Урок 2

Введение в структуру Rails

Структура Rails-приложения. Гем bundler. Rake-задачи. Генераторы. Роутинг. Шаблоны. Контроллеры. Сессии.

Новое Rails-приложение

На прошлом занятии мы установили Ruby on Rails и создали первое приложение с его помощью. Для этого использовали команду:

```
rails new blog --skip-test --skip-turbolinks --skip-spring --skip-coffee
--skip-action-cable
 --webpack --database=postgresql
```

В ходе установки утилита rails создает все необходимые файлы для функционирования минимального приложения-заглушки. Во-первых, устанавливаются дополнительные гемы, от которых зависит rails-приложение. Во-вторых, устанавливаются и настраиваются npm-пакеты, необходимые для функционирования клиентского кода, разработанного при помощи JavaScript.

Структура Rails-приложения

Если заглянуть в директорию *blog* с файлами приложения, мы увидим структуру:

```
blog
├── app
│       ├── assets
│       ├── chanels
│       ├── controllers
│       ├── helpers
│       ├── javascript
│       ├── jobs
│       ├── mailers
│       ├── models
│       └── views
├── bin
├── config
├── db
├── lib
├── log
├── node_modules
├── public
├── tmp
├── vendor
├── .babelrc
├── .gitignore
├── config.ru
├── Gemfile
├── Gemfile.lock
├── package.json
├── Rakefile
├── README.md
└── yarn.lock
```

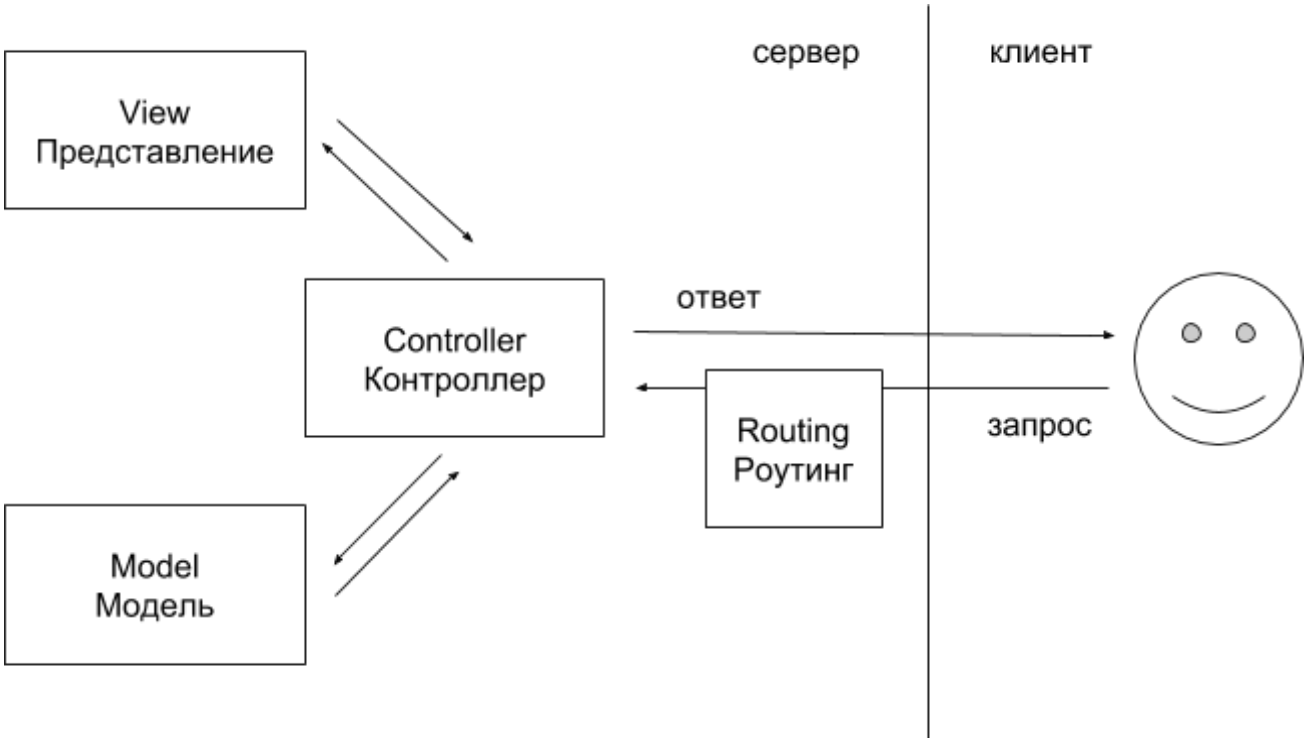
В Ruby on Rails большое внимание уделяется соглашениям. Фреймворк спроектирован таким образом, чтобы настройки по умолчанию подходили для подавляющего большинства случаев и практически не требовали вмешательства. В отличие от Ruby-приложений, в Ruby on Rails почти никогда не придется использовать метод require для подключения библиотек и файлов. Если

