

Ruby on Rails по-русски

Добро пожаловать!

Не секрет, что в Интернете есть множество ресурсов, посвященных Ruby on Rails, однако в большинстве случаев они англоязычные, а на русском языке очень мало подробной качественной информации об этой среде разработки.

На этом сайте выложены переводы официального руководства по Rails. Надеюсь, это руководство позволит вам немедленно приступить к использованию Rails и поможет разобраться, что и как там работает.

В свободное время переводы актуализируются и добавляются. Код проекта и тексты переводов открыты и размещены [на Гитхабе](#). Желающим помочь — велкам! Форкайте, предлагайте изменения, вносите их, отправляйте пул-реквесты!

Это перевод [Ruby on Rails Guides](#) для версии Rails 3.2. Переводы для ранних версий доступны в архиве или на гитхабе:

- [Rails 3.1](#)
- [Rails 3.0](#)
- [Rails 2.3](#)

Приступим!

С чего начать?

[Rails для начинающих](#)

Все, что вы должны знать, чтобы установить Rails и создать свое первое приложение.

Модели

[Миграции базы данных Rails](#)

Это руководство раскрывает, как вы должны использовать миграции Active Record, чтобы привести свою базу данных к структурированной и организованной форме.

[Валидации и обратные вызовы \(колбэки\) Active Record](#)

Это руководство раскрывает, как вы можете применять валидации и обратные вызовы Active Record.

[Связи \(ассоциации\) Active Record](#)

Это руководство раскрывает все связи, предоставленные Active Record.

[Интерфейс запросов Active Record](#)

Это руководство раскрывает интерфейс запросов к базе данных, предоставленный Active Record.

Вьюхи

[Макеты и рендеринг в Rails](#)

Это руководство раскрывает основы возможностей макетов Action Controller и Action View, включая рендеринг и перенаправление, использование содержимого для блоков и работу с частичными шаблонами.

[Хелперы форм Action View](#)

Руководство по использованию встроенных хелперов форм.

Контроллеры

[Обзор Action Controller](#)

Это руководство раскрывает, как работают контроллеры, и как они вписываются в цикл запроса к вашему приложению. Оно включает сессии, фильтры, куки, потоковые данные, работу с исключениями, вызванными запросами, и другие статьи.

[Роутинг Rails](#)

Это руководство раскрывает открытые для пользователя функции роутинга. Если хотите понять, как использовать роутинг в вашем приложении на Rails, начните отсюда.

Копаем глубже

[Расширения ядра Active Support](#)

Это руководство документирует расширения ядра Ruby, определенные в Active Support.

[Rails Internationalization API](#)

Это руководство раскрывает, как добавить интернационализацию в ваше приложение. Ваше приложение будет способно переводить содержимое на разные языки, изменять правила образования множественного числа, использовать правильные форматы дат для каждой страны и так далее.

[Основы Action Mailer](#)

Это руководство описывает, как использовать Action Mailer для отправки и получения электронной почты.

[Тестирование приложений на Rails](#)

Это достаточно полное руководство по осуществлению юнит- и функциональных тестов в Rails. Оно раскрывает все от "Что такое тест?" до тестирования API. Наслаждайтесь.

[Безопасность приложений на Rails](#)

Это руководство описывает общие проблемы безопасности в приложениях веб, и как избежать их в Rails.

[Отладка приложений на Rails](#)

Это руководство описывает, как отлаживать приложения на Rails. Оно раскрывает различные способы достижения этого, и как понять что произошло “за кулисами” вашего кода.

[Тестирование производительности приложений на Rails](#)

Это руководство раскрывает различные способы тестирования производительности приложения на Ruby on Rails.

[Конфигурирование приложений на Rails](#)

Это руководство раскрывает основные конфигурационные настройки для приложения на Rails.

[Руководство по командной строке Rails и задачам Rake](#)

Это руководство раскроет инструменты командной строки и задачи rake, предоставленные Rails.

[Кэширование с Rails](#)

Различные техники кэширования, предоставленные Rails.

[Asset Pipeline](#)

Это руководство документирует файлопровод (asset pipeline)

[Engine для начинающих](#)

Это руководство объясняет, как написать монтируемый engine

Rails для начинающих

Это руководство раскрывает установку и запуск Ruby on Rails. После его прочтения, вы будете ознакомлены:

- С установкой Rails, созданием нового приложения на Rails и присоединением Вашего приложения к базе данных
- С общей структурой приложения на Rails
- С основными принципами MVC (Model, View Controller – «Модель-представление-контроллер») и дизайна, основанного на RESTful
- С тем, как быстро создать изначальный код приложения на Rails.

Это руководство основывается на Rails 3.1. Часть кода, показанного здесь, не будет работать для более ранних версий Rails. Руководства для начинающих, основанные на Rails 3.0 и 2.3 вы можете просмотреть [в архиве](#)

Допущения в этом руководстве

Это руководство рассчитано на новичков, которые хотят запустить приложение на Rails с нуля. Оно не предполагает, что вы раньше работали с Rails. Однако, чтобы полноценно им воспользоваться, необходимо предварительно установить:

- Язык [Ruby](#) версии 1.8.7 или выше

Отметьте, что в Ruby 1.8.7 p248 и p249 имеются баги, роняющие Rails 3.0. Хотя для Ruby Enterprise Edition их исправили, начиная с релиза 1.8.7-2010.02. На фронте 1.9, Ruby 1.9.1 не стабилен, поскольку он явно ломает Rails 3.0, поэтому если хотите использовать Rails 3 с 1.9.x, переходите на 1.9.2 для гладкой работы.

- Систему пакетов [RubyGems](#)
 - Если хотите узнать больше о RubyGems, прочитайте [RubyGems User Guide](#)
- Рабочую инсталляцию [SQLite3 Database](#)

Rails – фреймворк для веб-разработки, написанный на языке программирования Ruby. Если у вас нет опыта в Ruby, возможно вам будет тяжело сразу приступить к изучению Rails. Есть несколько хороших англоязычных ресурсов, посвященных изучению Ruby, например:

- [Mr. Neighborly's Humble Little Ruby Book](#)
- [Programming Ruby](#)
- [Why's \(Poignant\) Guide to Ruby](#)

Из русскоязычных ресурсов, посвященных изучению Ruby, я бы выделил следующие:

- [Викиучебник по Ruby](#)
- [Руководство по Руби на 1 странице](#)
- [Учебник 'Учись программировать', автор Крис Пайн](#)

Кроме того, код примера для этого руководства доступен в [репозитории rails на github](#) в rails/railties/guides/code/getting_started.

