GeekBrains

Основы Ruby on Rails. Часть I

Утилиты
Генераторы
Логирование

На этом уроке

1. Познакомимся с командой rails и ее возможностями
2. Изучим команды для быстрой разработки Ruby on Rails приложения
3. Вспомним гем rake и попробуем создать собственные rake-задачи
4. Научимся использовать генераторы для создания кода
5. Разберемся с логированием в Ruby on Rails

Оглавление

Быстрая разработка

Успех Ruby on Rails заключается в быстрой разработке веб-приложений. Достигнуть высокой скорости помогают несколько инструментов генерации кода и отладки приложения:

- вспомогательные утилиты которые сосредоточены в папке bin любого Ruby on Rails приложения;
- инструменты автосоздания кода: генераторы;
- удобная система логирования.

На этом уроке мы сосредоточимся на утилитах Rails и генераторах. Утилитой rails мы уже пользовались при создании приложения на предыдущих уроках (команда rails new). Давайте рассмотрим ее более детально.

Утилита rails

Утилиту rails мы использовали для создания Ruby on Rails приложения. Для этого мы передавали ей параметр new

```
$ rails new
```

Помимо new, утилита rails может принимать еще несколько параметров, которые представлены в следующей таблице. У многих из этих команд имеются сокращенные формы, приведенные под основной командой.

Команда	Описание
rails new	Создание нового приложения
rails server rails s	Запуск приложения и веб-сервера
rails console rails c	Запуск rails-консоли, в которой можно запускать код на выполнение, точно так же, как в интерактивном Ruby (irb). Однако, в отличие от irb в консоли доступны все классы и модули Ruby on Rails и веб-приложения
rails dbconsole rails db	Запуск интерактивной консоли базы данных
rails generate rails g	Генераторы для быстрого создания кода
rails destroy rails d	Откат созданного генератором кода
rails runner rails r	Запуск произвольного Rails кода из командной строки
rails test	Запуск тестов

Помимо основных команд, приведенных в таблице выше, поддерживается большое количество других команд, список которых можно получить вызовом команды rails без параметров.

```
$ rails
The most common rails commands are:
 generate    Generate new code (short-cut alias: "g")
 console     Start the Rails console (short-cut alias: "c")
 server      Start the Rails server (short-cut alias: "s")
 test        Run tests except system tests (short-cut alias: "t")
 test:system  Run system tests
 dbconsole   Start a console for the database specified in config/database.yml
             (short-cut alias: "db")

 new         Create a new Rails application. "rails new my_app" creates a
             new application called MyApp in "./my_app"


All commands can be run with -h (or --help) for more information.
In addition to those commands, there are:

  about
  action_mailbox:ingress:exim
  ...
  tmp:create
  yarn:install
```

Эти команды немного отличаются по природе. Условно их можно разделить на две группы:

- rails-задачи генерации кода, запуска приложения и консоли (представлены в таблице);
- rake-задачи по обслуживанию приложения (в выводе rails-утилиты представлены в списке после фразы "In addition to those commands, there are").

С rake-задачами мы уже знакомились в начале изучения языка программирования Ruby, они создаются на языке Ruby и в течение урока мы вспомним как их создавать. rails-задачи выполняет утилита rails и расширить этот список своими собственными задачами не получится.

В предыдущих версиях Ruby on Rails задачи первой группы запускались утилитой rails, второй — rake. В настоящий момент все задачи можно выполнить при помощи rails. Список команд, который выдается командой rails, не является постоянным: команды могут быть добавлены гемами и самим разработчиком. Поэтому начинать знакомство с проектом следует со списка доступных команд.

В справке, которая выводит команда rails, Rake-задачи выводятся без описания. Однако, если мы воспользуемся командами rake --task или rails -T, можно получить список Rake-задач с подробным описанием:

```
$ rails -T
rails about               # List versions of all Rails frameworks
rails active_storage:install # Copy over the migration needed to the app
```

```
...
rails tmp:create          # Creates tmp directories for cache, sockets
rails yarn:install        # Install all JavaScript dependencies
```

Можно использовать как готовые Rake-задачи, так и создавать свои собственные. Например, среди готовых задач можно обнаружить rake-задачу about, при помощи которой можно получить информацию о текущем проекте. Еще более детальную информацию можно получить при помощи rake-задачи stats. С большинством остальных rake-задач, мы познакомимся по мере изучения фреймворка.

```
$ rails about
About your application's environment
Rails version       6.1.3.1
Ruby version        ruby 3.0.0p0 (2020-12-25 revision 95aff21468) [x86_64-darwin18]
RubyGems version    3.2.3
Rack version        2.2.3
Middleware          Webpacker::DevServerProxy, Rack::MiniProfiler,
ActionDispatch::HostAuthorization, Rack::Sendfile, ActionDispatch::Static,
ActionDispatch::Executor, ActiveSupport::Cache::Strategy::LocalCache::Middleware,
Rack::Runtime, Rack::MethodOverride, ActionDispatch::RequestId,
ActionDispatch::RemoteIp, Sprockets::Rails::QuietAssets, Rails::Rack::Logger,
ActionDispatch::ShowExceptions, WebConsole::Middleware,
ActionDispatch::DebugExceptions, ActionDispatch::ActionableExceptions,
ActionDispatch::Reloader, ActionDispatch::Callbacks,
ActiveRecord::Migration::CheckPending, ActionDispatch::Cookies,
ActionDispatch::Session::CookieStore, ActionDispatch::Flash,
ActionDispatch::ContentSecurityPolicy::Middleware,
ActionDispatch::PermissionsPolicy::Middleware, Rack::Head, Rack::ConditionalGet,
Rack::ETag, Rack::TempfileReaper
Application root    /home/user/app
Environment         development
```

Создание приложения. rails new

Создание нового Ruby on Rails приложения осуществляется при помощи команды rails new.

```
$ rails new todo
```

По умолчанию создается приложение, в котором в качестве базы данных выступает SQLite, фронт-часть представлена ассетами и гемом webpacker, подключена возможность использования веб-сокетов при помощи подсисистемы ActionCable, а файлы сохраняются с использованием ActionStorage. Со всеми этими компонентами мы познакомимся позже по мере изучения Ruby on Rails. Однако уже сейчас мы вынуждены создавать приложения и часть этих компонентов можно отключить, так как мы еще не скоро начнем работу с ними.

Если вы не хотите использовать какой-то компонент в своем приложении, его можно отключить. Для конфигурирования приложения используются skip-параметры. Ниже приводится команда для создания минимального Ruby on Rails приложения, которое предоставляет только API для внешнего фронт-приложения.

```
$ rails new todo --database=postgresql --skip-yarn --skip-action-mailer
--skip-active-storage --skip-action-cable --skip-sprockets --skip-spring
--skip-coffee --skip-javascript --skip-turbolinks --skip-test --no-rc --api
```

Детальный список параметров и их назначение можно найти в документации на странице https://guides.rubyonrails.org/command_line.html. На ближайших уроках мы будем пользоваться именно такой формой создания нового приложения.

Консоль. rails console

Утилита rails может принимать параметр console, которая запускает сессию интерактивного Ruby, в котором подгружены все компоненты Ruby on Rails приложения.

```
$ rails console
```

Чаще параметр console сокращают просто до c:

```
$ rails c
```

Рельсовая консоль — это один из основных инструментов для разработчика. В ней проводятся эксперименты, отрабатываются гипотезы, отлаживается работа приложения. Для того, чтобы испытать рельсовую консоль, давайте определим версию Ruby on Rails, обратившись к методу version класса Rails.

```
$ rails c
Loading development environment (Rails 6.1.3.2)
irb(main):001:0> Rails.version
=> "6.1.3.2"
irb(main):002:0>
```

Создание собственной rake-задачи

Можно легко создать собственную rake-задачу. Для этого в папке lib/tasks необходимо создать файл с расширением rake. Создадим rake-задачу test:hello, которая будет выводить в консоль строку ‘Hello’.

Внимание! *При работе в чистом Ruby для создания rake-задач мы пользовались конфигурационным файлом Rakefile. В Ruby on Rails он тоже есть и его можно обнаружить в корневой папке. Однако, к*

редактированию этого файла прибегают очень редко, собственные rake-задачи следует размещать в папке lib/tasks.

lib/tasks/test.rake

```
namespace :test do
  desc 'Say hello'
  task :hello do
    puts 'Hello'
  end
end
```

rake-задачи создаются на языке Ruby, но оформляются таким образом, чтобы максимально выглядеть в декларативном стиле.

Декларативный стиль программирования. *При декларативном подходе мы описываем, что мы хотим получить, а не как. До текущего момента мы в основном работали в императивном стиле программирования: детально описывали инструкции для компьютера, которые приводят к желаемой цели. Декларативный стиль программирования: это описание цели, которой должен достичь компьютер, при этом способ достижения этой цели программная библиотека выбирает сама. Примерами декларативного стиля разработки могут служить язык разметки HTML и каскадные таблицы стилей CSS.*

Пример выше можно было бы переписать с интенсивным использованием круглых скобок. Однако так при разработке Ruby on Rails не поступают.

lib/tasks/wrong.rake

```
namespace(:test) do
  desc('Say hello')
  task(:hello) do
    puts 'Hello'
  end
end
```

Таким образом namespace, desc и task — это методы, которые могут принимать аргументы и блоки.

