения. Здесь я не буду вдаваться в подробности реализации. За дополнительной информацией, пожалуйста, обратитесь к приложению «[Джем `lucid_http`]»([#prilozhenie-dzhem-lucid-http](#)), приведенному в конце данной книги.

По своей сути **`lucid_http`** представляет собой оболочку для **`http.rb`** — библиотеки Ruby, предоставляющей очень простой и согласованный **API** для выполнения **HTTP**-запросов. **`lucid_http`** предоставляет более высокий уровень абстракции представления, что упрощает работу с **`http.rb`**.

Чтобы продемонстрировать, как работает **`lucid_http`**, я создал небольшой скрипт, код которого размещен в файле [appendix_lucid_http_app.ru](#).

В **приложении** я подробно расскажу, как его запустить.

Начнем с отправки **GET**-запроса к пути **/hello**. Для этого нам нужно вызвать метод `GET`, передав требуемый путь в качестве аргумента (удобнее всего работать с **lucid_http** через консольный интерфейс **irb**):

```
require "lucid_http"

GET "/hello" # => "<h1>Hello World!</h1>"
```

Метод вернет отрендеренное тело ответа, которое отобразится в виде строки после знака комментария `# =>`.

По умолчанию базовый **URL**-адрес, на который **lucid_http** отправляет запросы, — <http://localhost:9292>. Обратите внимание, что в конце адреса отсутствует завершающая косая черта, а значит нам необходимо включить ее в путь, по которому мы хотим послать запрос (в нашем случае **/hello**).

Как мы видим, метод **GET** возвращает тело полученного ответа. Но что на счет другой важной информации? В арсенале **lucid_http** есть и другие методы:

```
require "lucid_http"

GET "/hello/you"
status # => 200 OK
status.to_i # => 200
content_type # => "text/html"
path # => "http://localhost:9292/hello/you"
```

Когда мы делаем следующий запрос, предыдущий ответ удаляется, так что каждый новый запрос начинается с чистого листа:

```
require "lucid_http"

GET "/hello/you"
status # => 200 OK
content_type # => "text/html"
path # => "http://localhost:9292/hello/you"
body[/\>(.*?)\</, 1] # => "Hello, You!"

GET "/403"
status # => 403 Forbidden
content_type # => "text/html"
path # => "http://localhost:9292/403"
body # => "The request returned a 403 status."
```

Мы также можем отслеживать перенаправления, передав атрибут `follow: true`:

```
require "lucid_http"

GET "/redirect_me"
status # => 302 Found

GET "/redirect_me", follow: true
status # => 200 OK
body # => "You have arrived here due to a redirection."
```

Если мы получаем код ошибки **500**, то сможем посмотреть, что произошло, вызвав метод `error`, который вернет первую строку тела ответа, чтобы показать краткое сообщение об ошибке:

```
require "lucid_http"

GET "/500"
status # => 500 Internal Server Error
error # => "SocketError: SocketError"
```

Если же запрос не возвращает код **500**, при вызове метода `error` библиотека любезно сообщит нам об этом:

```
require "lucid_http"

GET "/not_500"
status # => 200 OK
error # => "No 500 error found."
```

Если мы обращаемся к **JSON endpoint**, то строковый вывод окажется не лучшим выбором для отображения полученных данных:

```
require "lucid_http"

GET "/hello_world"
# => "You said: hello_world"

GET "/hello_world.json"
# => "{\"content\":\"You said: hello_world\\\", \"keyword\":\"hello_world\\\", \"timestamp\":\"2020-01-01T00:00:00Z\"}"
```

Однако передав атрибут `json: true`, мы получим данные в виде хэша, на который куда приятнее смотреть. Так-то лучше:

```
require "lucid_http"

GET "/hello_world"
# => "You said: hello_world"
```



```
GET "/hello_world.json", json: true
# => {"content"=>"You said: hello_world",
#     "keyword"=>"hello_world",
#     "timestamp"=>"2016-12-31 15:01:06 -0300",
#     "method"=>"GET",
#     "status"=>200}
```

lucid_http также поддерживает ряд других глаголов **HTTP**, которые мы можем использовать:

```
require "lucid_http"

GET      "/verb"          # => "<GET>"
POST     "/verb"          # => "<POST>"
PUT      "/verb"          # => "<PUT>"
PATCH   "/verb"          # => "<PATCH>"
DELETE   "/verb"          # => "<DELETE>"
OPTIONS  "/verb"          # => "<OPTIONS>"
```