Что такое Rails?

Этот раздел посвящен подробностям предпосылок и философии фреймворка Rails. Его можно спокойно пропустить и вернуться к нему позже. [Следующий раздел](#) направит вас на путь создания собственного первого приложения на Rails.

Rails – фреймворк для веб-разработки, написанный на языке программирования Ruby. Он разработан, чтобы сделать

программирование веб-приложений проще, так как использует ряд допущений о том, что нужно каждому разработчику для создания нового проекта. Он позволяет вам писать меньше кода в процессе программирования, в сравнении с другими языками и фреймворками. Профессиональные разработчики на Rails также отмечают, что с ним разработка веб-приложений более забавна =)

Rails – своеобразный программный продукт. Он делает предположение, что имеется “лучший” способ что-то сделать, и он так разработан, что стимулирует этот способ – а в некоторых случаях даже препятствует альтернативам. Если изучите “The Rails Way”, то, возможно, откроете в себе значительное увеличение производительности. Если будете упорствовать и переносить старые привычки с других языков в разработку на Rails, и попытаетесь использовать шаблоны, изученные где-то еще, ваш опыт разработки будет менее счастливым.

Философия Rails включает несколько ведущих принципов:

- DRY – “Don’t Repeat Yourself” – означает, что написание одного и того же кода в разных местах – это плохо.
- Convention Over Configuration – означает, что Rails сам знает, что вы хотите и что собираетесь делать, вместо того, чтобы заставлять вас по мелочам править многочисленные конфигурационные файлы.
- REST – лучший шаблон для веб-приложений – организация приложения вокруг ресурсов и стандартных методов HTTP это быстрый способ разработки.

Архитектура MVC

Rails организован на архитектуре Model, View, Controller, обычно называемой MVC. Преимущества MVC следующие:

- Отделяется бизнес-логика от пользовательского интерфейса
- Легко хранить неповторяющийся код DRY
- Легко обслуживать приложение, так как ясно, в каком месте содержится тот или иной код

Модели

Модель представляет собой информацию (данные) приложения и правила для обработки этих данных. В случае с Rails, модели в основном используются для управления правилами взаимодействия с таблицей базы данных. В большинстве случаев, одна таблица в базе данных соответствует одной модели вашего приложения. Основная масса бизнес логики вашего приложения будет сконцентрирована в моделях.

Представления

Представления (на жаргоне “вьюхи”) представляют собой пользовательский интерфейс Вашего приложения. В Rails представления часто являются HTML файлами с встроенным кодом Ruby, который выполняет задачи, связанные исключительно с представлением данных. Представления справляются с задачей предоставления данных веб-браузеру или другому инструменту, который может использоваться для обращения к Вашему приложению.

Контроллеры

Контроллеры “склеивают” вместе модели и представления. В Rails контроллеры ответственны за обработку входящих запросов от веб-браузера, запрос данных у моделей и передачу этих данных во вьюхи для отображения.

Компоненты Rails

Rails строится на многих отдельных компонентах. Каждый из этих компонентов кратко описан ниже. Если вы новичок в Rails, не закливайтесь на подробностях каждого компонента, так как они будут детально описаны позже. Для примера, в руководстве мы создадим приложение Rack, но вам не нужно ничего знать, что это такое, чтобы продолжить изучение руководства.

- Action Pack
 - Action Controller
 - Action Dispatch
 - Action View
- Action Mailer
- Active Model
- Active Record
- Active Resource
- Active Support
- Railties

Action Pack

Action Pack это отдельный гем, содержащий Action Controller, Action View и Action Dispatch. Буквы “VC” в аббревиатуре “MVC”.

Action Controller

Action Controller это компонент, который управляет контроллерами в приложении на Rails. Фреймворк Action Controller обрабатывает входящие запросы к приложению на Rails, извлекает параметры и направляет их в предназначенный экшн (action). Сервисы, предоставляемые Action Controller-ом включают управление сессиями, рендеринг шаблонов и управление перенаправлениями.

Action View

Action View управляет представлениями в вашем приложении на Rails. На выходе по умолчанию создается HTML или XML. Action View управляет рендерингом шаблонов, включая вложенные и частичные шаблоны, и содержит встроенную поддержку AJAX. Шаблоны вых более детально раскрываются в другом руководстве, [Макеты и рендеринг в Rails](#)

Action Dispatch

Action Dispatch управляет маршрутизацией веб запросов и рассылкой их так, как вы желаете, или к вашему приложению, или к любому другому приложению Rack. Приложения Rack – это продвинутая тема, раскрыта в отдельном руководстве, [Rails on Rack](#)

Action Mailer

Action Mailer это фреймворк для встроенных служб e-mail. Action Mailer можно использовать, чтобы получать и обрабатывать входящую электронную почту, или чтобы рассылать простой текст или или сложные multipart электронные письма, основанные на гибких шаблонах.

Active Model

Active Model предоставляет определенный интерфейс между службами гема Action Pack и гемами Object Relationship Mapping, такими как Active Record. Active Model позволяет Rails использовать другие фреймворки ORM вместо Active Record, если так нужно вашему приложению.

Active Record

Active Record это основа для моделей в приложении на Rails. Он предоставляет независимость от базы данных, базовый CRUD-функционал, расширенные возможности поиска и способность устанавливать связи между моделями и модели с другим сервисом.

Active Resource

Active Resource представляет фреймворк для управления соединением между бизнес-объектами и веб-сервисами на основе RESTful. Он реализует способ привязки веб-ресурсов к локальным объектам с семантикой CRUD.

Active Support

Active Support это большая коллекция полезных классов и расширений стандартных библиотек Ruby, которые могут быть использованы в Rails, как в ядре, так и в вашем приложении.

Railties

Railties это код ядра Rails, который создает новые приложения на Rails и соединяет разные фреймворки и плагины вместе в любом приложении на Rails.