Теперь, если выполнить команду rails -T, в списке задач появится созданная rake-задача с описанием ‘Say hello’:

```
$ rails -T
rails about                             # List versions of all Rails frameworks and
the environment
...
rake test:hello                         # Say hello
...
rails yarn:install                      # Install all JavaScript dependencies as
specified via Yarn
```

Можно выполнить задачу:

```
$ rails test:hello
Hello
```

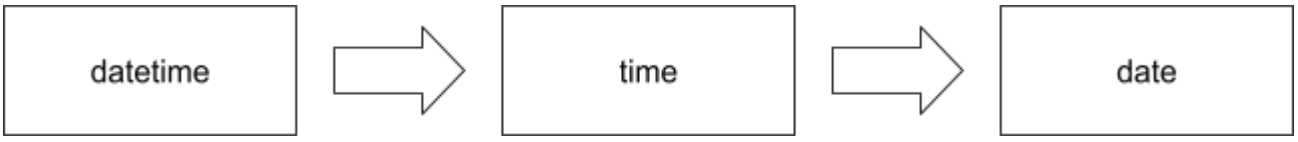
Метод namespace определяет имя пространства имен. Пространства имен можно вкладывать друг в друга, добиваясь глубокой вложенности, например db:migrate:status. Задача определяется методом task, которому может предшествовать необязательное объявление описания при помощи метода desc. Имена пространств имен и задач задаются при помощи символов языка Ruby.

Зависимые rake-задачи

rake-задачи можно выстраивать в цепочки, в которой одна задача зависит от другой. Таким образом вызов одной задачи приводит к автоматическому вызову всех остальных задач. Давайте продемонстрируем это на примере. Создадим три rake-задачи:

- date — вывод текущей даты;
- time — вывод текущего времени;
- datetime — задача, которая выводит в стандартный поток фразу "Вычисление текущих даты и времени...".

При этом задачу time сделаем зависимой от date, а datetime — зависимой от time.



Таким образом вызов задачи datetime должен приводить к автоматическому вызову задач date и time. В папке lib/tasks создадим первую задачу date, разместив ее в файле date.rake.

lib/tasks/date.rake

```
desc 'Вывод текущей даты'
task :date do
  puts Date.today
end
```

Запуск задачи на выполнение приводит к выводу текущей даты в стандартный поток вывода:

```
$ rails date
2021-05-22
```

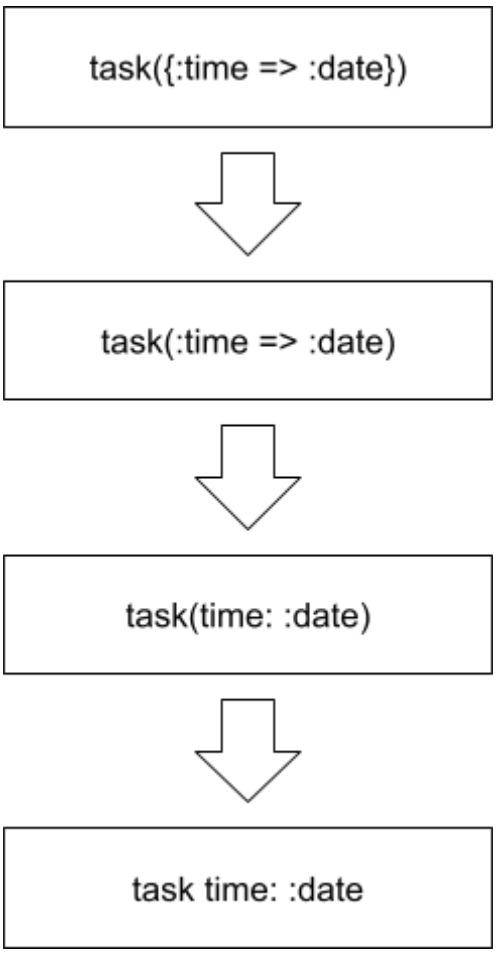
Теперь можно создать задачу time, для вывода текущего времени:

lib/tasks/time.rake

```
desc 'Вывод текущего времени'
task time: :date do
  puts Time.now.strftime('%T')
end
```

Для того, чтобы задача зависела от rake-задачи date, мы передаем методу task, хэш, в котором ключом выступает имя новой rake-задачи :time, а значением — уже существующая rake-задача date. Ниже представлена логика сокращения записи task time: :date в соответствии с соглашениями Ruby:

- У последнего хэша-аргумента метода можно не указывать фигурные скобки;
- Если ключом хэша выступает символ, можно не использовать символ =>, вместо этого двоеточие указывается не в начале, а в конце символа.
- При вызове метода круглые скобки можно опускать.



Если вызывать метод date на выполнение, сначала выполнится метод time и лишь затем метод date.

```
$ rails date
2021-05-22
15:34:33
```

Теперь приступаем к третьей rake-задаче datetime, которая выводит фразу "Вычисление текущих даты и времени...".

lib/tasks/datetime.rake

```
desc 'Вывод текущих даты и времени'
task datetime: :time do
  puts "Вычисление текущих даты и времени..."
end
```

Запуск задачи на выполнение приведет к запуску всех трех задач: сначала :time, потом :date и в конце :datetime.

```
$ rails datetime
2021-05-22
15:34:33
Вычисление текущих даты и времени...
```

В качестве зависимости rake-задаче можно указывать не только своим собственные задачи, но и те, которые предоставляет Ruby on Rails. Очень часто в качестве зависимости указывается rake-задача environment, которая погружает все классы Rails-приложения. Без нее не получится обратиться к классам вашего приложения. Ниже приводится пример rake-задачи :ver, в которой предпринимается попытка вывести версию Ruby on Rails. В ней демонстрируется зависимость задачи от внутренней rake-задачи :environment. Именно в таком виде вы будете видеть большинство rake-задач в реальных Ruby on Rails приложениях.

lib/tasks/version.rake

```
desc 'Вывод версии Ruby on Rails'
task ver: :environment do
  puts Rails.version
end
```

Если rake-задача должна зависеть от нескольких других, то в вместо одного символа с именем задачи передается массив:

lib/tasks/version.rake

```
desc 'Вывод версии Ruby on Rails'
task ver: [:environment, :datetime] do
  puts Rails.version
end
```

Вызов rake-задач из ruby-кода

До этого момента мы вызывали rake-задачи вручную, тем не менее их можно вызывать из кода ваших классов. Очень часто к этому приему прибегают, когда необходимо периодическое выполнение задач по cron-заданию в определенное время (например, раз в сутки) или фоновое выполнение задания по какому-то внешнему событию (нажатие кнопки).

Для вызова rake-задачи из ruby-кода предусмотрен класс Rake::Task которому в квадратных скобках передается название rake-задачи.

Для запуска rake-задачи предусмотрено два метода:

- invoke — вызывает rake-задачу со всеми зависимыми rake-задачами, но выполняет ее ровно один раз;
- execute — вызывает rake-задачу без предварительного вызова зависимостей, однако может выполнить ее несколько раз подряд;

Попробуем написать rake задачу, которая 3 раза выполняет ранее созданную задачу :datetime.

lib/tasks/three_datetime.rake

```
desc 'Вывод серии измерений даты и времени'
task :three_datetime do
  3.times do
    Rake::Task['datetime'].invoke
  end
end
```

Запуск задачи из примера выше приведет к однократному выполнению задачи :datetime

```
$ rails three_datetime
2021-05-22
16:19:45
Вычисление текущих даты и времени...
```

Теперь давайте заменим invoke-вызов на execute

lib/tasks/three_datetime.rake

```
desc 'Вывод серии измерений даты и времени'
task :three_datetime do
  3.times do
    Rake::Task['datetime'].execute
  end
end
```

Запуск программы на выполнение приводит к трехкратному вызову rake-задачи :datetime, однако, зависимые задачи :date и :time не запускаются.

```
$ rails three_datetime
Вычисление текущих даты и времени...
Вычисление текущих даты и времени...
Вычисление текущих даты и времени...
```

Параметры rake-задачи

Rake-задачи допускают передачу им входящих параметров. В примере выше rake-задача выводит фразу три раза. Если мы захотим изменить это значение придется исправлять код rake-задачи. Вместо этого, мы можем передать rake-задаче параметр, который позволит пользователю самому задавать количество итераций. Для этого методу task передается второй аргумент-массив с названиями параметров.

lib/tasks/params.rake

```
desc 'Проверка работы аргументов'
task :params, [:number] do |t, args|
  puts args[:number]
end
```

Блок метода task в этом случае принимает два параметра. Нас будет интересовать второй args. Для доступа к переданным значениям объекту args в квадратных скобках передается название параметра.