С помощью **lucid_http** мы можем отправить на сервер даже форму, воспользовавшись опцией `:form`:

```
require "lucid_http"

POST "/params?item=book", json: true
# => {"item"=>"book"}

POST "/params", json: true, form: { item: "book", quantity: 1, price: 50.0, title: "The complete guide to doing absolutely nothing at all."}
# => {"item"=>"book",
#     "quantity"=>"1",
#     "price"=>"50.0",
#     "title"=>"The complete guide to doing absolutely nothing at all."}
```

Теперь, когда у вас есть общее представление, как результаты запросов будут отображаться в книге, мы можем приступить к изучению **Roda**.

Раздел 2

Ядро Roda

В этом разделе мы рассмотрим основные классы **Roda**, а также поведение и возможности, которые **Roda** предлагает по умолчанию, иными словами, изучим ядро фреймворка. Однако при обсуждении многих основных функций **Roda** мы также будем упоминать соответствующие плагины.

Для начала мы изучим базовую структуру приложения **Roda**, разберемся, как оно перенаправляет запросы от пользователя к целевому коду, как возвращает соответствующий ответ и как обрабатывает сеансы.

2.1 Очень маленький Hello world

По традиции, начнем с создания Hello world чтобы понять, как вообще выглядит приложение **Roda**. И в первую очередь нам необходимо создать новый проект.

Если вы привыкли работать с **Rails**, то можете подумать, что наверное существует некая команда для генерации нового проекта. Однако напомним, что **Roda** — скорее библиотека, нежели фреймворк, и такой команды у нее попросту нет, так что все придется делать вручную.

Первый шаг — создание нового пустого каталога:

```
mkdir my_app
```

Теперь добавим в директорию проекта **Gemfile** для управления джемами. Очевидно, что нам понадобится сам фреймворк, а также веб-сервер (хорошим выбором для нас станет **Puma**). Кроме

того, практически каждый веб-фреймворк, написанный на **Ruby**, использует в качестве универсального слоя совместимости **Rack**, и **Roda** не является исключением. В свою очередь, джем **Rackup** предоставляет универсальный интерфейс для запуска приложений **Rack** на поддерживаемых серверах (включая **Puma**), поэтому наш стартовый **Gemfile** будет выглядеть так:

```
source "https://rubygems.org"

gem "roda"
gem "puma"
gem "rack"
gem "rackup"
```

Осталось установить выбранные джемы:

```
bundle install
```

Отмечу, что создать приведенный выше **Gemfile** можно непосредственно из консоли, выполнив следующие команды:

```
bundle init
bundle add roda puma rack rackup
```

Команда `bundle init` создает пустой **Gemfile**, содержащий лишь адрес репозитория джемов <https://rubygems.org>, а `bundle add roda puma rack rackup` добавляет необходимые джемы в **Gemfile** и сразу же устанавливает их. Далее в книге мы будем использовать именно эту связку, постепенно добавляя новые джемы. Однако в любом случае начиная с этого момента перечисленные выше джемы будут присутствовать в каждом создаваемом нами **Gemfile**.

Теперь мы готовы писать код! Начнем разработку с создания **rackup-файла**, используя стандартное имя **config.ru**.

В нем мы подключим **Roda**, а затем создадим новый класс для представления нашего приложения. Этот класс будет наследовать базовому классу **Roda**.

Roda построена на идее *древа маршрутизации*, что подразумевает создание *ветвей* путем добавления *маршрутов*. Начнем с определения блока `route`. В качестве аргумента данный блок будет получать запрос, который, по соглашению, мы будем обозначать, как `r`.