следовать соглашениям расположения файлов и компонентов, Ruby on Rails всегда сможет обнаружить необходимые классы. Однако это требует понимания структуры приложения: за что отвечает тот или иной файл, где следует располагать новые файлы.

Папка app

Директория app является ключевой для любого Ruby on Rails-приложения. В ней сосредотачивается весь код, созданный с использованием компонентов Ruby on Rails и обслуживающий приложение.

Именно в этой папке сосредотачиваются компоненты MVC-схемы приложения:



Внутри папки app есть несколько подпапок, среди которых выделим следующие:

- *Models* − содержит файлы с классами моделей;
- *Controllers* − содержит файлы с классами контроллеров;
- *Views* − содержит подпапки с шаблонами представлений.

Эти три подпапки − ключевые. Но помимо них есть еще несколько подпапок, содержащих вспомогательные компоненты:

- *Assets* − ассеты, CSS, JavaScript-файлы, а также файлы изображений. Их использует CSS, также они необходимы в дизайне. При отказе от старого способа обработки ассетов с использованием гема Sprokets в пользу нового гема Webpacker эту папку можно удалить. Более подробно мы рассмотрим оба подхода на седьмом занятии.
- *Channels* − ActionCable-классы, необходимые для обслуживания WebSocket-соединений.
- *Helpers* − небольшие вспомогательные методы, которые доступны в представлениях и позволяют убирать из них сложную или часто повторяющуюся логику.
- *JavaScript* − точка входа для JavaScript-приложения гема Webpacker.
- *Jobs* − классы для создания фоновых задач в рамках гема ActiveJob, встроенного в Rails. Фактически это интерфейс для единообразного оформления фоновых задач для целого класса гемов, реализующих очереди (sidekiq, resque, que, delayed job).
- *Mailers* − обслуживание задач по отправке почтовых сообщений.

Папка bin

Содержит скрипты для запуска приложения и некоторые полезные утилиты:

- **bundle** − исполняемый файл гема bundler, предназначенного для управления гемами и их зависимостями. Конфигурационный файл этого гема находится в Gemfile. После выполнения команды bundle install создается файл Gemfile.lock, содержащий версии гемов. Это замораживает версии. Это гарантирует, что в будущем, если положить Gemfile.lock в систему контроля версий, на сервере и у коллег по проекту будут одни и те же версии библиотек и гемов. Фактически bundler выполняет роль менеджера гемов в ruby-приложениях (как pip в Python, npm в Node.js, Composer в PHP).
- **rails** − главный файл управления RoR-проектом. Он позволяет генерировать код и выполнять задачи по обслуживанию базы данных, ассетов и т.д. Список задач и команд может расширяться сторонними гемами, а также задачами, созданными разработчиком.
- **rake** − исполняемый файл гема rake, позволяющий выстраивать цепочки зависимых друг от друга задач. Например, при создании Rails-приложения выполняется множество задач по генерации кода, выполнению команды bundle, генерации конфигурации Node.js-приложения и установке npm-пакетов. В случае доставки приложения на сервер необходимо выполнить ряд зависимых друг от друга rake-задач. Даже задачи, которые выполняются утилитой rails, большей частью являются замаскированными rake-задачами. Rake − это сокращение от ruby make, отсылка к утилите make, которая вот уже более 40 лет используется в UNIX для сборки C-программ.
- **setup** и **update** − обертка для выполнения rake-задач по первоначальной установке и последующему обновлению приложения. В ходе обучения мы будем выполнять эти задачи вручную, чтобы лучше изучить экосистему обслуживающих задач Ruby on Rails.
- **webpack** и **webpack-dev-server** − утилиты, обслуживающие Webpack и сервер. Для функционирования этих скриптов в системе должны быть установлены Node.js и npm.
- **yarn** − обертка для менеджера node.js-пакетов npm, удобная и безопасная в обслуживании. Для функционирования этого скрипта в системе должна быть установлена утилита yarn.