При вызове такой rake-задачи аргументы следует передавать в квадратных скобках:

```
$ rails params[1]
1
$ rails params[14]
14
```

Теперь можно переработать rake-задачу по вызову rake-задачи datetime, таким образом, чтобы при вызове можно было задать сколько раз должна вызываться задача datetime.

lib/tasks/datetime_execute.rake

```
desc 'Вывод серии измерений даты и времени'
task :datetime_execute, [:number] do |t, args|
  args[:number].to_i.times do
    Rake::Task['datetime'].execute
  end
end
```

При вызове такой rake-задачи аргументы следует передавать в квадратных скобках:

```
$ rails datetime_execute[2]
Вычисление текущих даты и времени...
Вычисление текущих даты и времени...
```

Если при вызове rake-задачи параметр не задается, то внутри rake-задачи он получит значение nil. Эту проблему можно решить, задав значение по умолчанию, для чего используется метод with_defaults объекта args.

lib/tasks/datetime_execute.rake

```
desc 'Вывод серии измерений даты и времени'
task :datetime_execute, [:number] do |t, args|
  args.with_defaults(number: 2)
  args[:number].to_i.times do
    Rake::Task['datetime'].execute
  end
end
```

В этом случае, если параметр не задан при вызове rake-задачи, он получит по умолчанию значение 2.

```
$ rails datetime_execute
Вычисление текущих даты и времени...
Вычисление текущих даты и времени...
```

Генераторы

Ruby on Rails не только выполняет первоначальное развертывание заглушки, но и позволяет осуществить генерацию моделей, контроллеров и представлений. Для авто-генерации нового кода предназначена специальная команда — rails generate. Чтобы узнать ее возможности, команду следует запустить без дополнительных параметров. Вместо полного варианта rails generate можно использовать сокращенный — rails g:

```
$ rails g
...
Rails:
```

```
  application_record
  assets
  benchmark
  channel
  controller
  generator
  helper
  integration_test
  jbuilder
  job
  mailbox
  mailer
  migration
  model
  resource
  scaffold
  scaffold_controller
  system_test
  task
...
```

Как видим, представлено большое количество самых разнообразных генераторов. Давайте воспользуемся генератором rails g task, который позволяет создать заглушку для новой Rake-задачки. Для того, чтобы получить справочную информацию по этой команде, ее достаточно запустить без дополнительных параметров:

```
$ rails g task
...
Example:
       `bin/rails generate task feeds fetch erase add`

       Task:        lib/tasks/feeds.rake
```

В ответ выводится подробное описание команды, а в секции Example можно найти пример ее использования.

Использование генераторов

Создадим простейшую Rake-задачу в пространстве имен example:

```
$ rails g task example
       create  lib/tasks/example.rake
```

В отчете команды rails generate всегда выводится список созданных файлов и их расположение. Rake-задачи находятся по пути lib/task — здесь должен находиться только что созданный файл example.rake.

lib/tasks/example.rake

```
namespace :example do
end
```

В файле пока прописан только вызов метода namespace. Ниже представлен модифицированный файл example.rake с задачей hello.

lib/tasks/example.rake

```
namespace :example do
  desc 'Пример rake-задачи'
  task :hello do
    puts 'Hello, world!'
  end
end
```

Теперь, если перейти в консоль и запустить команду rails -T, среди списка Rake-задач можно обнаружить новую задачу example:hello:

```
$ rails -T
rails about               # List versions of all Rails frameworks
rails active_storage:install # Copy over the migration needed to the app
...
rails example:hello       # Пример rake-задачи
...
rails tmp:create          # Creates tmp directories for cache, sockets
rails yarn:install        # Install all JavaScript dependencies
```

Задачу example:hello можно запустить на выполнение:

```
$ rake example:hello
Hello, world!
```

Rake-задачу можно было создать и без генератора. Все, что делают генераторы, можно выполнять вручную, однако генераторы значительно ускоряют процесс разработки.

Настройка генераторов

Поведение генераторов можно настроить. Для этого в конфигурационном файле config/environments/development.rb можно вызывать метод generators. Его блок позволяет настраивать, какие файлы будут создаваться при вызове генераторов.

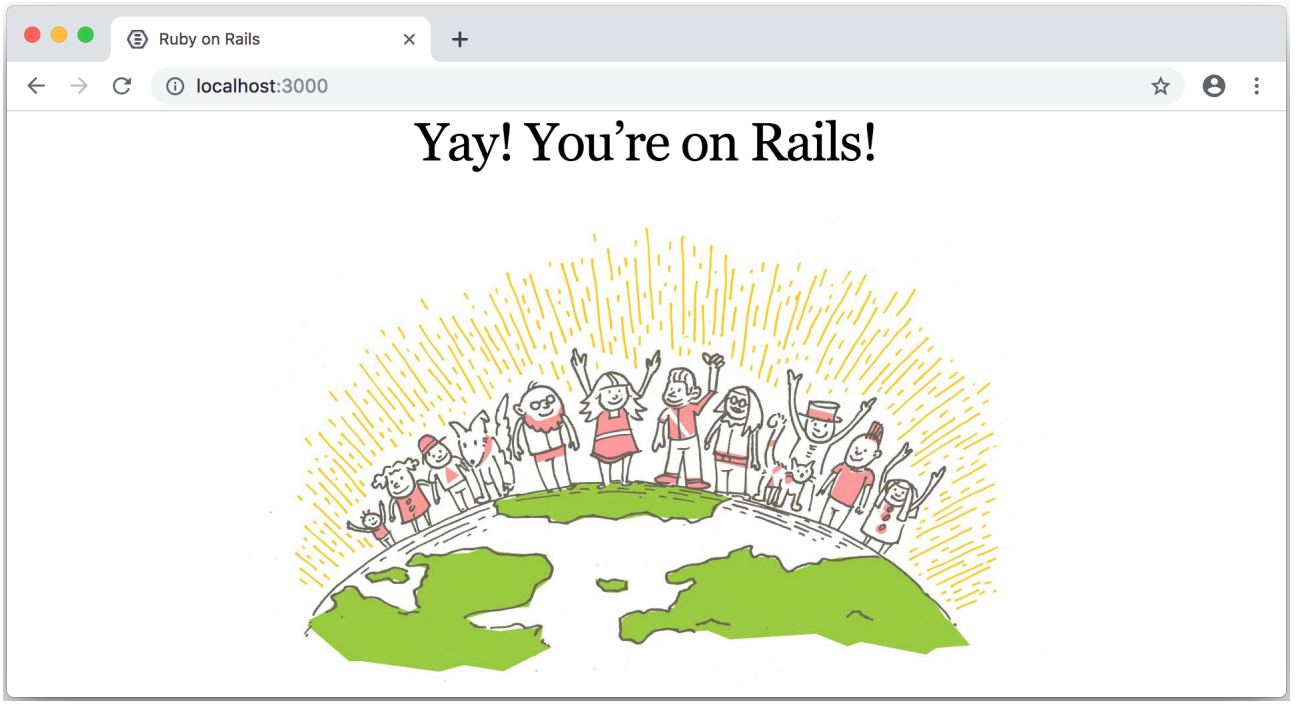
config/environments/development.rb

```
config.generators do |g|
  g.orm             :active_record
  g.test_framework  nil
  g.helper          false
  g.stylesheets     false
  g.javascripts     false
end
```

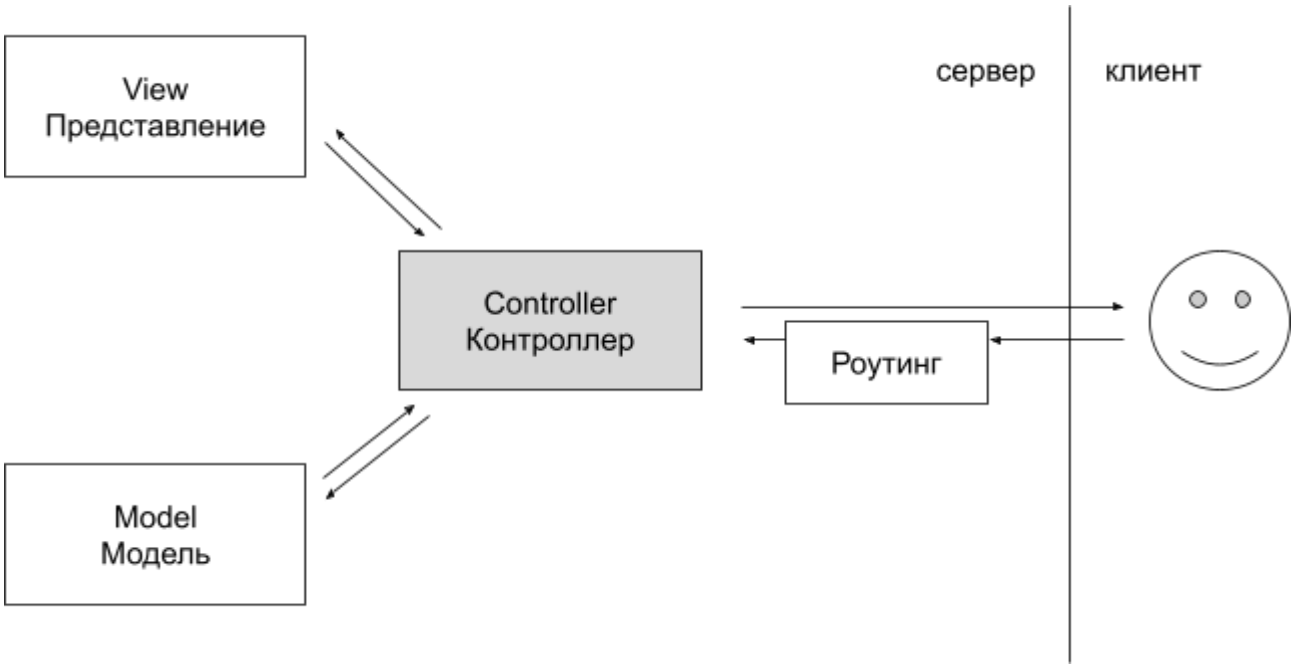
Здесь для генерации моделей по умолчанию используется гем ActiveRecord. Дальнейшие инструкции запрещают генерацию тестов, хелперов, каскадных таблиц стилей и JavaScript-файлов.