Наш первый маршрут проверяет, относится ли полученный **HTTP-запрос** к пути `/hello`. Если это так, соответствующий блок кода будет выполнен и вернет строку `"hello!"`.

Наследуя классу **Roda**, наш класс **App** неявно становится **Rack-приложением**. Чтобы указать **Rack** (и веб-серверу) выполнять наше

приложение для **HTTP-запросов**, мы должны использовать команду run:

```
require "roda"

class App < Roda
  route do |r|
    r.get "hello" do
      "hello!"
    end
  end
end

run App
```

Теперь, чтобы запустить веб-сервер и начать обрабатывать запросы, достаточно выполнить в терминале команду rackup:

```
rackup
```

Если после этого перейти по ссылке <http://localhost:9292/hello>, то мы увидим наше первое сообщение. Ура!

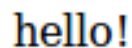


Figure 2.1: Первое сообщение от Roda

Давайте внесем небольшие изменения в наше крошечное приложение. Заменим возвращаемое блоком значение:

```
require "roda"

class App < Roda
  route do |r|
    r.get "hello" do
      "Hello, world!"
    end
  end
end
```

```
end
end
end

run App
```

Если мы снова перейдем по ссылке <http://localhost:9292/hello>, то увидим, что ... ничего не изменилось! Почему? Все потому, что на сервере до сих пор работает наш исходный скрипт.

Вот еще одно свидетельство того, что мы используем не **Rails**, а нечто куда более простое. И если нам нужны навороты вроде автоматической перезагрузки при изменении кода, мы должны самостоятельно добавить их.

Наиболее простым решением для перезагрузки приложения «на лету» при каждом изменении кода является использование джема **Rerun**. Добавим его в наш **Gemfile** в группы `:development` и `:test` (по очевидным причинам в продакшене данный джем будет лишним). Сюда же можно добавить и **lucid_http** (он понадобится нам совсем скоро):

```
group :development, :test do
  gem "rerun"
  gem "lucid_http"
end
```

Выполним установку:

```
bundle install
```

Теперь для запуска нашего приложения можно использовать команду:

```
rerun rackup
```

Перейдем на <http://localhost:9292/hello> и убедимся, что вывод обновляется без перезапуска приложения:




Figure 2.2: Hello world!

В дальнейшем мы будем постоянно использовать `rerun rackup`, чтобы избежать необходимости постоянного перезапуска сервера вручную.

Итак, у нас есть работающее веб-приложение, однако мы бы хотели иметь возможность взаимодействовать с ним. Давайте предоставим пользователю возможность указать в приветствии собственное имя. Для этого создадим второй маршрут, указав в качестве нового сопоставителя класс **String**, благодаря чему любое строковое значение (например, **/hello/Federico** или **/hello/Denis**) будет считаться валидным.

Когда мы указываем в качестве сопоставителя класс **String**, в блок кода передается соответствующий строковый аргумент. Давайте дадим ему подходящее имя, а затем интерполируем в вывод:

```
require "roda"

class App < Roda
  route do |r|
    r.get "hello", String do |name|
      "<h1>Hello #{name}!</h1>"
    end
  end
end

run App
```

Если теперь мы перейдем на <http://localhost:9292/hello/Roda>, то увидим, что строка Roda берется из пути и отображается на странице.



Hello Roda!

Figure 2.3: Hello Roda!

Проведем небольшой рефакторинг. До сих пор мы работали с **rack-файлом config.ru**, однако это является не самым лучшим подходом. Как можно догадаться из названия, данный файл предназначен лишь

для конфигурации параметров запуска сервера приложений, а не для хранения кода всего приложения.

Давайте удалим класс **App** вместе с оператором **require** и перенесем код в отдельный файл **app.rb**:

```
require "roda"

class App < Roda
  route do |r|
    r.get "hello", String do |name|
      "<h1>Hello #{name}!<h1>"
    end
  end
end
```

Подключим приложение в **config.ru**:

```
require "./app"

run App
```

Убедимся, что код все еще работает:



Hello Roda!

Figure 2.4: Hello Roda!

Вот оно, наше первое приложение Roda!