REST

Rest обозначает Representational State Transfer и основан на архитектуре RESTful. Как правило, считается, что она началась с докторских тезисов Roy Fielding, [Architectural Styles and the Design of Network-based Software Architectures](#). Хотя можно и почитать эти тезисы, REST в терминах Rails сводится к двум главным принципам в своих целях:

- Использование идентификаторов ресурса, таких как URL, чтобы представлять ресурсы
- Передача представлений о состоянии этого ресурса между компонентами системы.

Например, запрос HTTP:

```
DELETE /photos/17
```

будет воспринят как ссылка на ресурс photo с идентификатором ID 17, и желаемым действием – удалить этот ресурс. REST это естественный стиль для архитектуры веб-приложений, и Rails ограждает Вас от некоторых сложностей RESTful и причуд браузера.

Если Вы хотите побольше узнать о REST, как о стиле архитектуры, эти англоязычные ресурсы более подходящие, чем тезисы Fielding:

- [A Brief Introduction to REST](#) by Stefan Tilkov

- [An Introduction to REST](#) (video tutorial) by Joe Gregorio
- [Representational State Transfer](#) article in Wikipedia
- [How to GET a Cup of Coffee](#) by Jim Webber, Savas Parastatidis & Ian Robinson

На русском языке могу посоветовать только [Введение в службы RESTful с использованием WCF](#) Джона Фландерса.

Создание нового проекта Rails

Лучший способ использования этого руководства – проходить каждый шаг и смотреть, что получится, пропустите код или шаг и учебное приложение не заработает, поэтому следует буквально все делать шаг за шагом. Можно получить законченный код [здесь](#).

Следуя этому руководству, вы создадите проект Rails с названием blog, очень простой веб-блог. Прежде чем начнем создавать приложение, нужно убедиться, что сам Rails установлен.

Нижеследующие примеры используют # и \$ для обозначения строки ввода терминала. Если вы используете Windows, ваша строка будет выглядеть наподобие `c:\source_code>`

Чтобы проверить, что все установлено верно, должно запускаться следующее:

```
$ rails --version
```

Если выводится что-то вроде “Rails 3.1.3”, можно продолжать.

Установка Rails

В основном, самый простой способ установить Rails это воспользоваться возможностями RubyGems:

```
Просто запустите это как пользователь root:  
# gem install rails
```

Если вы работаете в Windows, тогда можно быстро установить Ruby и Rails, используя [Rails Installer](#).

Создание приложения Blog

Чтобы начать, откройте терминал, войдите в папку, в которой у вас есть права на создание файлов и напишите:

```
$ rails new blog
```

Это создаст приложение на Rails с именем Blog в директории blog.

Можно посмотреть все возможные ключи, которые принимает билдер приложения на Rails, запустив `rails new -h`.

После того, как вы создали приложение blog, перейдите в его папку, чтобы продолжить работу непосредственно с этим приложением:

```
$ cd blog
```

Команда ‘rails new blog’, запущенная ранее, создаст папку в вашей рабочей директории, названную blog. В папке blog имеется несколько автоматически созданных папок, задающих структуру приложения на Rails. Большая часть работы в этом самоучителе будет происходить в папке `app/`, но сейчас пробежимся по функциям каждой папки, которые создает Rails в новом приложении по умолчанию:

Файл/Папка	Цель
<code>app/</code>	Содержит контроллеры, модели и вьюхи вашего приложения. Мы рассмотрим эту папку подробнее далее.
<code>config/</code>	Конфигурации правил, маршрутов, базы данных вашего приложения, и т.д. Более подробно это раскрыто в Конфигурирование приложений на Rails
<code>config.ru</code>	Конфигурация Rack для серверов, основанных на Rack, используемых для запуска приложения.
<code>db/</code>	Содержит текущую схему вашей базы данных, а также миграции базы данных.
<code>doc/</code>	Углубленная информация по вашему приложению.
<code>Gemfile</code> <code>Gemfile.lock</code>	Эти файлы позволяют определить, какие нужны зависимости от гемов для вашего приложения на Rails.
<code>lib/</code>	Внешние модули для вашего приложения.
<code>log/</code>	Файлы логов приложения.
<code>public/</code>	Единственная папка, которая доступна извне как есть. Содержит статичные файлы и скомпилированные ресурсы.
<code>Rakefile</code>	Этот файл содержит набор команд, которые могут быть запущены в командной строке. Определения команд производятся во всех компонентах Rails. Вместо изменения Rakefile, вы можете добавить свои собственные задачи, добавив файлы в директорию <code>lib/tasks</code> вашего приложения.
<code>README.md</code>	Это вводный мануал для вашего приложения. Его следует отредактировать, чтобы рассказать остальным,

README.rdoc	Что ваше приложение делает, как его настроить, и т.п.
script/	Содержит скрипт rails, который запускает ваше приложение, и может содержать другие скрипты, используемые для развертывания или запуска вашего приложения.
test/	Юнит-тесты, фикстуры и прочий аппарат тестирования. Это раскрывается в руководстве Тестирование приложений на Rails
tmp/	Временные файлы
vendor/	Место для кода внешних разработчиков. В типичном приложении на Rails, включает Ruby Gems, исходный код Rails (если вы опционально установили его в свой проект) и плагины, содержащие дополнительную упакованную функциональность.

Конфигурирование базы данных

Почти каждое приложение на Rails взаимодействует с базой данных. Какую базу данных использовать, определяется в конфигурационном файле config/database.yml. Если вы откроете этот файл в новом приложении на Rails, то увидите базу данных по умолчанию, настроенную на использование SQLite3. По умолчанию, файл содержит разделы для трех различных сред, в которых может быть запущен Rails:

- Среда development используется на вашем рабочем/локальном компьютере для того, чтобы вы могли взаимодействовать с приложением.
- Среда test используется при запуске автоматических тестов.
- Среда production используется, когда вы развертываете свое приложения во всемирной сети для использования.

Вам не нужно обновлять конфигурации баз данных вручную. Если взглянете на опции генератора приложения, то увидите, что одна из опций называется -database. Эта опция позволяет выбрать адаптер из списка наиболее часто используемых СУБД. Вы даже можете запускать генератор неоднократно: `cd .. && rails new blog --database=mysql`. После того, как подтвердите перезапись config/database.yml, ваше приложение станет использовать MySQL вместо SQLite. Подробные примеры распространенных соединений с базой данных указаны ниже.