Стартовая страница

В этом разделе мы заменим главную страницу сайта своей собственной и воспользуемся для этого генератором контроллера. Сейчас стартовая страница выглядит следующим образом:



Контроллер необходим для обработки запросов клиентов к веб-приложению — это точка входа в схеме MVC.



Rails-контроллер — это класс, в котором определены методы-экшены (действия). Последним сопоставляются роуты, которые вводятся в адресной строке браузера.

С помощью команды rails generate controller можно создать контроллер и представления. Набрав команду без аргументов, можно ознакомиться со справкой по ее использованию:

```
$ rails g controller
Usage:
  rails generate controller NAME [action action] [options]
...
Example:
      `rails generate controller CreditCards open debit credit close`

    CreditCards controller with URLs like /credit_cards/debit.
        Controller: app/controllers/credit_cards_controller.rb
        Test:       test/controllers/credit_cards_controller_test.rb
      Views:       app/views/credit_cards/debit.html.erb [...]
      Helper:      app/helpers/credit_cards_helper.rb
```

В секции Example приводится пример создания контроллера CreditCards с четырьмя экшенами: open, debit, credit и close. Для стартовой страницы можно создать контроллер Home с единственным экшеном index:

```
$ rails g controller Home index
      create    app/controllers/home_controller.rb
       route    get 'home/index'
      invoke    erb
      create    app/views/home
      create    app/views/home/index.html.erb
       ...
```

Как можно видеть, генератор формирует контроллер в каталоге app/controllers, запись в файле config/routes.rb с роутингом для контроллера, а также шаблоны представлений в каталоге app/views.

Когда запрос поступает приложению Ruby on Rails, после его первичной обработки он передается в подсистему роутинга. Управление роутингом сосредоточено в файле config/routes.rb.

config/routes.rb

```
Rails.application.routes.draw do
  get 'home/index'
end
```

Сейчас в файле записан только один роут — в виде метода get, которому передается путь 'home/index'.

Метод get сопоставляет GET-запрос по адресу http://localhost:3000/home/index с методом index контроллера Home.

Контроллеры, как уже отмечалось, сосредоточены в каталоге apps/controllers — здесь можно обнаружить созданный генератором файл home.rb.

app/controllers/home_controller.rb

```
class HomeController < ApplicationController
  def index
  end
end
```

Внутри файла прописан класс HomeController, который был создан генератором. Как можно видеть, наследуется он от класса ApplicationController, который расположен тут же — в каталоге apps/controllers.

app/controllers/home_controller.rb

```
class ApplicationController < ActionController::Base
end
```

ApplicationController — это базовый класс для всех контроллеров веб-приложения. Он наследуется от класса ActionController::Base, являющегося компонентом Ruby on Rails.

Возвращаясь к HomeController, можно заметить, что в нем определен метод index. Этот метод является экшеном — одной из точек входа веб-приложения, которой может быть сопоставлен один или несколько роутов.

Если сейчас перейти по адресу http://localhost:3000/home/index, то в консоли сервера можно будет обнаружить журнальную запись следующего вида:

```
Started GET "/home/index" for 127.0.0.1 at 2019-02-06 09:33:49 +0300
Processing by HomeController#index as HTML
  Rendering home/index.html.erb within layouts/application
  Rendered home/index.html.erb within layouts/application (1.2ms)
Completed 200 OK in 381ms (Views: 374.0ms | ActiveRecord: 0.0ms)
```

В первой строке можно видеть обращение к роуту /home/index, во второй строке сообщается, что за обработку запроса отвечает контроллер HomeController и его экшен index.

Один и тот же экшен может обслуживать запросы от множества роутов. Для этого в файл config/routes.rb необходимо добавить новые роуты, которые будут указывать на экшен index контроллера HomeController. Это можно сделать добавлением вызова метода get и передачей ему дополнительного параметра to: со значением 'home#index'. Так, ниже формируется роут для адреса http://localhost:3000/hello.

config/routes.rb

```
Rails.application.routes.draw do
  get 'home/index'
  get 'hello', to: 'home#index'
end
```

Если необходимо, чтобы контроллер обслуживал главную страницу, можно воспользоваться специальным методом root.

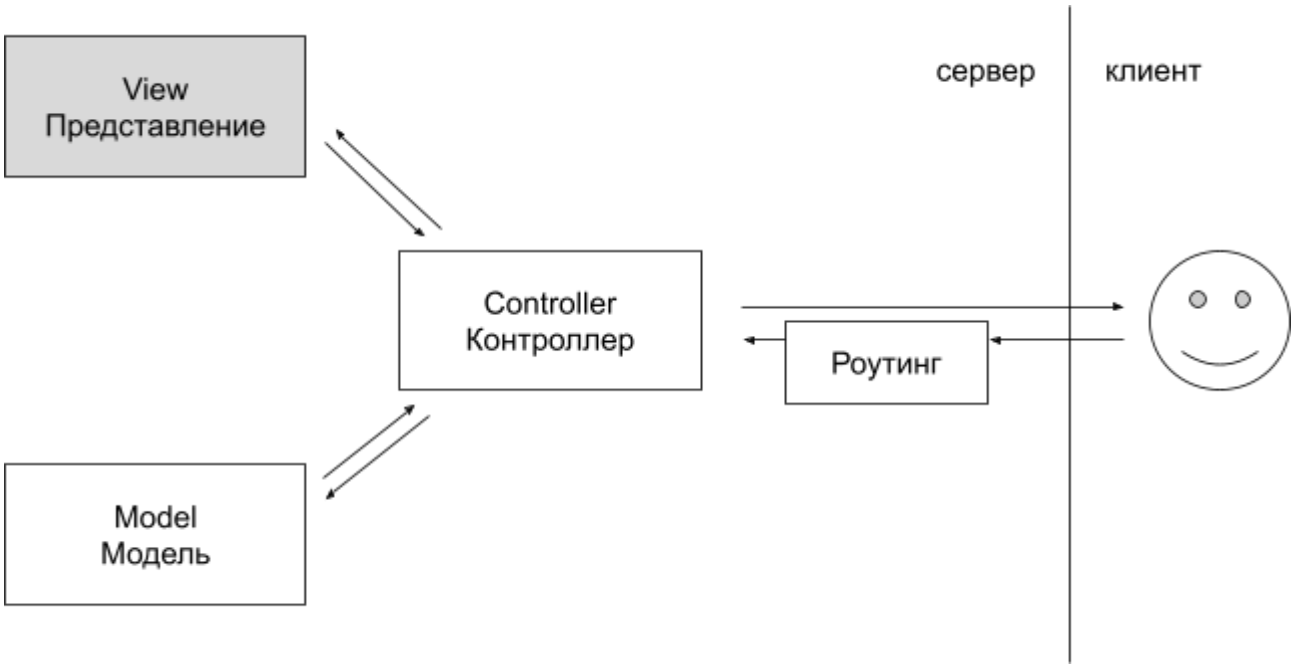
config/routes.rb

```
Rails.application.routes.draw do
  get 'home/index'
  get 'hello', to: 'home#index'
  root 'home#index'
end
```

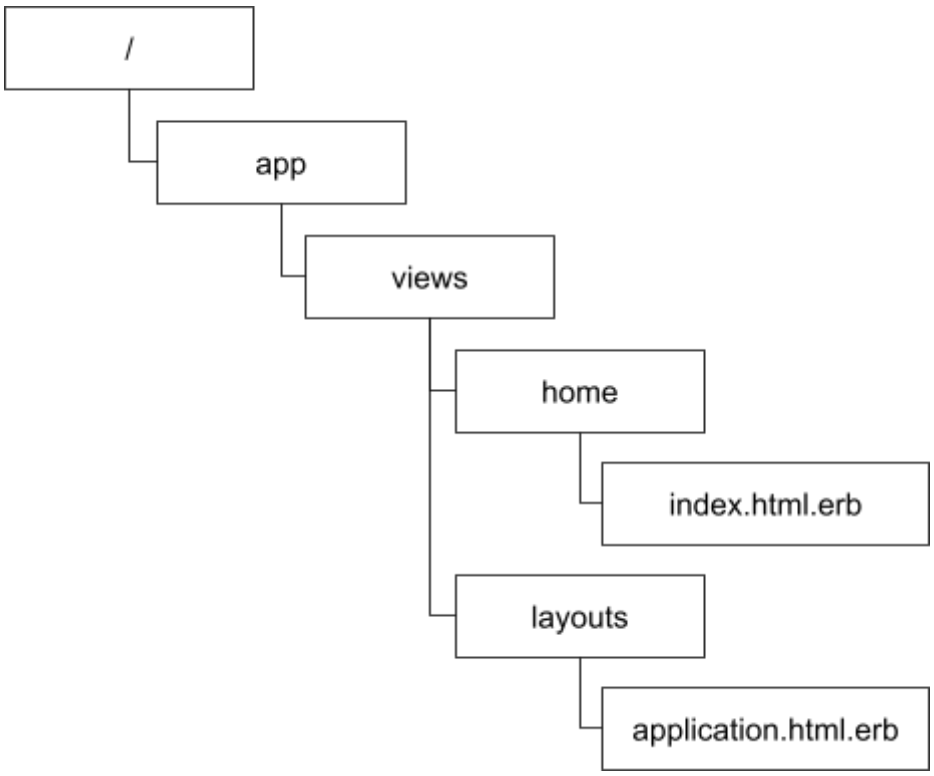
Теперь на главной странице вместо приветственной заглушки фреймворка Ruby on Rails будет отображаться результат работы контроллера HomeController.