Давайте теперь представим, что мы создаем сайт, который рассказывает посетителям о некоем таинственном госте. Пользователь не знает, кто это такой, однако стоит ему перейти по маршруту **/mystery_guest**, как на экране отобразится имя загадочного гостя.

По чрезвычайно веским причинам, истоки которых выходят за рамки этой книги, таинственным гостем будет пицца с моцареллой. Для создания класса **Pizza** мы воспользуемся классом **Struct** стандартной библиотеки Ruby, а затем создадим экземпляр этого класса, чтобы использовать его в маршруте **mystery_guest**. Just for lulz, я допустил опечатку в слове Guest:

```
Pizza = Struct.new(:flavor)

class App < Roda
  mystery_guest = Pizza.new("Mozzarella")

  route do |r|
    r.get 'mystery_guest' do
      "The Mystery Gest is: #{mystery_guest}"
    end
  end
end
```

Что-то явно пошло не так:

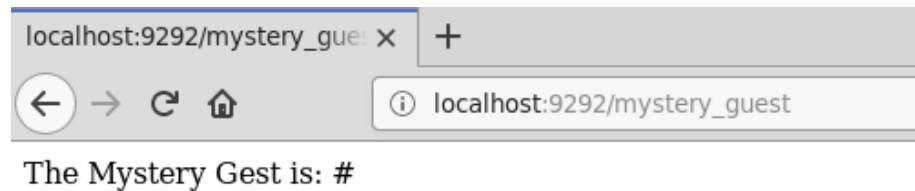


Figure 2.5: Hello Roda!

Но что же произошло? Если мы воспользуемся **lucid_http**, то убедимся, что приложение возвращает верный результат:

```
require "lucid_http"

GET "/mystery_guest"
# => "#<struct Pizza flavor=\"Mozzarella\">"
```

Но почему мы не видим этого в браузере? Давайте рассуждать. Вывод ломается сразу после знака интерполяции **#**. За ним в фигурных

скобках следует строка `<struct Pizza flavor="Mozzarella">`, очень похожая на **HTML-тег**. И в этом проблема: такого **HTML-тега** не существует, и браузер не может корректно распознать строку. В противном случае все работало бы идеально, в чем мы можем убедиться, заключив вывод в тег `h1` (строка отобразится, как заголовок первого уровня):

```
require "roda"

Pizza = Struct.new(:flavor)

class App < Roda
  mystery_guest = Pizza.new("Mozzarella")

  route do |r|
    r.get 'mystery_guest' do
      "<h1>The Mystery Gest is: #{mystery_guest}</h1>"
    end
  end
end
```

Как решить данную проблему? Самым очевидным решением было бы воспользоваться методом **to_s** в нашем классе **Pizza**, и все стало бы на свои места. Однако данный подход является сколь наивным, столь и непрактичным. На то есть две важные причины: - Бывают ситуации, когда нам действительно необходимо отобразить значение переменной или результат отработки того или иного метода, который на самом деле возвращает класс, подобный нашему классу **Pizza**, чего мы не ожидаем. В этом случае мы не сможем точно сказать, где именно необходимо поместить метод **to_s**, так как не уверены, с чем имеем дело. - Часто нам требуется визуализировать **HTML-код**, который браузер по очевидным причинам всегда интерпретирует, как есть.

В обоих случаях решение заключается в том, чтобы экранировать вывод. В **Roda** для этого предусмотрен плагин `:h`. После его загрузки мы можем передать строку, которую требуется обработать, в одноименный метод `h`:

```
require "roda"

Pizza = Struct.new(:flavor)

class App < Roda
  plugin :h

  mystery_guest = Pizza.new("Mozzarella")
```

```

route do |r|
  r.get 'mystery_guest' do
    "The Mystery Gest is: #{h mystery_guest}"
  end
end
end
end

```

Теперь ответ будет правильно экранирован:

```

require "lucid_http"

GET "/mystery_guest"
# => "The Mystery Gest is: #<struct Pizza flavor="Mozzarella">"

```

Конечно необработанный **HTML** выглядит довольно уродливо, зато в браузере результат отображается корректно:



Figure 2.6: Результат выводится кооректно благодаря плагину h

2.2 Базовая маршрутизация

На базовом уровне протокол HTTP можно рассматривать как набор запросов и ответов. И если бы нам нужно было отвечать только на один тип запросов, мы могли бы справиться с этой задачей с помощью одного большого фрагмента кода. Но на практике дела обстоят несколько сложнее.