Конфигурирование базы данных SQLite3

В Rails есть встроенная поддержка [SQLite3](#), являющейся легким несерверным приложением по управлению базами данных. Хотя нагруженная среда production может перегрузить SQLite, она хорошо работает для разработки и тестирования. Rails при создании нового проекта использует базу данных SQLite, но Вы всегда можете изменить это позже.

Вот раздел дефолтного конфигурационного файла (config/database.yml) с информацией о соединении для среды development:

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

В этом руководстве мы используем базу данных SQLite3 для хранения данных, поскольку эта база данных работает с нулевыми настройками. Rails также поддерживает MySQL и PostgreSQL “из коробки”, и имеет плагины для многих СУБД. Если Вы уже используете базу данных в работе, в Rails скорее всего есть адаптер для нее.

Конфигурирование базы данных MySQL

Если Вы выбрали MySQL вместо SQLite3, Ваш config/database.yml будет выглядеть немного по другому. Вот секция development:

```
development:
  adapter: mysql2
  encoding: utf8
  database: blog_development
  pool: 5
  username: root
  password:
  socket: /tmp/mysql.sock
```

Если на вашем компьютере установленная MySQL имеет пользователя root с пустым паролем, эта конфигурация у Вас заработает. В противном случае измените username и password в разделе development как следует.

Конфигурирование базы данных PostgreSQL

Если Вы выбрали PostgreSQL, Ваш config/database.yml будет модифицирован для использования базы данных PostgreSQL:

```
development:
  adapter: postgresql
  encoding: unicode
  database: blog_development
  pool: 5
  username: blog
```

```
password:
```

Конфигурирование базы данных SQLite3 для платформы JRuby

Если вы выбрали SQLite3 и используете JRuby, ваш config/database.yml будет выглядеть немного по-другому. Вот секция development:

```
development:
  adapter: jdbcsqlite3
  database: db/development.sqlite3
```

Конфигурирование базы данных MySQL для платформы JRuby

Если вы выбрали MySQL и используете JRuby, ваш config/database.yml будет выглядеть немного по-другому. Вот секция development:

```
development:
  adapter: jdbcmysql
  database: blog_development
  username: root
  password:
```

Конфигурирование базы данных PostgreSQL для платформы JRuby

Наконец, если вы выбрали PostgreSQL и используете JRuby, ваш config/database.yml будет выглядеть немного по-другому. Вот секция development:

```
development:
  adapter: jdbcpostgresql
  encoding: unicode
  database: blog_development
  username: blog
  password:
```

Измените username и password в секции development как следует.

Создание базы данных

Теперь, когда вы конфигурировали свою базу данных, пришло время позволить Rails создать для вас пустую базу данных. Это можно сделать, запустив команду rake:

```
$ rake db:create
```

Это создаст базы данных SQLite3 development и test в папке db/.

Rake это одна из основных консольных команд, которую Rails использует для многих вещей. Можно посмотреть список доступных команд rake в своем приложении, запустив rake -T.

Hello, Rails!

Одним из традиционных мест начала изучения нового языка является быстрый вывод на экран какого-либо текста, чтобы это сделать, нужен запущенный сервер вашего приложения на Rails.


Запуск веб-сервера

Фактически у вас уже есть функциональное приложение на Rails. Чтобы убедиться, нужно запустить веб-сервер на вашей машине. Это можно осуществить, запустив:

```
$ rails server
```

Компилирование CoffeeScript в JavaScript требует JavaScript runtime, и его отсутствие приведет к ошибке execjs. Обычно Mac OS X и Windows поставляются с установленным JavaScript runtime. therubyracer and therubyrhino — обыкновенно используемые runtime для Ruby и JRuby соответственно. Также можно посмотреть список runtime-ов в [ExecJS](#).

По умолчанию это запустит экземпляр веб-сервера WEBrick (Rails может использовать и некоторые другие веб-серверы). Чтобы увидеть приложение в действии, откройте окно браузера и пройдите по адресу <http://localhost:3000>. Вы должны увидеть дефолтную информационную страницу Rails:



Welcome aboard

You're riding Ruby on Rails!

[About your application's environment](#)

Getting started

Here's how to get rolling:

1. Use `rails generate` to create your models and controllers
To see all available options, run it without parameters.
2. Set up a default route and remove or rename this file
Routes are set up in `config/routes.rb`.
3. Create your database
Run `rake db:migrate` to create your database. If you're not using SQLite (the default), edit `config/database.yml` with your username and password.

the Rails site

Join the community

[Ruby on Rails](#)
[Official weblog](#)
[Wiki](#)

Browse the documentation

[Rails API](#)
[Ruby standard library](#)
[Ruby core](#)
[Rails Guides](#)

Чтобы остановить веб-сервер, нажмите Ctrl+C в терминале, где он запущен. В режиме development, Rails в основном не требует остановки сервера; все изменения, которые Вы делаете в файлах, автоматически подхватываются сервером.

Страница "Welcome Aboard" это своеобразный тест для нового приложения на Rails: она показывает, что ваши программы настроены достаточно правильно для отображения страницы. Также можете нажать по ссылке *About your application's environment* чтобы увидеть сводку о среде вашего приложения.

Скажите "привет", Рельсы

Чтобы Rails сказал "Привет", нужно создать, как минимум, контроллер и вьюху. К счастью, это можно сделать одной командой. Введите эту команду в вашем терминале:

```
$ rails generate controller home index
```

Если появляется ошибка, что команда не найдена, необходимо явно передать команду Rails rails в Ruby: `ruby \path\to\rails generate controller home index`.

Rails создаст несколько файлов, включая `app/views/home/index.html.erb`. Это шаблон, который используется для отображения результатов экшна (метода) `index` контроллера `home`. Откройте этот файл в текстовом редакторе и отредактируйте его, чтобы он содержал одну строчку кода:

```
<h1>Hello, Rails!</h1>
```

Настройка домашней страницы приложения

Теперь, когда мы сделали контроллер и вьюху, нужно сказать Rails, что мы хотим увидеть "Hello Rails!". В нашем случае мы хотим это увидеть, когда зайдём в корневой URL нашего сайта, <http://localhost:3000>, вместо тестовой "Welcome Aboard".

Первым шагом осуществления этого является удаление дефолтной страницы из вашего приложения:

```
$ rm public/index.html
```

Это нужно сделать, так как Rails предпочитает доставлять любой статичный файл из директории `public` любому динамическому содержимому, создаваемому из контроллеров.