Представления

Помимо контроллера, для формирования страницы необходимы шаблоны представления.



В Ruby on Rails приложении они сосредоточены в каталоге views.



По пути app/views/home/index.html.erb можно обнаружить заготовку шаблона представления для экшена HomeController#index.

app/views/home/index.html.erb

```
<h1>Home#index</h1>
<p>Find me in app/views/home/index.html.erb</p>
```

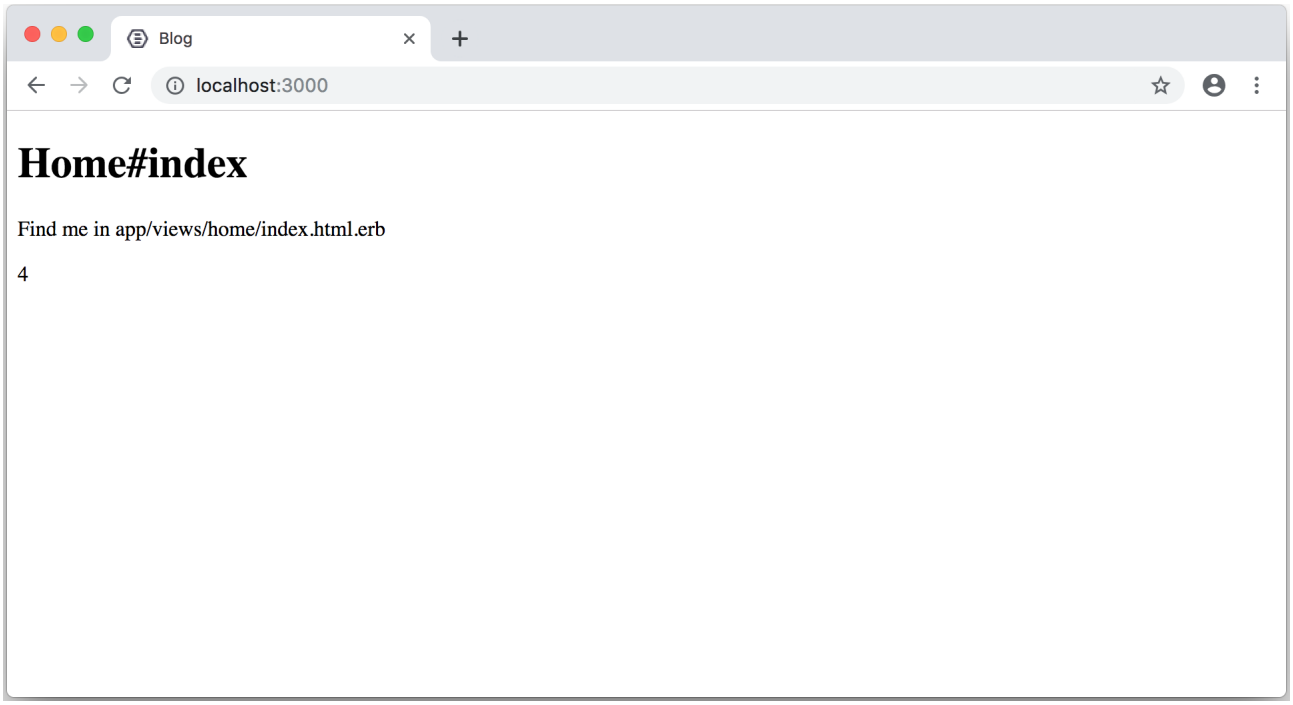
Обработкой такого шаблона занимается стандартная библиотека Erb языка Ruby.

ERB-шаблоны позволяют при помощи специальных тэгов <%= ... %> вставлять любой Ruby-код. В следующем примере вычисляется выражение 2 + 2, которое помещается в HTML-параграф ... . Последнее необходимо, чтобы результирующее выражение на HTML-странице было размещено на отдельной строке.

app/views/home/index.html.erb

```
<h1>Home#index</h1>
<p>Find me in app/views/home/index.html.erb</p>
<p><%= 2 + 2 %></p>
```

Теперь страница, которую обслуживает это представление, будет выглядеть следующим образом.



Так как контроллер называется HomeController, его представления сосредоточены в каталоге app/views/home. Экшен называется index, поэтому и файл с шаблоном называется index.html.erb. Если файлы и каталоги будут называться по-другому, все сломается. Для исправления ситуации потребуется явно прописать вызов метода render в экшене HomeController#index.

app/controllers/home_controller.rb

```ruby
class HomeController < ApplicationController
  def index
    render 'home/index'
  end
end
```

Однако, если файлы и каталоги названы в соответствии с соглашениями, принятыми в Ruby on Rails, можно не указывать явный вызов метода render. И поскольку его можно не указывать, он обычно никогда и не указывается.

Ruby on Rails построен на многочисленных соглашениях. С одной стороны, для работы с фреймворком нужно изучать и знать эти соглашения, с другой стороны, благодаря им код становится очень компактным.

Когда вы только начинаете изучать Ruby on Rails, очень легко совершить ошибки, нарушив какое-либо из соглашений. Поэтому на начальных этапах важно использовать генераторы, которые правильно назовут файлы и не позволят ошибиться.

Логгирование

Логгирование или журналирование — это сохранение отладочной информации о запросах и фоновых операций. Если что-то идет не так, отказывают сервисы, базы данных, в вашем коде имеется ошибка, по логам потом можно восстановить порядок действий пользователей и события которые при этом возникали.

По умолчанию в режиме разработки (development) логи выводятся в стандартный поток ввода-вывода. В результате при обращении к страницам вашего сайта, вы можете видеть, что происходит. Если мы запустим приложение при помощи команды rails s и обратимся к главной странице сайта, мы можем увидеть примерно следующие строки, которые будут выведены в консоль.

```
$ bundle exec rails s
=> Booting Puma
=> Rails 6.1.3.2 application starting in development
=> Run `bin/rails server --help` for more startup options
Puma starting in single mode...
* Puma version: 5.3.2 (ruby 3.0.0-p0) ("Sweetnighter")
*  Min threads: 5
*  Max threads: 5
*  Environment: development
*            PID: 47595
* Listening on http://127.0.0.1:3000
* Listening on http://[::1]:3000
Use Ctrl-C to stop
Started GET "/" for 127.0.0.1 at 2021-06-06 10:00:20 +0300
Processing by HomeController#index as HTML
  Rendering layout layouts/application.html.erb
  Rendering home/index.html.erb within layouts/application
```

```
  Rendered home/index.html.erb within layouts/application (Duration: 1.8ms |
Allocations: 320)
[Webpacker] Everything's up-to-date. Nothing to do
  Rendered layout layouts/application.html.erb (Duration: 25.4ms | Allocations: 6292)
Completed 200 OK in 46ms (Views: 30.2ms | ActiveRecord: 0.0ms | Allocations: 9592)
```

У нас сейчас не так много знаний о работе Ruby on Rails приложения, однако, мы попробуем расшифровать содержимое логов, чтобы вы начинали ими пользоваться с самого начала. Анализ логов — это одна из составляющих быстрой разработки на Ruby on Rails. С их помощью вы можете быстро вскрывать ошибки, неоптимальные режимы работы приложения. Логи у вас постоянно должны быть перед глазами, именно туда Ruby on Rails будет выводить сообщения об ошибках.

Информация об обращении к странице начинается со строки

```
Started GET "/" for 127.0.0.1 at 2021-06-06 10:00:20 +0300
```

Это строка сообщает, что мы обратились методом GET по адресу / с локальной машины (IP-адрес 127.0.0.1) в такой-то момент времени.

Информация о контроллере (HomeController) и экшене (index):

```
Processing by HomeController#index as HTML
```

Далее сообщается о том, какие шаблоны были задействованы для формирования страницы, а так же сколько времени ушло на рендеринг страницы

```
  Rendering layout layouts/application.html.erb
  Rendering home/index.html.erb within layouts/application
  Rendered home/index.html.erb within layouts/application (Duration: 1.8ms |
Allocations: 320)
[Webpacker] Everything's up-to-date. Nothing to do
  Rendered layout layouts/application.html.erb (Duration: 25.4ms | Allocations: 6292)
```

В конце обработки запроса выводится отчет об HTTP-коде, который был отправлен клиенту (здесь это 200), общем времени выполнения запроса (46 милисекунд), а так же сколько из этого времени ушло на рендеринг представления (30.2 миллисекунды) и сколько времени было затрачено на взаимодействие с базой данных. Так как активных SQL-запросов у нас пока нет, мы сохраняем и не извлекаем информацию из базы данных, затраченное время составило 0 миллисекунд. Напоминаем, что в одной секунде 1000 миллисекунд.

```
Completed 200 OK in 46ms (Views: 30.2ms | ActiveRecord: 0.0ms | Allocations: 9592)
```

Значение Allocations сообщает о количестве Ruby-объектов, созданных для обработки операции. С точки зрения производительности, чем меньше это число тем лучше.