Современные веб-приложения обычно поддерживают более одного типа запросов. И чтобы не сойти с ума, нам, как разработчикам, приходится отделять части кода, которые обрабатывают разные типы запросов, друг от друга. Также мы стараемся избегать дублирования

фрагментов кода, используемых для обработки запросов разного типа.

Получение запроса от клиента и перенаправление его к фрагменту кода, который обрабатывает соответствующий запрос, называется *маршрутизацией*, и в этом разделе книги мы посмотрим, как она работает в **Roda**.

Как мы видели в предыдущих примерах, первым шагом в процессе разработки приложения **Roda** является создание класса, который наследует от `Roda`. Затем мы вызываем в классе нашего приложения метод `route` класса `Roda`. Мы передаем в него блок кода, который принимает в качестве параметра объект, представляющий собой текущий запрос.

Давайте посмотрим, что представляет собой содержимое параметра `r` с помощью `Kernel`-метода `p`. Раз уж мы используем `p` только для отладки, то не будем делать стандартный отступ, чтобы в дальнейшем его было легче найти и удалить из нашего скрипта. Кроме того, чтобы не загромождать вывод сообщениями об ошибках, мы также будем возвращать пустую строку (немного позже я объясню, почему это необходимо).

```
require "roda"

class App < Roda
  route do |r|
p r
    ""
  end
end
```

Когда мы переходим по любому пути нашего приложения то видим, что в наш блок передается экземпляр `App::RodaRequest`:

```
#<App::RodaRequest GET />
127.0.0.1 - - [19/Sep/2016:19:46:26 -0300] "GET / HTTP/1.1" 404 - 0.0016
```

`Roda::RodaRequest` является подклассом класса `Rack::Request`, к которому добавлены методы для обработки маршрутизации. В свою очередь, `App::RodaRequest` является подклассом `Roda::RodaRequest`, который создается автоматически при создании класса `App` и позволяет тонко настраивать обработку экземпляров запроса, в том числе с помощью плагинов.

Из вывода `p` видно, что запрос является **GET-запросом** к пути `/`. Для обработки маршрутизации мы можем вызывать методы объекта запроса `r`, которые будем изучать в следующих нескольких разделах. А пока давайте начнем с определения первого маршрута.

Мы вызываем метод `on` для нашего объекта запроса, передавая ему строковый параметр и блок. Из блока мы возвращаем строку:

```
class App < Roda
  route do |r|
    r.on "hello" do
      "Hello Lucid!"
    end
  end
end
```

Метод `r.on` относится к группе *методов сопоставления*, каждый из которых принимает аргументы, называемые *сопоставителями*, и проверяет, соответствуют ли *сопоставители* запросу. Если сопоставление успешно, *метод сопоставления* выполняет соответствующий блок, называемый *блоком сопоставления*, и обработка запроса завершается, как только код *блока сопоставления* будет выполнен. Если же сопоставление не удалось, код блока не выполняется, а *метод сопоставления* завершает работу.

Метод `r.on` является самым простым *методом сопоставления*. Он лишь проверяет, соответствуют ли *сопоставители* запросу, не выполняя каких-либо дополнительных проверок. Давайте посмотрим, что же произошло в примере, приведенном выше.