Окружения

Ruby on Rails из коробки предоставляет сразу как минимум три окружения:

- **development** − окружение по умолчанию, предназначенное для разработки. В этом окружении разработчик проводит большую часть времени. Оно сконфигурировано таким образом, чтобы предоставить максимально удобную среду для разработки и отладки: более подробные логи, моментальная перезагрузка файлов без необходимости перезапуска сервера, отсутствие этапа сборки и компиляции ассетов, отсутствие минификации JS-кода и т.п.
- **test** − тестовое окружение, в котором выполняются автоматические тексты.
- **production** − продакшн-окружение, предназначенное для запуска приложения на конечном сервере. Это окружение сконфигурировано таким образом, чтобы добиться максимальной скорости работы приложения.

Помимо трех стандартных окружений можно заводить собственные. Например, для сервера, предназначенного для демонстрации нового релиза заказчику перед выпуском в продакшн, можно завести дополнительное окружение **preprod**. А если в технологической цепочке выпуска нового релиза предусмотрено ручное тестирование, можно завести дополнительную тестовую площадку **staging,** на которой будут работать тестировщики.

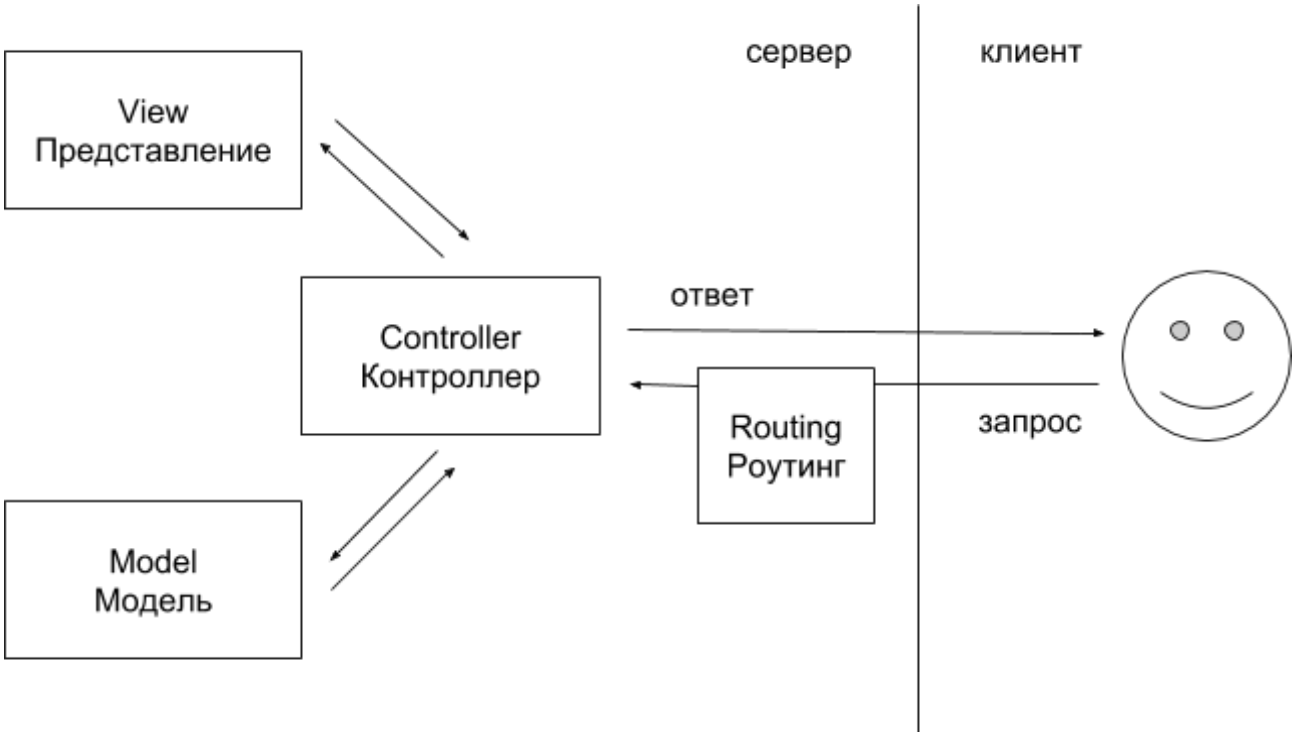
Папка config

Данная папка содержит конфигурационные файлы как самого Ruby on Rails-приложения, так и всех устанавливаемых сторонних гемов. Как правило, начальные установки таких файлов требуют лишь минимального вмешательства.