Производительность

В development-режиме скорость редеринга страницы может быть медленной, достигая секунды и больше. Это связано с тем, что Ruby on Rails функционирует в отладочном режиме, облегчая разработку в ущерб производительности.

В development-режиме классы загружаются при каждом обращении к странице, а не один раз при старте приложения, кэш, как правило, отключается. Кроме того в логи выводится более подробный вывод по сравнению с production-режимом. Все это требует дополнительных затрат процессора, обращений к жесткому диску и выливается в дополнительные миллисекунды. В реальности в production-окружении на сервере приложение будет работать гораздо быстрее.

Для того, чтобы оценить медленно или быстро работает ваш сайт, можно ориентироваться на следующие цифры. При генерации страницы на сервере в 600 миллисекунд и выше пользователи начинают замечать задержки в работе сайта и жаловаться на его медленную работу. В интервале от 200 до 600 миллисекунд пользователи ничего не замечают, но поисковые роботы отмечают ваш сайт как медленный, а поэтому в поисковой выдаче вы будете ранжироваться ниже. Если вам важно, чтобы в поиске вы были выше ваших конкурентов, добивайтесь более высокой скорости отдачи страниц с сервера. Отдача страницы за 100-200 мс это работа хорошо-оптимизированных сайтов.

Самый простой вариант анализа работы приложения, это подключение сервиса NewRelic. Для его подключения необходимо зарегистрироваться на сайте и подключить к приложению в Gemfile специальной разработанный гем newrelic_rpm.

Gemfile

```
...
gem 'newrelic_rpm'
...
```

Кроме того, вам потребуется создать yaml-файл config/newrelic.yml для задания параметров подключения:

config/newrelic.yml

```
development:
  license_key: '...'
  app_name: my_blog
  agent_enabled: false

production:
  license_key: '...'
  app_name: my_blog
  agent_enabled: true
```

Ключ (license_key) вам выдается после регистрации в NewRelic. В app_name вы можете задать название вашего приложения, чтобы отличать его от других приложений, если к аккаунту их будет подключено несколько. При помощи параметра agent_enabled можно включать (true) или отключать (false) логгирование в NewRelic. В примере выше мы включили логгиирование в сервис в продакшен-окружении и отключили его для режима разработки.

У нас еще будут уроки посвященный оптимизации работы приложения, а так же инструментам для контроля и анализа производительности. Тем не менее вы уже сейчас можете использовать информацию в логах для оптимизации вашего приложения.

Выводим собственную информацию в лог

В лог можно выводить свою собственную информацию, например, с целью отладить работу приложения. Для этого можно использовать объект Rails.logger. Ниже приводится пример, в котором в экшене index контроллера HomeController при помощи метода info в лог выводится фраза "Мы находимся в HomeController#inex".

app/controllers/home_controller.rb

```
class HomeController < ApplicationController
  def index
    Rails.logger.info 'Мы находимся в HomeController#inex'
    render 'home/index'
  end
end
```

После обращения к странице http://localhost:3000/ можно обнаружить эту фразу в лог-выводе.

```
Started GET "/" for 127.0.0.1 at 2021-06-06 11:02:57 +0300
Processing by HomeController#index as HTML
Мы находимся в HomeController#inex
...
Completed 200 OK in 50ms (Views: 44.2ms | ActiveRecord: 0.0ms | Allocations: 9413)
```

Иногда трудно найти свою фразу среди сотни других строк. В этом случае можно окружить ее какими-то хорошо-выделяющимися элементами. Например, длинной последовательностью из символов решетки #.

app/controllers/home_controller.rb

```
class HomeController < ApplicationController
  def index
    Rails.logger.info '##################################'
    Rails.logger.info 'Мы находимся в HomeController#inex'
    Rails.logger.info '##################################'
    render 'home/index'
```

```
  end
end
```

В этом случае обнаружить нужную строку в лог-выводе гораздо проще

```
Started GET "/" for 127.0.0.1 at 2021-06-06 11:11:10 +0300
Processing by HomeController#index as HTML
###################################
Мы находимся в HomeController#inex
###################################
  ...
  Rendered layout layouts/application.html.erb (Duration: 9.2ms | Allocations: 3504)
Completed 200 OK in 11ms (Views: 10.2ms | Allocations: 4142)
```

Метод info, это не единственный способ отправить информацию в лог. Помимо этого метода объект Rails.logger предоставляет методы debug, warn, error, fatal. Названия этих методов совпадают с названиями уровней логирования.

Уровни логгирования

Сообщения в логах имеют разную степень важности: это может быть отладочная информация, предупреждения, ошибки. Всего предусмотрено пять уровней ошибок, представленных в таблице ниже. Название методов объекта Rails.logger совпадают с названиями уровней в таблице.

Уровень	Описание
debug	Отладочная информация для разработчиков
info	Информация о штатной работе приложения
warn	Предупреждения, недочеты системы, которые пока не мешают ее работе. Однако предупреждения требуют внимания разработчика.
error	Ошибки в работе приложения, которые не требуют перезапуска сервера
fatal	Ошибки в работе приложения, которые требуют перезапуска сервера

В таблице уровень критичности сообщений падает сверху вниз. В development-режиме выводятся все сообщения. Если заглянуть в конфигурационный файл config/environments/production.rb в нем можно обнаружить директиву config.log_level, задающую уровень, начиная с которого сообщения помещаются в лог-файл.

config/environments/production.rb

```
...
config.log_level = :info
...
```

Выставленный по умолчанию уровень :info означает, что на сервере в лог будут попадать сообщения уровня info, warn, error и fatal. debug-сообщения в лог попадать не будут.

Логгирование в production-режиме

В режиме разработки лог-информация выводится в стандартный поток ввода-вывода (STDOUT). Так ее удобнее анализировать во время разработки. Однако, во время работы приложения на сервере, наблюдать эту информацию в режиме реального времени некому. Поэтому вместо вывода в стандартный поток, она отправляется в файл log/production.log (название файла будет совпадать с названием окружения).

Впрочем этот режим можно отменить. Если внимательно прочитать содержимое файла config/environments/production.rb, в нем можно обнаружить следующие строки.

config/environments/production.rb

```
...
  if ENV["RAILS_LOG_TO_STDOUT"].present?
    logger           = ActiveSupport::Logger.new(STDOUT)
    logger.formatter = config.log_formatter
    config.logger    = ActiveSupport::TaggedLogging.new(logger)
  end
...
```

Эти строки означают, что если Ruby on Rails будет передана переменная окружения RAILS_LOG_TO_STDOUT, то будет включен режим отправки лог-записей в стандартный поток ввода-вывода. Это может пригодится при запуске приложения в docker-контейнере или при отладке production-окружения.

Если вы захотите запустить приложение в production-режиме локально, можно воспользоваться параметром -e, которому передается название окружения. Чтобы логи по прежнему выводились в консоль, как в случае режима разработки, нам потребуется задать значение переменной окружения RAILS_LOG_TO_STDOUT.

```
$ RAILS_LOG_TO_STDOUT=true rails c -e production
```

Переменные окружения часто будут использоваться для настройки Ruby on Rails приложения. Не только для включения различных режимов работы, как в примере выше, но и для передачи различной чувствительной информации, например, паролей. Вместо пароля в конфигурационном файле можно задать переменную окружения и код приложения можно без опаски отправлять в открытый git-репозиторий: никто не сможет узнать пароль к базе данных или почтовому ящику. На сервере или в режиме разработки пароли можно передать в переменной окружения. Для удобства управления

переменными окружениями в Rails-сообществе используют гем dotenv с которым мы познакомимся на следующих уроках.