Если *метод сопоставления* передает управление блоку сопоставления, а *блок сопоставления*, в свою очередь, возвращает строку, эта строка используется в качестве тела ответа. Мы также можем убедиться, что код состояния ответа — 200, а тип контента — `text/html`.

```
require "lucid_http"

GET "/hello"

body           # => "Hello Lucid!"
status         # => "200 OK"
content_type   # => "text/html"
```

Если блок возвращает `nil` или `false`,

```
class App < Roda
  route do |r|
    r.on "hello" do
      nil
    end
  end
end
```

Roda интерпретирует это, как необработанный маршрут, возвращая

пустое тело с кодом ответа 404 (страница не найдена):

```
require "lucid_http"

GET "/hello"

body          # => ""
status        # => "404 Not Found"
```

Если же мы попробуем вернуть нечто, с чем **Roda** не знает, как правильно обращаться (например, Integer):

```
class App < Roda
  route do |r|
    r.on "hello" do
      1
    end
  end
end
```

Roda вызовет исключение `Roda::RodaError`, которое большинство веб-серверов будет интерпретировать, как внутреннюю ошибку сервера, возвращая соответствующее тело:

```
require "lucid_http"

GET "/hello"

body          # => "..."
status        # => "500 Internal Server Error"
```

По умолчанию в **Roda** в качестве возвращаемых значений *блоков сопоставления* поддерживаются только `String`, `nil` и `false`. Однако **Roda** поставляется с плагинами, которые поддерживают дополнительные возвращаемые значения для *блоков сопоставления*, которые мы обсудим позже.

На этом история не заканчивается. Если мы внимательно рассмотрим наш предыдущий пример, то убедимся, что у нас имеются два блока кода: `route`, принимающий запрос, и `r.on`, генерирующий контент. Давайте поместим операторы отладки `p` внутрь каждого из них:

```
require "roda"

class App < Roda
  route do |r|
    p "ROUTE block"

    r.on "hello" do
```

```
p "HELLO block"

    "Hello Lucid!"
end
end
end
```

Если теперь мы перейдем по адресу <http://localhost:9292/>, то увидим, что была напечатана только строка, соответствующая блоку `route`.

```
"ROUTE block"
127.0.0.1 - - [16/Nov/2016:12:56:22 -0300] "GET / HTTP/1.1" 404 - 0.0020
```

Когда мы перейдем на <http://localhost:9292/hello>, то будут напечатаны уже обе строки:

```
"ROUTE block"
"HELLO block"
127.0.0.1 - - [16/Nov/2016:12:58:35 -0300] "GET /hello HTTP/1.1" 200 5 0.0011
```


2.3 Методы сопоставления

2.3.1 r.on и r.is

2.3.2 r.get и r.post

2.3.3 r.root

2.3.4 Пользовательские методы сопоставления

2.3.5 Условные выражения в блоках сопоставления

2.3.6 Динамическая маршрутизация

2.4 Сопоставители

2.4.1 Сопоставители строк

2.4.2 Сопоставители классов

2.4.3 Логические сопоставители

2.4.4 Сопоставители регулярных выражений

2.4.5 Сопоставители массивов

2.4.6 Сопоставители хэшей

2.4.7 Сопоставители символов

2.4.8 Сопоставители процедур

2.4.9 Сопоставители для чего угодно

2.5 Прочие методы RodaRequest

2.5.1 r.redirect

2.5.2 r.halt

2.5.3 r.run

2.6 RodaResponse

2.7 Область видимости блока route

2.8 Класс Roda_

2.8.1 app, приложение rack

2.8.2 freeze, для предотвращения неожиданных изменений

2.8.3 opts, параметры класса и плагина

2.8.4 plugin, для загрузки плагинов

Раздел 3

Приложение: джем lucid_http

В ходе создания «Mastering Roda» мне понадобился эффективный и наглядный способ продемонстрировать читателям взаимодействие между пользователем и сервером. Мне не хотелось полагаться на браузер и засорять книгу лишними скриншотами: вместо этого я хотел иметь возможность получать максимум информации непосредственно через терминал.

Первое, что пришло мне в голову — воспользоваться **http.rb**. И тут я встал перед дилеммой. С одной стороны **http.rb** оказалась именно тем, что мне было нужно: эта библиотека отлично подходит для создания HTTP-запросов, оказываясь достаточно мощной и гибкой для того, чтобы наглядно показать, как именно **Roda** обрабатывает запросы. Проблема в том, что **http.rb** создавалась, в первую очередь, для *выполнения* запросов, а не *отображения* ответов в удобоваримой форме.