Ряд конфигурационных файлов имеют несколько вариантов по количеству окружений поддерживаемых системой. Например, в папке config/environments (конфигурация окружений) 3 файла: для development-, test- и production-окружений. Это относится и к папке webpack, внутри которой js-файлы, настраивающие работу webpack под каждое из окружений.

Разбиение конфигурации по разным окружениям может быть реализовано на уровне структуры конфигурационных файлов. Например, внутри файла config/database.yml есть yaml-структура, которая задает параметры доступа к базе данных (логин, пользователь, пароль, имя базы данных) под каждое из окружений.

Отметим файл routes.rb. Это конфигурационный файл подсистемы роутинга. Именно в нем определяется, какой из контроллеров будет обслуживать тот или иной запрос от клиента.



Папка db

Содержит все, что относится к базе данных:

- миграции, позволяющие воссоздавать структуру базы данных на сервере, при установке приложения на новой машине разработчика или при тестировании;
- схема базы данных;
- seed-файл для первичного заполнения базы данных тестовыми или рабочими данными.

Более подробно с базами данных и порядком работы с ними мы познакомимся на 3 и 4 занятиях.

Папка lib

Папка для модулей и классов, которые создаются для приложения, но не являются непосредственными компонентами Ruby on Rails или структуры MVC. Например, мы хотим сделать онлайн-игру *Морской бой*. В игре есть множество классов и объектов Ruby, которые не имеют отношения к Web-программированию и Ruby on Rails: корабли, координаты, выстрел и т.д. Все эти вспомогательные классы, как правило, располагаются в папке lib, а не внутри app, которая обслуживает Web-приложения.

Здесь размещается код rake-задач, которые разработчик создает для обслуживания приложения, базы данных, процедуры деплоя и т.п.

Папка log

Папка содержит логи приложения, причем для каждого окружения предназначен отдельный файл:

- development.log;
- test.log;
- production.log.

Папка public

Папка, являющаяся точкой входа приложения. Любой файл, который помещается в эту папку, становится доступным из приложения. Внутри папки по умолчанию есть ряд файлов-заготовок:

- 404.html − обработчик HTTP-кода 404: страница не найдена.
- 422.html − обработчик HTTP-кода 422: корректный запрос, который сервер не может обработать. Например index.json, если приложение не поддерживает отдачу в json-формате.
- 500.html − обработчик HTTP-кода 500: ошибка на сервере.
- robots.txt − файл с инструкциями для роботов, обходящих сайт. Можно сообщить разделы, которые не следует подвергать индексации, или местоположение sitemap с полным перечнем адресов сайта, чтобы помочь роботу с его индексацией.

Если запустить приложение при помощи команды rails s, то по адресу http://localhost:3000/robots.txt можно обратиться к одному из файлов, расположенных в папке public. В ней находятся статические файлы, а также размещаются файлы ассетов в продакшн режиме после предварительной компиляции.

Папка test

Содержит все, что относится к тестам: сами тесты, вспомогательные хелперы и классы, обслуживающие тесты. Здесь же размещается код, упрощающий заполнение моделей базы данных перед выполнением тестов: фикстуры и фабрики.

Папка tmp

Содержит временные файлы: сокеты, pid-файлы, временные файлы, открываемые ruby-кодом.

Папка vendor

Место хранения кода внешних библиотек. Очень напоминает по функциональности папку lib. Но в lib хранится код, созданный участниками команды разработки текущего проекта, а в vendor хранятся библиотеки внешних команд.

Гем bundler

Обычно гемы устанавливаются при помощи команды gem install. Например, гем rails можно установить при помощи команды:

```
gem install rails
```

При установке гема rails устанавливается большое количество зависимостей. Одна из них − это гем bundler. Именно поэтому его не требуется устанавливать отдельной командой:

```
gem install bundler
```

Более того, в рамках Ruby on Rails-проекта не требуется ставить гемы при помощи команды gem, так как bundler выступает в качестве менеджера гемов (как pip в Python, npm в Node.js, Composer в PHP).