Теперь нужно сказать Rails, где находится настоящая домашняя страница. Откройте файл `config/routes.rb` в редакторе. Это *маршрутный файл* вашего приложения, который содержит варианты входа на сайт на специальном языке DSL (domain-specific language, предметно-ориентированный язык программирования), который говорит Rails, как соединять входящие запросы с контроллерами и экшнами. Этот файл содержит много закомментированных строк с примерами, и один из них фактически показывает, как соединить корень сайта с определенным контроллером и экшном. Найдите строку, начинающуюся с `root :to` и раскомментируйте ее. Должно получится следующее:

```
Blog::Application.routes.draw do
```



```
#...
# You can have the root of your site routed with "root"
# just remember to delete public/index.html.
root :to => "home#index"
```

root :to => "home#index" говорит Rails направить обращение к корню в экшн index контроллера home.

Теперь, если вы пройдете по адресу <http://localhost:3000> в браузере, то увидите Hello, Rails!.

Чтобы узнать больше о роутинге, обратитесь к руководству [Роутинг в Rails](#).

Создание ресурса

Разрабатываем быстро с помощью Scaffolding

“Строительные леса” Rails, *scaffolding*, это быстрый способ создания больших кусков кода приложения. Если хотите создать модели, вьюхи и контроллеры для нового ресурса одной операцией, воспользуйтесь scaffolding.

Создание ресурса

В случае с приложением блога, можно начать с генерации скаффолда для ресурса Post: это будет представлять собой отдельную публикацию в блоге. Чтобы осуществить это, напишите команду в терминале:

```
$ rails generate scaffold Post name:string title:string content:text
```

Генератор скаффолда создаст несколько файлов в Вашем приложении в разных папках, и отредактирует config/routes.rb. Вот краткое описание того, что он создаст:

Файл	Цель
db/migrate/20100207214725_create_posts.rb	Миграция для создания таблицы posts в вашей базе данных (у вашего файла будет другая временная метка)
app/models/post.rb	Модель Post
test/unit/post_test.rb	Каркас юнит-тестирования для модели posts
test/fixtures/posts.yml	Образцы публикаций для использования в тестировании
config/routes.rb	Отредактирован, чтобы включить маршрутную информацию для posts
app/controllers/posts_controller.rb	Контроллер Posts
app/views/posts/index.html.erb	Вьюха для отображения перечня всех публикаций
app/views/posts/edit.html.erb	Вьюха для редактирования существующей публикации
app/views/posts/show.html.erb	Вьюха для отображения отдельной публикации
app/views/posts/new.html.erb	Вьюха для создания новой публикации
app/views/posts/_form.html.erb	Партиал, контролирующий внешний вид и поведение форм, используемых во вьюхах edit и new
test/functional/posts_controller_test.rb	Каркас функционального тестирования для контроллера posts
app/helpers/posts_helper.rb	Функции хелпера, используемые из вьюх posts
test/unit/helpers/posts_helper_test.rb	Каркас юнит-тестирования для хелпера posts
app/assets/javascripts/posts.js.coffee	CoffeeScript для контроллера posts
app/assets/stylesheets/posts.css.scss	Каскадная таблица стилей для контроллера posts
app/assets/stylesheets/scaffolds.css.scss	Каскадная таблица стилей, чтобы вьюхи скаффолда смотрелись лучше

Хотя скаффолд позволяет быстро разрабатывать, стандартный код, который он генерирует, не всегда подходит для вашего приложения. Как правило, необходимо модифицировать сгенерированный код. Многие опытные разработчики на Rails избегают работы со скаффолдом, предпочитая писать весь или большую часть кода с нуля. Rails, однако, позволяет легко настраивать шаблоны для генерируемых моделей, контроллеров, вьюх и других источников файлов. Больше информации вы найдете в [Руководстве по созданию и настройке генераторов и шаблонов Rails](#).

Запуск миграции

Одним из продуктов команды rails generate scaffold является *миграция базы данных*. Миграции – это класс Ruby, разработанный для того, чтобы было просто создавать и модифицировать таблицы базы данных. Rails использует команды rake для запуска миграций, и возможна отмена миграции после того, как она была применена к вашей базе данных. Имя файла миграции включает временную метку, чтобы быть уверенным, что они выполняются в той последовательности, в которой они создавались.

Если Вы заглянете в файл db/migrate/20100207214725_create_posts.rb (помните, у вас файл имеет немного другое имя), вот что там обнаружите:

```
class CreatePosts < ActiveRecord::Migration
  def change
```

```
create_table :posts do |t|
  t.string :name
  t.string :title
  t.text :content

  t.timestamps
end
end
end
```

Эта миграция создает метод `change`, вызываемый при запуске этой миграции. Действие, определенное в этой миграции, также является обратимым, что означает, что Rails знает, как отменить изменения, сделанные этой миграцией, в случае, если вы решите их отменить позже. Когда вы запустите эту миграцию, она создаст таблицу `posts` с двумя строковыми столбцами и текстовым столбцом. Она также создаст два поля временных меток для отслеживания времени создания и обновления публикации. Подробнее о миграциях Rails можно прочесть в руководстве [Миграции базы данных Rails](#).

Сейчас нам нужно использовать команду `rake`, чтобы запустить миграцию:

```
$ rake db:migrate
```

Rails запустит эту команду миграции и сообщит, что он создал таблицу `Posts`.

```
== CreatePosts: migrating =====
-- create_table(:posts)
   => 0.0019s
== CreatePosts: migrated (0.0020s) =====
```

Так как вы работаете по умолчанию в среде `development`, эта команда будет применена к базе данных, определенной в секции `development` вашего файла `config/database.yml`. Если хотите запустить миграции в другой среде, например в `production`, следует явно передать ее при вызове команды: `rake db:migrate RAILS_ENV=production`.

Добавляем ссылку

Чтобы подключить контроллер `posts` к домашней странице, которую уже создали, можно добавить ссылку на ней. Откройте `/app/views/home/index.html.erb` И измените его следующим образом:

```
<h1>Hello, Rails!</h1>
<%= link_to "Мой блог", posts_path %>
```

Метод `link_to` – один из встроенных хелперов Rails. Он создает гиперссылку, на основе текста для отображения и указания куда перейти – в нашем случае путь для контроллера `posts`.