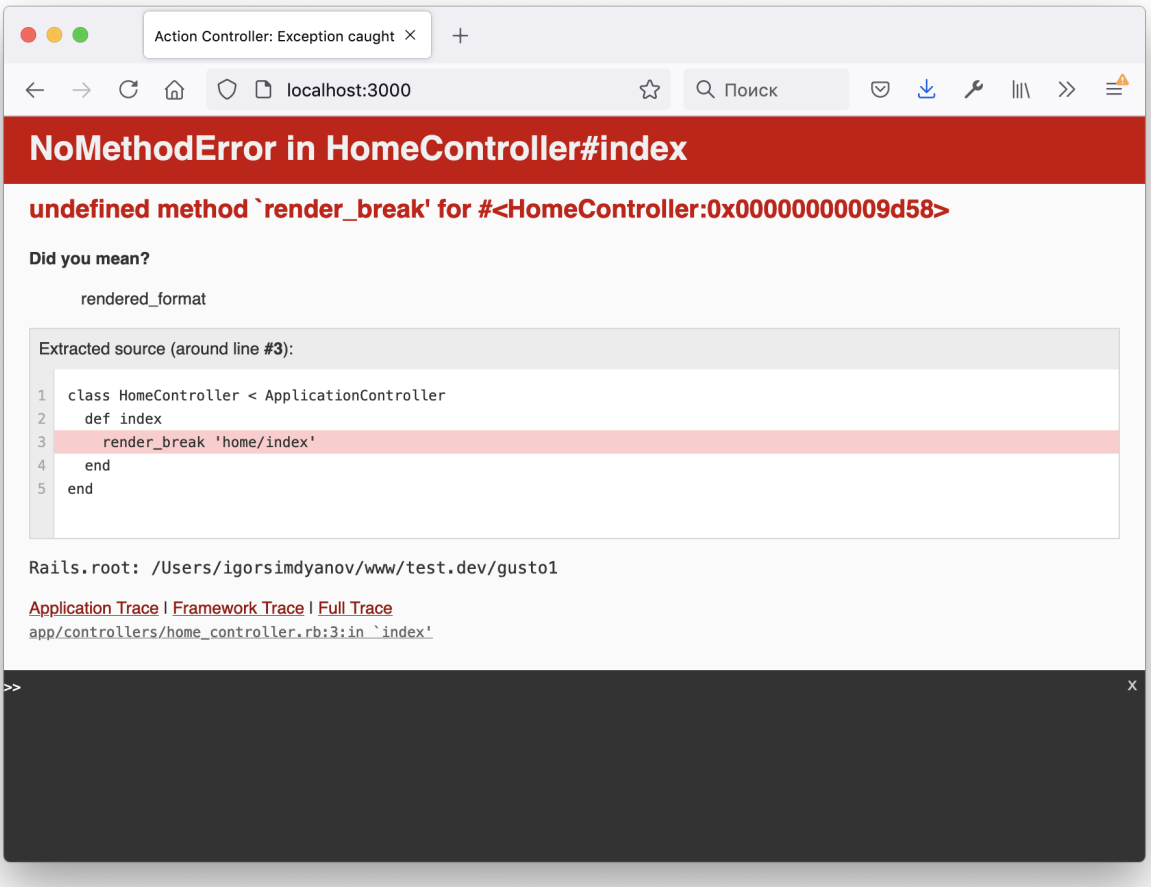
Вывод ошибки в лог-выводе

В случае возникновения ошибки в вашем Ruby-коде, помимо сообщения об ошибке, выводится backtrace-вывод исключительной ситуации для облегчения локализации проблемы. Давай намеренно добавим опечатку в экшен HomeController#index, чтобы посмотреть на реакцию RoR-приложения. В примере ниже вместо метода render, мы используем несуществующий метод render_break.

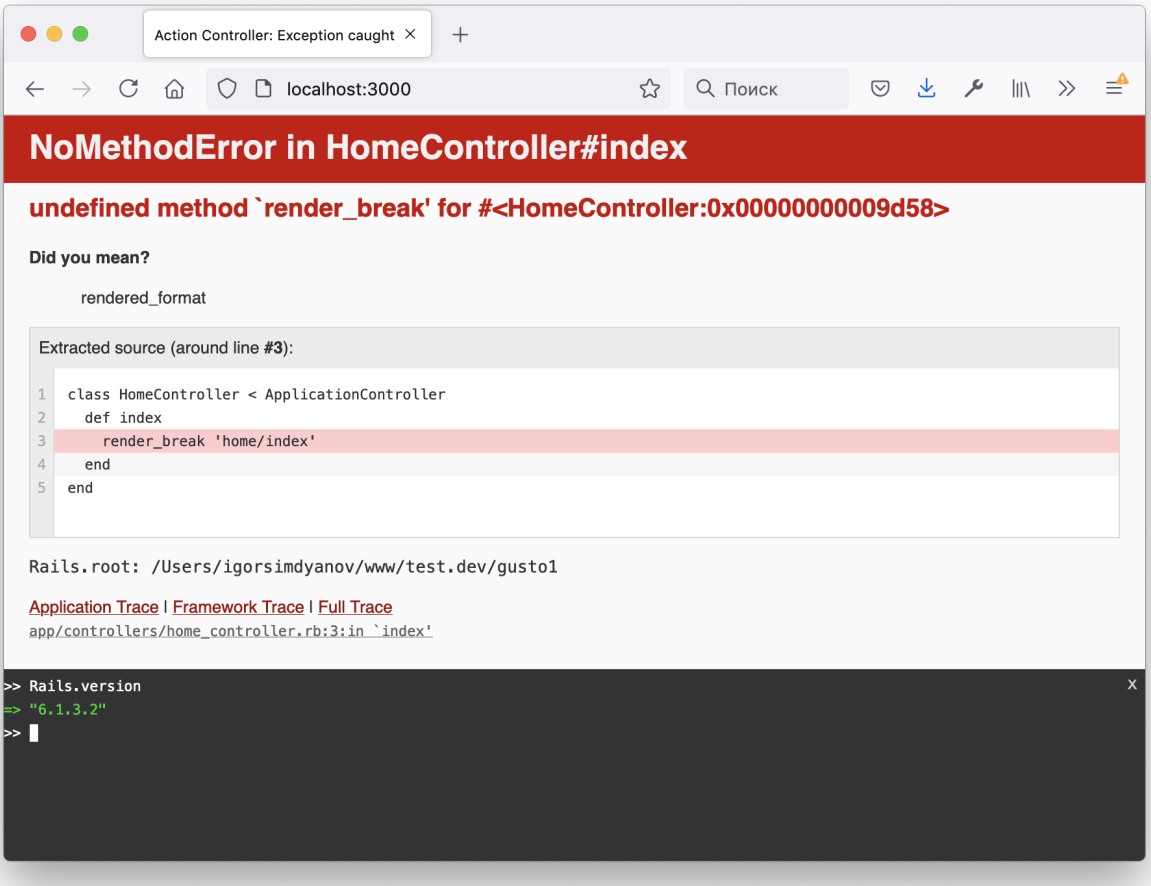
app/controllers/home_controller.rb

```
class HomeController < ApplicationController
  def index
    render_break 'home/index'
  end
end
```

Перезагрузив страницу в браузере, мы получаем страницу с сообщением об ошибке. На этой странице имеет вся информация, включая путь к файлу и номер строки, где возникла ошибка.



Внизу страницы имеется даже rails-консоль, в которой можно поэкспериментировать, не обращаясь к командной строке.



Обратившись к консоли, в которой у нас запущена команда rails s, можно обнаружить лог-вывод.

```
Started GET "/" for 127.0.0.1 at 2021-06-06 13:53:30 +0300
Processing by HomeController#index as HTML
Completed 500 Internal Server Error in 14ms (ActiveRecord: 0.0ms | Allocations: 4071)

NoMethodError (undefined method `render_break' for #<HomeController:0x000000000075d0>
Did you mean?  rendered_format):

app/controllers/home_controller.rb:3:in `index'
```

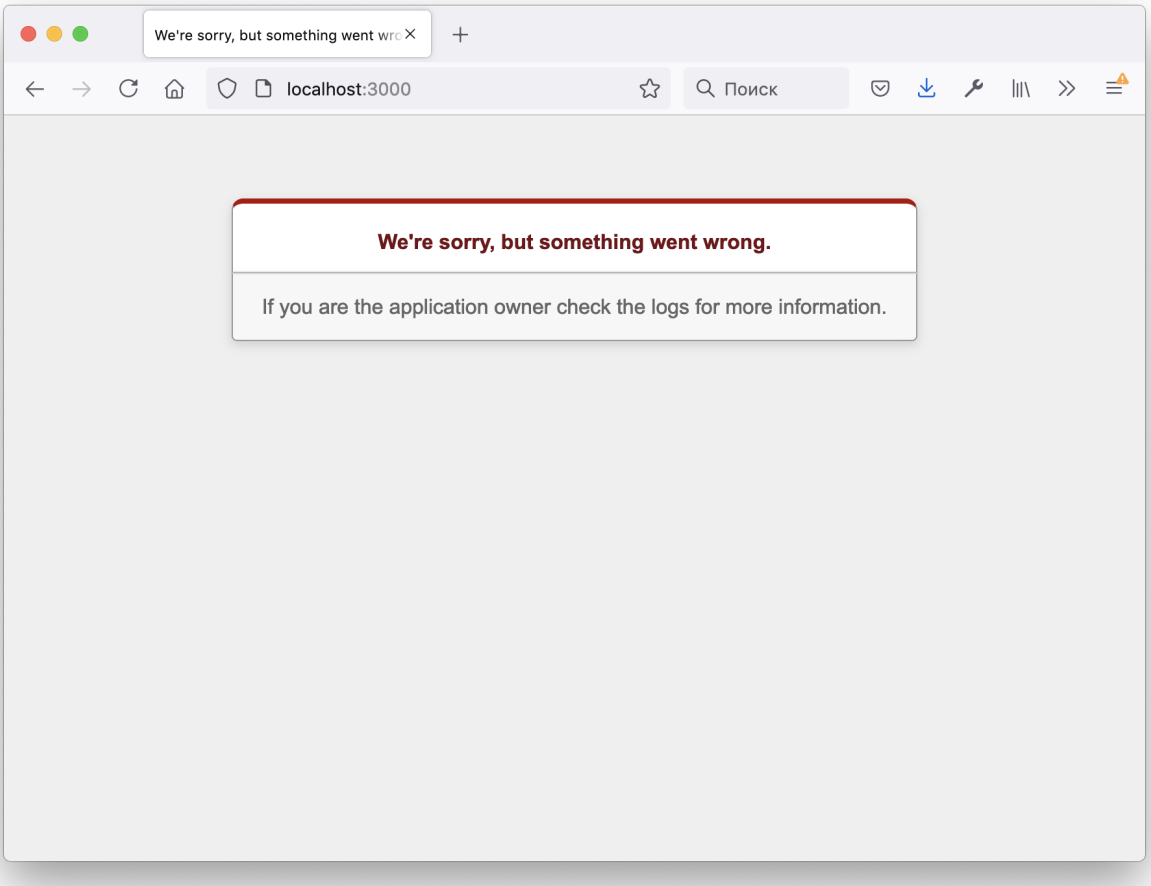
В отличие от успешного ответа, здесь вместо HTTP-кода 200 возвращается код 500 — ошибка на сервере. Кроме этого выводится backtrace исключительной ситуации, по которому можно понять, что ошибка возникла в файле app/controllers/home_controller.rb на третьей строке.