Манифест Gemfile

Состав гемов управляется при помощи файла-манифеста Gemfile. Ниже представлен типичный Gemfile:

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

gem 'rails', '~> 5.1.4'
gem 'pg', '~> 0.18'
gem 'puma', '~> 3.7'
gem 'sass-rails', '~> 5.0'
gem 'uglifier', '>= 1.3.0'
gem 'webpacker'

gem 'jbuilder', '~> 2.5'

group :development, :test do
  gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  gem 'web-console', '>= 3.3.0'
  gem 'listen', '>= 3.0.5', '< 3.2'
end
```

Внутри Gemfile расположен обычный ruby-код, однако оформлен он в виде своеобразного декларативного DSL-языка. Создание таких декларативных объявлений очень типично для Ruby on Rails.

При выполнении команды bundle install (при создании проекта она была запущена автоматически) по манифесту Gemfile генерируется файл Gemfile.lock, в котором фиксируются зависимости гемов, а также их версии. Файл Gemfile.lock включается в состав файлов, которые управляются системой контроля версий. Это гарантирует, что у всех участников команды и на сервере будут одни и те же версии гемов.

Манифест Gemfile состоит из объявлений. Каждый вызов метода gem подключает один из gem-файлов:

```
gem 'kaminari'
gem 'nokogiri'
```

Если гемы подключены в глобальной области файла, они будут доступны всем окружениям. Но при помощи метода group можно задать, в каких окружениях гемы будут установлены, а в каких − нет. Это позволяет не устанавливать в продакшн-окружении гемы, нужные только для разработки или тестирования, или, наоборот, поставить в продакшн режим сервер, который не будет использоваться в development-окружении.

```
group :development do
  gem 'byebug'
end

group :test do
  gem 'capybara'
  gem 'database_cleaner'
end

group :development, :test do
  gem 'rspec-rails'
end
```

Метод gem может принимать второй необязательный аргумент, который задает версию гема. Если версия не указывается, используется последняя стабильная версия.

Версии, как правило, состоят из трех цифр: мажорная, минорная и патч-версия. Мажорная версия обычно меняется при кардинальной перестройке проекта: Rails 4.0.0, Rails 5.0.0 и т.д. Минорная версия меняется при релизе новой стабильной версии − 5.1.0. Патч-версия меняется при исправлении багов или усовершенствованиях, которые не затрагивают интерфейсы и поведение программного обеспечения − 5.1.1.

Влиять на версии гемов можно в манифест-файле Gemfile. Например, символ >= требует установки любой версии гема выше указанной:

```
gem 'web-console', '>= 3.3.0'
```

Можно указать диапазон версий:

```
gem 'listen', '>= 3.0.5', '< 3.2'
```

Или можно зафиксировать мажорную и минорную версии при помощи последовательности ~>:

```
gem 'rails', '~> 5.1'
```

В примере выше допускается версия Rails от 5.1.0 до 5.2.0, не включая последнюю.

Установка гема slim-rails

На прошлом уроке мы говорили про шаблонизаторы. Давайте подключим альтернативный шаблонизатор slim-rails. Для этого в файле манифеста Gemfile мы должны разместить объявление:

```
gem 'slim-rails'
```

Чтобы установить гем, необходимо выполнить команду bundle install (или просто bundle):

```
bundle install
```

В результате этой команды будет установлен гем slim-rails и внесены изменения в файл Gemfile.lock. Это сделает установленную нами версию доступной всем остальным разработчикам, когда изменения будут заброшены в git-репозиторий. После установки новых зависимостей сервер необходимо перезапустить.

Конструкция require

При изучении языка Ruby все файлы и расширения, которые не входят в ядро языка Ruby, подключаются при помощи конструкции require. В Ruby on Rails почти никогда не придется прибегать к этой конструкции. Дело в том, что гем bundler автоматически подключает все гемы. Если заглянуть в файл config/application.rb, обнаружим следующую строку:

```
Bundler.require(*Rails.groups)
```

Именно она несет ответственность за подключение гемов к Rails-приложению.

Rake-задачи

Утилита rails поддерживает несколько типов задач, которые условно можно разделить на:

- rails-задачи генерации кода, запуска приложения и консоли;
- rake-задачи по обслуживанию приложения.

Чтобы познакомиться со списком задач, в корне проекта следует запустить команду rails без параметров:

```
rails
...
Rails:
  console
  dbconsole
```

```
  new
  ...
  server
  test
  version

Rake:
  about
  app:template
  app:update
  assets:clean[keep]
  assets:clobber
  assets:environment
  assets:precompile
  cache_digests:dependencies
  cache_digests:nested_dependencies
  db:create
  db:drop
  ...
  webpacker:yarn_install[arg1,arg2]
  yarn:install
```

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, который выдается командой rails --help, не является постоянным: команды могут быть добавлены гемами и самим разработчиком. Поэтому начинать знакомство с проектом нужно со списка доступных команд.

Rake-задачи, как правило, имеют описание, ознакомиться с которым можно, воспользовавшись командой rails --task или rails -T

```
rails -T
rails about                                 # List versions of all Rails
frameworks and the environment
rails app:template                          # Applies the template supplied by
LOCATION=(/path/to/template) or URL
rails app:update                            # Update configs and some other
initially generated files
rails assets:clean[keep]                    # Remove old compiled assets
rails assets:clobber                        # Remove compiled assets
rails assets:environment                    # Load asset compile environment
rails assets:precompile                     # Compile all the assets named in
config.assets.precompile
...
rails webpacker:install:react               # Install everything needed for React
rails webpacker:install:vue                 # Install everything needed for Vue
rails webpacker:verify_install              # Verifies if Webpacker is installed
rails webpacker:yarn_install[arg1,arg2]  # Support for older Rails versions
rails yarn:install                                    # Install all JavaScript
dependencies as specified via Yarn
```

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’.

test.rake

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

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
```

Теперь 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. Причем задачи могут зависеть от других задач. Например, выше мы указали, что задача hello зависит от задачи environment, которая загружает полностью все классы RoR-приложения. Таким образом, внутри задачи можно обращаться ко всем остальным классам, например моделям.

Генераторы

При создании проекта командой rails new Ruby on Rails не только выполняет первоначальное развертывание приложения-заглушки, но и позволяет осуществлять генерацию моделей, контроллеров, представлений и т.п. при дальнейшей разработке. Для этого предназначена специальная команда – rails generate. Чтобы узнать ее возможности, команду следует запустить без дополнительных параметров:

```
rails generate
...
Rails:
```

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

На протяжении всего курса мы будем интенсивно использовать генераторы. Чтобы получить описание генератора и его параметры, нужно расширить команду rails generate именем генератора:

```
rails generate task
...
Example:
      `rails generate task feeds fetch erase add`

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

В каждом детальном описании приводится пример использования генератора в разделе Example. Так, выше приводится пример создания заготовок для трех rake-задач: feeds:fetch, feeds:erase и feeds:add.

Поведение генераторов можно настроить. Для этого в конфигурационный файл config/environments/development.rb помещают вызов метода generators. В его блоке настраивают, какие файлы будут создаваться при вызове генераторов.

config/environments/development.rb

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

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

Стартовая страница

Давайте создадим главную страницу. Для этого воспользуемся генератором контроллера (controller).

Контроллер нужен для обработки запросов к приложению – это точка входа в схеме MVC. С помощью *роутинга* определяется, какой запрос каким контроллером будет обрабатываться. Контроллер собирает информацию и формирует ответ за счет рендеринга представления (view). Задача представления – отображение этой информации в нужном формате.

Rails-контроллер – это класс, в котором определены методы – экшены (действия). Последними сопоставляются роуты.

С помощью команды rails generate controller можно создать контроллер и представления. Наберите команду rails generate controller без аргументов, чтобы посмотреть краткую справку по использованию.