Вот как будет выглядеть код, если мы захотим отправить простой **GET-запрос**:

```
require "http"

res = HTTP.get("http://localhost:9292/hello")
res.body.to_s      # => "<h1>Hello World!</h1>"
res.status.to_s    # => "200 OK"
```

Слишком многословно и громоздко.

Именно поэтому я решил разработать собственный **DSL**, который в

итоге воплотился в небольшую библиотеку — **lucid_http**. Данный репозиторий содержит приложение [appendix_lucid_http_app.ru](#), которое поможет наглядно продемонстрировать все возможности **lucid_http**. Чтобы его запустить, достаточно выполнить команду `rackup`:

```
rackup appendix_lucid_http_app.ru
```

Эта команда запустит веб-сервер и наше приложение для экспериментов с **lucid_http**. Как только вы закончите, можете просто нажать **Ctrl+C** в терминале, чтобы его остановить.

Теперь мы можем приступить к изучению джема **lucid_http**. Начнем с открытия оболочки терминала **irb** и загрузки библиотеки:

```
irb -r lucid_http
```

Допустим, мы хотим сделать **GET-запрос** к пути **hello**. Для этого достаточно вызвать одноименный метод и передать ему в качестве аргумента требуемый путь:

```
GET "/hello"  
# => "<h1>Hello World!<h1>"
```

По умолчанию **lucid_http** отправляет запрос на URL-адрес <http://localhost:9292>, в чем мы можем убедиться, вызвав метод `target_url`:

```
LucidHttp.target_url  
# => "http://localhost:9292"
```

Мы можем изменить его, и с этого момента каждый новый запрос в рамках сессии будет использовать новый **URL**:

```
LucidHttp.target_url  
# => "http://localhost:9292"  
  
LucidHttp.target_url = "http://lucid_code.org"  
  
LucidHttp.target_url  
# => "http://lucid_code.org"
```

Или же, чтобы каждый раз не устанавливать `target_url` вручную, мы можем просто задать переменную окружения `TARGET_URL`:

```
ENV["TARGET_URL"] = "http://lucid_code.org"  
  
LucidHttp.target_url  
# => "http://lucid_code.org"
```

Вы могли обратить внимание, что перед именем пути стоит косая черта. Все дело в том, что **lucid_http** принимает целевой **URL** как есть,

оставляее неизменным. Таким образом, если мы уберем косую черту, строка `hello` добавится к номеру порта, что закономерно вызовет ошибку:

```
GET "/hello"
# => "<h1>Hello World!<h1>"

GET "hello" # ~> Addressable::URI::InvalidURIError:..."
```

Вызывая метод `GET`, мы возвращаем тело нашего ответа. Но что на счет другой важной информации? **lucid_http** предоставляет ряд методов, позволяющих ее получить. Например, мы можем увидеть код состояния, тип контента и полный путь, а метод `body` вернет отрэндереное тело ответа в том случае, если нам требуется каким-то образом с ним манипулировать:

```
GET "/hello/you"
status                # => 200 OK
status.to_i           # => 200
content_type          # => "text/html"
path                  # => "http://localhost:9292/hello/you"
body[/\>(.)\</, 1]    # => "Hello, You!"
```

Когда мы делаем следующий запрос, предыдущий ответ удаляется, так что каждый новый запрос начинается с чистого листа:

```
GET "/hello/you"
status                # => 200 OK
content_type          # => "text/html"
path                  # => "http://localhost:9292/hello/you"
body[/\>(.)\</, 1]    # => "Hello, You!"

GET "/403"
status                # => 403 Forbidden
content_type          # => "text/html"
path                  # => "http://localhost:9292/403"
body                  # => "The request returned a 403 status."
```

Рассмотрим другой пример. Если мы сделаем запрос к `/over_there`, то получим статус 302. Это значит, что мы были перенаправлены на другой адрес. Если теперь мы посмотрим на тело ответа, то увидим, что оно пусто:

```
GET "/over_there"
status                # => 302 Found
body                  # => ""
```

Однако если мы откроем ссылку в браузере, то получим иной результат:

You have arrived here due to a redirection.