Работа с логом является более предпочтительным вариантом. Дело в том, что страница ошибки в production-окружении отличается от режима разработки. Давайте запустим веб-сервер в продакшен режиме и снова попробуем воспроизвести ошибку.

```
$ rails c -e production
```

Внимание! *Если не используется переменная окружения RAILS_LOG_TO_STDOUT, то лог-записи следует искать в config/production.log.*

В этом случае страница с ошибкой будет выглядеть следующим образом.



Это то, что увидят пользователи вашего приложения на сервере. Здесь нет отладочной страницы об ошибке, так как она может дать злоумышленнику информацию о приложении. Как мы видели страница в development-окружении даже позволяет выполнять Ruby-код на стороне сервера. Поэтому в случае ошибок на сервере у вас будет только лог-файл, с сообщением вида:

```
I, [2021-06-06T13:56:21.261631 #91788]  INFO -- :
[6d424b88-bcf1-48a1-8042-cc5c9a694abd] Started GET "/" for 127.0.0.1 at 2021-06-06
13:56:21 +0300
I, [2021-06-06T13:56:21.263351 #91788]  INFO -- :
[6d424b88-bcf1-48a1-8042-cc5c9a694abd] Processing by HomeController#index as HTML
I, [2021-06-06T13:56:21.276311 #91788]  INFO -- :
[6d424b88-bcf1-48a1-8042-cc5c9a694abd] Completed 500 Internal Server Error in 13ms
(Allocations: 3583)
F, [2021-06-06T13:56:21.277074 #91788] FATAL -- :
[6d424b88-bcf1-48a1-8042-cc5c9a694abd]
```

```
[6d424b88-bcf1-48a1-8042-cc5c9a694abd] NoMethodError (undefined method `render_break'
for #<HomeController:0x00000000007ad0>
Did you mean?  rendered_format):
[6d424b88-bcf1-48a1-8042-cc5c9a694abd]
[6d424b88-bcf1-48a1-8042-cc5c9a694abd] app/controllers/home_controller.rb:3:in
`index'
```

Как видно из примера выше, каждая строка снабжается уровнем лог-записи (INFO, FATAL) и уникальным идентификатором запроса (6d424b88-bcf1-48a1-8042-cc5c9a694abd). Последний нужен для того, чтобы отличать записи относящиеся к запросу от других параллельных запросов. Это позволяет отфильтровать все записи относящиеся к проблемному запросу при помощи утилиты grep:

```
$ cat production.log | grep '6d424b88-bcf1-48a1-8042-cc5c9a694abd'
```

Команда отберет из log-файла только те записи, в которых имеется последовательность '6d424b88-bcf1-48a1-8042-cc5c9a694abd'.

Иногда бывает полезно посмотреть окрестности ошибки. Например если мы будем искать только по вхождению слова FATAL, то поиска по лог-файлу при помощи утилиты grep выдаст нам слишком мало информации

```
$ cat production.log | grep FATAL
F, [2021-06-06T14:09:27.688517 #94256] FATAL -- :
[5d534e26-29e9-44cf-ba5e-0517761f23e4]
```

Чтобы посмотреть окрестности найденной строки можно воспользоваться параметрами -B (before) и -A (after), которые позволяют задать количество отображаемых строк до и после найденной строки.

```
$ cat production.log | grep FATAL -A 3 -B 3
I, [2021-06-06T14:09:27.668552 #94256]  INFO -- :
[5d534e26-29e9-44cf-ba5e-0517761f23e4] Started GET "/" for 127.0.0.1 at 2021-06-06
14:09:27 +0300
I, [2021-06-06T14:09:27.673986 #94256]  INFO -- :
[5d534e26-29e9-44cf-ba5e-0517761f23e4] Processing by HomeController#index as HTML
I, [2021-06-06T14:09:27.687454 #94256]  INFO -- :
[5d534e26-29e9-44cf-ba5e-0517761f23e4] Completed 500 Internal Server Error in 13ms
(Allocations: 3583)
F, [2021-06-06T14:09:27.688517 #94256] FATAL -- :
[5d534e26-29e9-44cf-ba5e-0517761f23e4]
[5d534e26-29e9-44cf-ba5e-0517761f23e4] NoMethodError (undefined method `render_break'
for #<HomeController:0x00000000007ad0>
Did you mean?  rendered_format):
[5d534e26-29e9-44cf-ba5e-0517761f23e4]
```

Или можно воспользоваться параметром -C (contex) который выводит заданное количество строк до и после найденной строки:

```
$ cat production.log | grep FATAL -C 3
I, [2021-06-06T14:09:27.668552 #94256]  INFO -- :
[5d534e26-29e9-44cf-ba5e-0517761f23e4] Started GET "/" for 127.0.0.1 at 2021-06-06
14:09:27 +0300
I, [2021-06-06T14:09:27.673986 #94256]  INFO -- :
[5d534e26-29e9-44cf-ba5e-0517761f23e4] Processing by HomeController#index as HTML
I, [2021-06-06T14:09:27.687454 #94256]  INFO -- :
[5d534e26-29e9-44cf-ba5e-0517761f23e4] Completed 500 Internal Server Error in 13ms
(Allocations: 3583)
F, [2021-06-06T14:09:27.688517 #94256] FATAL -- :
[5d534e26-29e9-44cf-ba5e-0517761f23e4]
[5d534e26-29e9-44cf-ba5e-0517761f23e4] NoMethodError (undefined method `render_break'
for #<HomeController:0x00000000007ad0>
Did you mean?  rendered_format):
[5d534e26-29e9-44cf-ba5e-0517761f23e4]
```

Класс Logger стандартной библиотеки Ruby

Класс Logger, который используется Ruby on Rails на самом деле класс из стандартной библиотеки Ruby. Если вы будете писать на чистом Ruby или создавать rack-приложение, вы сможете организовать логгированиие, как в Ruby on Rails.

```
$ irb
 > require 'logger'
=> true
 > logger = Logger.new STDOUT
=> #<Logger:0x00007fbe5b013558 @level=0,...
 > logger.warn 'Предупрежение: мы находимся в чистом Ruby'
W, [2021-06-06T14:15:26.620151 #95496]  WARN -- : Предупрежение: мы находимся в
чистом Ruby
=> true
 > logger.info 'Класс Logger - это элемент стандартной библиотеки Ruby'
I, [2021-06-06T14:16:04.334038 #95496]  INFO -- : Класс Logger - это элемент
стандартной библиотеки Ruby
=> true
```

Домашняя работа

Задания со звездочкой факультативны, необязательны.

- Создайте rake-задачу, которая в tmp-каталоге rails-приложения создает файл development.txt в который записана текущая дата и время.
- Создайте rake-задачу, которая принимает дату рождения пользователя и выводит его возраст.
- Создайте rake-задачу, которая подсчитывает количество ruby-файлов в текущем rails-проекте.
- Создайте rake-задачу, которая подсчитывает количество строк в ruby-файлах текущего rails-проекта.

- **(*)** Rake-задача, которая выводит список баз данных, развернутых в PostgreSQL.
- **(*)** После знака вопроса ? в URL можно указывать GET-параметры. На самом деле эти параметры можно использовать и с POST-запросами. Но так как в GET-запросах не задействовано тело HTTP-запроса, это самый распространенный способ передачи динамических значений. Например, http://localhost:3000/?hello=world. Здесь hello выступает ключом GET-параметра, а wold — значение. При помощи Rails.logger.info выведите значение GET-параметра hello в лог Rails-приложения.

Задание к концу курса

Разработайте на Ruby on Rails приложение для планирования списка дел TODO. Одно должно позволять заводить дела, привязанные к календарю, которые можно привязывать к датам, отмечать выполненными.

Дополнительные материалы

1. Инструкция по установке Ruby в Windows, MacOS и Linux. https://docs.google.com/document/d/1306CXG3XOJxv0ICJYrfYCMSg1p98Xa2Xgr_0JY1k5us/edit#
2. Гем rack https://github.com/rack/rack
3. Тренажерный зал для решения задач на разную тематику почти на всех языках программирования, в том числе на Ruby: https://www.codewars.com/

Используемые источники

1. Самоучитель Ruby. Игорь Симдянов. BHV, 2020.
2. Ruby on Rails для начинающих. Хартл М. ДМК-Пресс. 2017.
3. The Rails 5 Way. Obie Fernandez. Leanpub. 2017.
4. Командная строка Rails https://guides.rubyonrails.org/command_line.html
5. Создание и настройка генераторов и шаблонов https://guides.rubyonrails.org/generators.html
6. Документация класса Logger стандартной библиотеки Ruby https://ruby-doc.org/stdlib-3.0.0/libdoc/logger/rdoc/Logger.html