```
rails generate controller
...
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
```

Давайте создадим контроллер Home и экшен index для отображения главной страницы приложения:

```
rails generate controller Home index
  create  app/controllers/home_controller.rb
   route  get 'home/index'
  invoke  slim
  create    app/views/home
  create    app/views/home/index.html.slim
  invoke  assets
  invoke    js
  invoke    scss
```

Как видно, rails generate controller сгенерировал контроллер, роут и представление.

Роутинг

Откроем файл config/routes.rb. В этом файле задаются роуты (сопоставление запроса контроллеру) приложения. Сейчас его содержимое должно выглядеть следующим образом:

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

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

Можно предоставить доступ к этой странице по произвольному запросу. Для этого необходимо передать дополнительный параметр to: в метод get. Например, мы хотим видеть эту страницу по адресу http://localhost:3000/ololo/home/index. Тогда мы должны изменить (или добавить новый) метод get следующим образом:

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

Чтобы установить экшен по умолчанию, который отвечает за GET-запрос к корню сайта “/”, Rails предоставляет специальный метод root:

```
Rails.application.routes.draw do
  root 'home#index'
end
```

Теперь запустим сервер командой rails s и проверим, что все работает, перейдя по адресу http://localhost:3000/.

Шаблоны

Шаблоны в Rails реализуют компонент представления View из паттерна MVC. Располагаются шаблоны в директории app/views:

```
app/views
├── home
│   └── index.html.slim
└── layouts
    ├── application.html.erb
    ├── mailer.html.erb
    └── mailer.text.erb
```

Лейауты

Помимо app/views/index/show.html.slim есть директория app/views/layouts. В ней содержатся базовые шаблоны для представления. На каждой странице так или иначе у нас содержится одинаковый код в html: <head>, а также, возможно, футер, хедер, меню или что-то ещё. Но в программировании есть принцип DRY – Don’t Repeat Yourself или *не повторяйся*. Поэтому были придуманы лэйауты. Давайте заглянем внутрь application.html.erb.

application.html.erb

```
<!DOCTYPE html>
```

```
<html>
  <head>
    <title>Blog</title>
    <%= csrf_meta_tags %>
    <%= stylesheet_link_tag    'application', media: 'all' %>
    <%= javascript_include_tag 'application' %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

В тегах <%= %> в erb-файлах заключен вызов методов ruby. С чем-то похожим вы могли сталкиваться, если изучали PHP. Важно обратить внимание на вызов метода yield. В данном случае этот метод возвращает результат вычисления блока в подсистеме представления. Таким образом, если мы хотим отрендерить index.html.slim, результат рендеринга этого представления будет подставлен в лэйаут на место вызова функции yield.

Гем html2slim

Поскольку мы хотим уйти от xml-подобного html-синтаксиса, мы установили в зависимость гем slim-rails. Давайте заменим файл application.html.erb на application.html.slim.

application.html.slim

```
doctype html
html
  head
    title Blog
    = csrf_meta_tags
    = stylesheet_link_tag 'application', media: 'all'
    = javascript_include_tag 'application'
  body
    = yield
```

Чтобы сделать это автоматически, воспользуемся гемом html2slim, который можно поставить следующей командой:

```
gem install html2slim
```

После этого в консоли доступна команда html2slim, которой следует указать путь к папке с erb-файлами. Команда рекурсивно обойдет все вложенные папки и создаст slim-эквиваленты для erb-шаблонов. Чтобы старые erb-шаблоны были удалены, необходимо добавить параметр -d:

```
erb2slim . -d
```

В результате выполнения команды структура папки app/view должна выглядеть следующим образом:

```
app/views
```

```
├── home
│   └── index.html.slim
└── layouts
    ├── application.html.slim
    ├── mailer.html.slim
    └── mailer.text.slim
```

Контроллеры

В Rails контроллеры хранятся в папке app/controllers. После создания нового rails-приложения в этой папке уже находится один файл с одноименным классом application_controller.rb. Это базовый контроллер, который будет родителем для всех будущих контроллеров.