Figure 3.1: Работа с редиректами

Все потому, что по умолчанию метод GET не обрабатывает редиректы. Мы можем изменить данное поведение, передав аргумент `follow: true`:

```
GET "/over_there", follow: true
status      # => 200 OK
body        # => "You have arrived here due to a redirection."
```

До этого момента мы имели дело только с «успешными» запросами. Но что произойдет, если мы отправим запрос по маршруту, в котором присутствует какая-то ошибка? Мы получим код состояния 500, что вполне ожидаемо. Однако в теле ответа мы увидим длинную уродливую строку со всей обратной трассировкой:

```
GET "/500"
status      # => 500 Internal Server Error
body        # => "ArgumentError: wrong number of arguments (given 0, ..."
```

Это не особо удобно. К счастью для таких случаев в арсенале **lucid_http** имеется метод `error`, который выводит лишь первую обратной трассировки:

```
GET "/500"
status      # => 500 Internal Server Error
error       # => "SocketError: SocketError"
```

Так-то лучше.

Полученный в данном примере ответ обратной трассировки создается **Rack::ShowExceptions**, используемый в development-среде **rackup** по умолчанию. В продакшене при возврате исключения вывод будет зависеть от используемого веб-сервера.

Если запрос не возвращает ошибок, **lucid_http** любезно сообщим об этом:

```
GET "/not_500"
status      # => 200 OK
```

```
error # => "No 500 error found."
```

Теперь предположим, что у нас есть пара **URL-адресов**, один из которых возвращает ответ в формате **HTML**, а другой — в формате **JSON**. Вот, что мы увидим:

```
GET "/hello_world"
# => "You said: hello_world"

GET "/hello_world.json"
# => "{\"content\":\"You said: hello_world\" + \
#     \",\"keyword\":\"hello_world\" + \
#     \",\"timestamp\":\"2016-12-31 15:00:42 -0300\" + \
#     \",\"method\":\"GET\" + \
#     \",\"status\":200}"
```

По умолчанию **lucid_http** показывает необработанные данные **JSON**, которые не очень-то удобно читать. Однако передав методу GET атрибут `json: true`, мы получим данные в виде хэша, на который куда приятнее смотреть:

```
GET "/hello_world.json", json: true
# => {"content"=>"You said: hello_world",
#     "keyword"=>"hello_world",
#     "timestamp"=>"2016-12-31 15:01:06 -0300",
#     "method"=>"GET",
#     "status"=>200}
```

lucid_http также поддерживает ряд других глаголов HTTP, которые мы можем использовать:

```
GET      "/verb"      # => "<GET>"
POST     "/verb"      # => "<POST>"
PUT      "/verb"      # => "<PUT>"
PATCH   "/verb"      # => "<PATCH>"
DELETE   "/verb"      # => "<DELETE>"
OPTIONS  "/verb"      # => "<OPTIONS>"
```

Наконец, предположим, что мы хотим отправить на сервер некие данные, используя **POST-запрос**. Это также можно сделать при помощи одноименного метода:

```
POST "/receive?item=book"
# => "{\"item\":\"book\"}"
```

Чтобы отправить форму, достаточно воспользоваться аргументом `form`, передав ему в качестве значения хэш, содержащий необходимые данные:

```
POST "/receive", json: true, form: {  
  item: "book",  
  quantity: 1,  
  price: 50.0,  
  title: "The complete guide to doing absolutely nothing at all."  
}  
# => {"item"=>"book",  
#   "quantity"=>"1",  
#   "price"=>"50.0",  
#   "title"=>"The complete guide to doing absolutely nothing at all."}
```

Вот как работает **lucid_http**. Разумеется, данный джем вовсе не является полнофункциональной **HTTP-библиотекой**: его возможности весьма ограничены. Тем не менее он предоставляет вполне достаточный набор инструментов, которые могут понадобиться нам во время изучения **Roda**.