Работаем с публикациями в браузере

Теперь Вы готовы работать с публикациями. Чтобы это сделать, перейдите по адресу <http://localhost:3000> и нажмите на ссылку “Мой блог”:

Listing posts

Name Title Content

[New post](#)

Это результат рендеринга Rails вьюхи ваших публикаций `index`. Сейчас нет никаких публикаций в базе данных, но если нажать на ссылку `New Post`, вы можете создать одну. После этого вы увидите, что можете редактировать публикацию, просматривать или уничтожить ее. Вся логика и HTML были построены одной единственной командой `rails generate scaffold`.

В режиме `development` (с которым вы работаете по умолчанию), Rails перегружает ваше приложение с каждым запросом браузера, так что не нужно останавливать и перезапускать веб-сервер.

Поздравляем, вы начали свой путь по рельсам! =) Теперь настало время узнать, как это все работает.

Модель

Файл модели `app/models/post.rb` выглядит проще простого:

```
class Post < ActiveRecord::Base
end
```

Не так уж много написано в этом файле, но заметьте, что класс `Post` наследован от `ActiveRecord::Base`. `Active Record` обеспечивает огромную функциональность для Ваших моделей Rails, включая основные операции для базы данных CRUD (`Create`, `Read`, `Update`, `Destroy` – создать, читать, обновить, уничтожить), валидации данных, сложную поддержку поиска и возможность устанавливать отношения между разными моделями.

Добавляем немного валидации

Rails включает методы, помогающие проверить данные, которые вы передаете в модель. Откройте файл `app/models/post.rb` и отредактируйте:

```
class Post < ActiveRecord::Base
  validates :name, :presence => true
  validates :title, :presence => true,
              :length => { :minimum => 5 }
end
```

Эти изменения позволят быть уверенным, что все публикации имеют имя и заголовок, и что заголовок длиной как минимум пять символов. Rails может проверять разные условия в модели, включая существование или уникальность полей, их формат и существование связанных объектов. Подробнее валидации раскрыты в [Валидации и колбэки Active Record](#)

Использование консоли

Чтобы увидеть валидации в действии, можно использовать консоль. Консоль это инструмент, который позволяет запускать код Ruby в контексте вашего приложения:

```
$ rails console
```

По умолчанию, консоль изменяет вашу базу данных. Вместо этого можно запустить консоль, которая откатывает все изменения, которые вы сделали, используя `rails console —sandbox`.

После загрузки консоли можно работать с моделями вашего приложения:

```
>> p = Post.new(:content => "A new post")
=> #<Post id: nil, name: nil, title: nil,
    content: "A new post", created_at: nil,
    updated_at: nil>
>> p.save
=> false
>> p.errors.full_messages
=> ["Name can't be blank", "Title can't be blank", "Title is too short (minimum is 5 characters)"]
```

Этот код показывает создание нового экземпляра `Post`, попытку его сохранения и возврата в качестве ответа `false` (что означает, что сохранить не получилось), и просмотр ошибок публикации.

Когда закончите, напишите `exit` и нажмите `return`, чтобы вернуться в консоль.

В отличие от веб-сервера `development`, консоль автоматически не загружает код после выполнения строки. Если вы внесли изменения в свои модели (в своем редакторе) в то время, когда консоль была открыта, напишите в консоли `reload!`, чтобы консоль загрузила эти изменения.

Отображение всех публикаций

Давайте поглубже погрузимся в код Rails и увидим, как приложение отображает нам список публикаций. Откройте файл `app/controllers/posts_controller.rb` и взгляните на экшн `index`:

```
def index
  @posts = Post.all

  respond_to do |format|
    format.html # index.html.erb
    format.json { render :json => @posts }
  end
end
```

`Post.all` возвращает все публикации, которые сейчас есть в базе данных, как массив, содержащий все хранимые записи `Post`, который сохраняется в переменную экземпляра с именем `@posts`.

Дальнейшую информацию о поиске записей с помощью `Active Record` можете посмотреть в руководстве [Интерфейс запросов Active Record](#).

Блок `respond_to` отвечает за вызов HTML и JSON для этого экшна. Если вы пройдете по адресу <http://localhost:3000/posts.json>, то увидите все публикации в формате JSON. Формат HTML ищет вьюху в `app/views/posts/` с именем, соответствующим имени экшна. В Rails все переменные экземпляра экшна доступны во вьюхе. Вот код `app/view/posts/index.html.erb`:

```
<h1>Listing posts</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Title</th>
    <th>Content</th>
```

```

      <th></th>
      <th></th>
      <th></th>
    </tr>

    <% @posts.each do |post| %>
      <tr>
        <td><%= post.name %></td>
        <td><%= post.title %></td>
        <td><%= post.content %></td>
        <td><%= link_to 'Show', post %></td>
        <td><%= link_to 'Edit', edit_post_path(post) %></td>
        <td><%= link_to 'Destroy', post, :confirm => 'Are you sure?',
                                :method => :delete %></td>
      </tr>
    <% end %>
  </table>

  <br />

  <%= link_to 'New post', new_post_path %>

```

Эта выюха перебирает содержимое массива `@posts`, чтобы отразить содержимое и ссылки. Следует кое-что отметить во выюхе:

- `link_to` создает гиперссылку определенного назначения
- `edit_post_path` и `new_post_path` это хелперы, предоставленные Rails как часть роутинга RESTful. Вы увидите, что есть много таких хелперов для различных экшенов, включенных в контроллер.

В прежних версиях Rails следовало использовать `<%=h post.name %>`, чтобы любой HTML был экранирован перед вставкой на страницу. Теперь в Rails 3.0 это по умолчанию. Чтобы получить неэкранированный HTML, сейчас следует использовать `<%= raw post.name %>+.`

Более детально о процессе рендеринга смотрите тут: [Шаблоны и рендеринг в Rails](#).