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
end
```

Сессии и CSRF-атаки

В классе application_controller.rb предлагается реализовывать общее поведение для всех контроллеров по умолчанию. Например, для каждого контроллера будет вызван метод protect_from_forgery. Этот метод защищает приложение от CSRF-атак и обеспечивает надежность определения принадлежности пользователя сессии. Вы могли обратить внимание на строку = csrf_meta_tags в шаблоне, которая генерирует специальный токен для работы с этим методом.

CSRF-атака основана на межсайтовых запросах. Дело в том, что протокол HTTP не поддерживает сессии: каждое новое обращение в рамках него выглядит как обращение нового клиента. Не важно, что перед этим клиент загрузил главную страницу и теперь загружает изображения, на которые ссылается HTML-страница. Для классического HTTP-сервера эти запросы никак не связаны между собой.

Для того, чтобы отличать пользователей друг от друга, вводится понятие сессии. Браузер пользователя записывает имя сессии в cookie и при каждом запросе отправляет ее серверу. Приложение ориентируется на уникальный идентификатор, присланный с запросом, и предоставляет по нему доступ к данным сессии.

Сессия приобретает особое значение, если пользователю доступны какие-то действия на сайте, например удаление аккаунта, смена пароля, перевод денежных средств. Если пользователь аутентифицирован на сайте, ему обманным путем может предоставляться ссылка на ресурс, выполняющий действия, которые пользователь не хочет выполнять. Переход по такой ссылке может приводить к нежелательным действиям.

Именно для этого Ruby on Rails вводит CSRF-токен. Каждая страница, каждая HTML-форма помечается уникальным токеном и выполнить действия на этой странице можно, только прислав такой же токен с HTTP-запросом. Это не панацея, но значительно затрудняет мошеннические операции.

Подсчет количества посещений

После генерации контроллера был создан файл app/controllers/home_controller.rb с одноименным классом − наследником класса ApplicationController с единственным методом index.

app/controllers/home_controller.rb

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

Именно к этому методу контроллера мы настроили роут по умолчанию. После работы этого метода Rails вернет представление с соответствующим названием. У нас такое представление как раз есть по пути app/views/home/index.html.slim. Давайте на стороне контроллера сделаем счетчик посещений данной страницы, храня его значение в сессии, для управления которой Rails предоставляет хэш session.

app/controllers/home_controller.rb

```
class HomeController < ApplicationController
  def index
    session[:times_here] ||= 0
    session[:times_here] += 1
  end
end
```

app/views/home/index.html.slim

```
h1 Welcome to my Blog
p wow, you visit that page #{session[:times_here]} times
```

Теперь, перейдя на главную страницу и обновив ее несколько раз, мы увидим, как изменяется счетчик посещений страницы. При этом никаких данных на стороне клиента не хранится. Посмотрите куки через браузер и увидите там только идентификатор сессии.

Соглашения

Чтобы контроллер отрендерил представление и отправил результат клиенту, в экшене следовало бы разместить вызов метода render:

```
class HomeController < ApplicationController
  def index
    session[:times_here] ||= 0
    session[:times_here] += 1
    render 'home/index'
  end
end
```

Однако по соглашениям, принятым в Ruby on Rails, если имя папки, в которой располагается представление, совпадает с именем контроллера, а имя экшена − с именем файла представления, явный вызов метода render можно опускать. Если же код можно сократить за счет неявных соглашений, rails-разработчики всегда используют такую возможность.

Обратите также внимание, что контроллер HomeController располагается в файле с именем home_controller.rb. Если файл переименовать, Ruby on Rails уже не сможет найти класс HomeController − автозагрузчик классов его не обнаружит. Имена классов в Ruby принято называть в соответствии со CamelCase-стилем. При этом имена файлов должны быть названы в соответствии со snake_case-стилем.

Сохранение результатов в Git

Сейчас время поместить наработки в систему контроля версий Git:

```
git add --all
git commit -m 'Main page'
```

Домашнее задание

1. Оформите домашнюю страницу и базовый шаблон, не используя JavaScript:
    a. В файле базового шаблона, как упоминалось, может содержаться еще и хедер с футером приложения. Добавьте хедер и футер на свой вкус, используйте файлы стилей.
    b. Заполните стартовую страницу приложения чем-нибудь статическим.
2. Создайте статическую страницу, например «обо мне». Настройте роут на нее и добавьте ссылку с главной страницы.
3. Познакомьтесь с командой rails routes.
4. *Создайте rake-задачу, которая подсчитывает количество ruby-файлов в проекте.
5. *Создайте rake-задачу, которая подсчитывает количество строк в ruby-файлах проекта.

    «*» задание повышенной сложности (по желанию)

Дополнительные материалы

1. Подробнее про структуру директорий и создание проекта.
2. Философия Ruby on Rails.
3. Установка GIT.
4. Как работают сессии в rails.
5. Википедия. CSRF.
6. Документация RubyGems.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Майкл Хартл. Ruby on Rails для начинающих. Изучаем разработку веб-приложений на основе Rails.
2. Ruby on Rails Guides.
3. Obie Fernandez. The Rails 5 Way.
4. HTTP: The Definitive Guide. David Gourley and Brian Totty
5. Документация RubyGems.
6. Генераторы.