Настройка макета

Вьюха это только часть того, как HTML отображается в вашем браузере. В Rails также есть концепция макетов, которые являются контейнерами для вьюх. Когда Rails рендерит вьюху для браузера, он делает это вкладывая вьюшный HTML в HTML макета. В прежних версиях Rails команда `rails generate scaffold` создала бы автоматически макет для определенного контроллера, такой как `app/views/layouts/posts.html.erb` для контроллера `posts`. Однако, это изменилось в Rails 3.0. Определенный для приложения макет используется для всех контроллеров и располагается в `app/views/layouts/application.html.erb`. Откройте этот макет в своем редакторе и измените тег `body+`, указав `style`:

```

<!DOCTYPE html>
<html>
<head>
  <title>Blog</title>
  <%= stylesheet_link_tag "application" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body style="background: #EEEEEE;">

  <%= yield %>

</body>
</html>

```

Теперь, когда вы обновите страницу `/posts`, то увидите серый бэкграунд страницы. Тот же серый бэкграунд будет использоваться везде на всех вьюхах контроллера `posts`.

Создание новой публикации

Создание новой публикации включает два экшна. Первый экшн это `new`, который создает экземпляр пустого объекта `Post`:

```

def new
  @post = Post.new

  respond_to do |format|
    format.html { # new.html.erb }
    format.json { render :json => @post }
  end
end

```

Вьюха `new.html.erb` отображает этот пустой объект `Post` пользователю:

```

<h1>New post</h1>

```

```
<%= render 'form' %>

<%= link_to 'Back', posts_path %>
```

Строка `<%= render 'form' %>` это наше первое введение в *партиалы* Rails. Партиал это фрагмент HTML и кода Ruby, который может использоваться в нескольких местах. В нашем случае форма, используемая для создания новой публикации, в основном сходна с формой, используемой для редактирования публикации, обе имеют текстовые поля для имени и заголовка и текстовую область для содержимого с кнопкой для создания новой публикации или обновления существующей.

Если заглянете в файл `views/posts/_form.html.erb`, то увидите следующее:

```
<%= form_for(@post) do |f| %>
  <% if @post.errors.any? %>
    <div id="errorExplanation">
      <h2><%= pluralize(@post.errors.count, "error") %> prohibited
        this post from being saved:</h2>
      <ul>
        <% @post.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

Этот партиал получает все переменные экземпляра, определенные в вызывающем файле `view`. В нашем случае контроллер назначил новый объект `Post` в `@post`, который, таким образом, будет доступен и во `view`, и в партиале как `@post`.

Чтобы узнать больше о партиалах, обратитесь к руководству [Макеты и рендеринг в Rails](#).

Блок `form_for` используется, чтобы создать форму HTML. Внутри этого блока у вас есть доступ к методам построения различных элементов управления формы. Например, `f.text_field :name` говорит Rails создать поле ввода текста в форме и подключить к нему атрибут `name` экземпляра для отображения. Эти методы можно использовать только для атрибутов той модели, на которой основана форма (в нашем случае `name`, `title` и `content`). В Rails предпочтительней использовать `form_for` чем писать на чистом HTML, так как код получается более компактным и явно связывает форму с конкретным экземпляром модели.

Блок `form_for` также достаточно сообразительный, чтобы понять, что вы собираетесь выполнить экшн *New Post* или *Edit Post*, и установит теги формы `action` и имена кнопок подтверждения подходящим образом в результирующем HTML.

Если нужно создать форму HTML, которая отображает произвольные поля, не связанные с моделью, нужно использовать метод `form_tag`, который предоставляет ярлыки для построения форм, которые непосредственно не связаны с экземпляром модели.

Когда пользователь нажмет кнопку *Create Post* в этой форме, браузер пошлет информацию назад к экшну `create` контроллера (Rails знает, что нужно вызвать экшн `create`, потому что форма посылает POST запрос; это еще одно соглашение, о котором говорилось ранее):

```
def create
  @post = Post.new(params[:post])

  respond_to do |format|
    if @post.save
      format.html { redirect_to(@post,
                               :notice => 'Post was successfully created.' ) }
      format.json { render :json => @post,
                          :status => :created, :location => @post }
    else
      format.html { render :action => "new" }
      format.json { render :json => @post.errors,
                          :status => :unprocessable_entity }
    end
  end
end
```

```
end
```

Экшн `create` создает новый экземпляр объекта `Post` из данных, предоставленных пользователем в форме, которые в Rails доступны в хэше `params`. После успешного сохранения новой публикации, `create` возвращает подходящий формат, который запросил пользователь (HTML в нашем случае). Затем он перенаправляет пользователя на экшн `show` получившейся публикации и устанавливает уведомление пользователя, что `Post was successfully created`.

Если публикация не была успешно сохранена, в связи с ошибками валидации, то контроллер возвратит пользователя обратно на экшн `new` со всеми сообщениями об ошибке, таким образом у пользователя есть шанс исправить их и попробовать снова.

Сообщение `"Post was successfully created"` хранится в хэше Rails `flash`, (обычно называемый просто *Flash*), с помощью него сообщения могут переноситься в другой экшн, предоставляя пользователю полезную информацию от статусе своих запросов. В случае с `create`, пользователь никогда не увидит какой-либо отрендеренной страницы в процессе создания публикации, так как происходит немедленный редирект, как только Rails сохраняет запись. `Flash` переносит сообщение в следующий экшн, поэтому когда пользователь перенаправляется в экшн `show` ему выдается сообщение `"Post was successfully created"`.

Отображение отдельной публикации

Когда нажмете на ссылку `show` для публикации на индексной странице, вы перейдете на URL такого вида `http://localhost:3000/posts/1`. Rails интерпретирует это как вызов экшна `show` для ресурса и передает 1 как параметр `:id`. Вот код экшна `show`:

```
def show
  @post = Post.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.json { render :json => @post }
  end
end
```

Экшн `show` использует `Post.find`, чтобы найти отдельную запись в базе данных по ее значению `id`. После нахождения записи, Rails отображает ее, используя `app/views/posts/show.html.erb`:

```
<p class="notice"><%= notice %></p>

<p>
  <b>Name:</b>
  <%= @post.name %>
</p>

<p>
  <b>Title:</b>
  <%= @post.title %>
</p>

<p>
  <b>Content:</b>
  <%= @post.content %>
</p>

<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
```

Редактирование публикаций

Подобно созданию новой публикации, редактирование публикации двусторонний процесс. Первый шаг это запрос определенной публикации `>edit_post_path(@post)`. Это вызывает экшн `edit` в контроллере:

```
def edit
  @post = Post.find(params[:id])
end
```

После нахождения требуемой публикации, Rails использует видюху `edit.html.erb` чтобы отобразить ее:

```
<h1>Editing post</h1>

<%= render 'form' %>

<%= link_to 'Show', @post %> |
<%= link_to 'Back', posts_path %>
```

Снова, как и в случае с экшном `new`, экшн `edit` использует парциал `form`. Однако в этот раз форма выполнит действие `PUT` в `PostsController` и кнопка подтверждения будет называться `"Update Post"`.

Подтверждение формы, созданной в этой видюхе, вызывает экшн `update` в контроллере:

```
def update
  @post = Post.find(params[:id])

  respond_to do |format|
    if @post.update_attributes(params[:post])
      format.html { redirect_to(@post,
                               :notice => 'Post was successfully updated.' ) }
      format.json { head :no_content }
    else
      format.html { render :action => "edit" }
      format.json { render :json => @post.errors,
                           :status => :unprocessable_entity }
    end
  end
end
```

В экшне `update`, Rails сначала использует параметр `:id`, возвращенный от вьюхи `edit`, чтобы обнаружить запись в базе данных, которую будем редактировать. Затем вызов `update_attributes` берет параметр `post` (хэш) из запроса и применяет его к записи. Если все проходит хорошо, пользователь перенаправляется на вьюху `show`. Если возникает какая-либо проблема, направляет обратно в экшн `edit`, чтобы исправить ее.

Уничтожение публикации

Наконец, нажатие на одну из ссылок `destroy` пошлет соответствующий `id` в экшн `destroy`:

```
def destroy
  @post = Post.find(params[:id])
  @post.destroy

  respond_to do |format|
    format.html { redirect_to posts_url }
    format.json { head :no_content }
  end
end
```

Метод `destroy` экземпляра модели `Active Record` убирает соответствующую запись из базы данных. После этого отображать нечего и Rails перенаправляет браузер пользователя на экшн `index` контроллера.

Добавляем вторую модель

Теперь, когда вы увидели, что представляет собой модель, построенной скаффолдом, настало время добавить вторую модель в приложение. Вторая модель будет управлять комментариями на публикации блога.

Генерируем модель

Модели в Rails используют имена в единственном числе, а их соответствующие таблицы базы данных используют имя во множественном числе. Для модели, содержащей комментарии, соглашением будет использовать имя `Comment`. Даже если вы не хотите использовать существующий аппарат настройки с помощью скаффолда, большинство разработчиков на Rails все равно используют генераторы для создания моделей и контроллеров. Чтобы создать новую модель, запустите эту команду в своем терминале:

```
$ rails generate model Comment commenter:string body:text post:references
```

Эта команда создаст четыре файла:

Файл	Назначение
db/migrate/20100207235629_create_comments.rb	Миграция для создания таблицы <code>comments</code> в вашей базе данных (ваше имя файла будет включать другую временную метку)
app/models/comment.rb	Модель <code>Comment</code>
test/unit/comment_test.rb	Каркас для юнит-тестирования модели комментариев
test/fixtures/comments.yml	Образцы комментариев для использования в тестировании

Сначала взглянем на `comment.rb`:

```
class Comment < ActiveRecord::Base
  belongs_to :post
end
```

Это очень похоже на модель `post.rb`, которую мы видели ранее. Разница в строке `belongs_to :post`, которая устанавливает связь `Active Record`. Вы ознакомитесь со связями в следующем разделе руководства.

В дополнение к модели, Rails также сделал миграцию для создания соответствующей таблицы базы данных:

```
class CreateComments < ActiveRecord::Migration
  def change
```



```
create_table :comments do |t|
  t.string :commenter
  t.text :body
  t.references :post

  t.timestamps
end

add_index :comments, :post_id
end
end
```

Строка `t.references` устанавливает столбец внешнего ключа для связи между двумя моделями. А строка `add_index` настраивает индексирование для этого столбца связи. Далее запускаем миграцию:

```
$ rake db:migrate
```

Rails достаточно сообразителен, чтобы запускать только те миграции, которые еще не были запущены для текущей базы данных, в нашем случае Вы увидите:

```
== CreateComments: migrating =====
-- create_table(:comments)
--> 0.0008s
-- add_index(:comments, :post_id)
--> 0.0003s
== CreateComments: migrated (0.0012s) =====
```

Связываем модели

Связи Active Record позволяют Вам легко объявлять отношения между двумя моделями. В случае с комментариями и публикациями, Вы можете описать отношения следующим образом:

- Каждый комментарий принадлежит одной публикации.
- Одна публикация может иметь много комментариев.

Фактически, это очень близко к синтаксису, который использует Rails для объявления этой связи. Вы уже видели строку кода в модели `Comment`, которая делает каждый комментарий принадлежащим публикации:

```
class Comment < ActiveRecord::Base
  belongs_to :post
end
```

Вам нужно отредактировать файл `post.rb`, добавив другую сторону связи:

```
class Post < ActiveRecord::Base
  validates :name, :presence => true
  validates :title, :presence => true,
            :length => { :minimum => 5 }

  has_many :comments
end
```

Эти два объявления автоматически делают доступным большое количество возможностей. Например, если у вас есть переменная экземпляра `@post`, содержащая публикацию, вы можете получить все комментарии, принадлежащие этой публикации, в массиве, вызвав `@post.comments`.

Более подробно о связях Active Record смотрите руководство [Связи Active Record](#).

Добавляем маршрут для комментариев

Как в случае с контроллером `home`, нам нужно добавить маршрут, чтобы Rails знал, по какому адресу мы хотим пройти, чтобы увидеть комментарии. Снова откройте файл `config/routes.rb`. Вверху вы увидите вхождение для `posts`, автоматически добавленное генератором скаффолда: `resources +:posts`. Отредактируйте его следующим образом:

```
resources :posts do
  resources :comments
end
```

Это создаст `comments` как *вложенный ресурс* в `posts`. Это другая сторона захвата иерархических отношений, существующих между публикациями и комментариями.

Более подробно о роутинге написано в руководстве [Роутинг в Rails](#).

Генерируем контроллер

Имея модель, обратим свое внимание на создание соответствующего контроллера. Вот генератор для него:

```
$ rails generate controller Comments
```

Создадутся шесть файлов и пустая директория:

Файл/Директория	Назначение
app/controllers/comments_controller.rb	Контроллер Comments
app/views/comments/	Вьюхи контроллера хранятся здесь
test/functional/comments_controller_test.rb	Функциональные тесты для контроллера
app/helpers/comments_helper.rb	Хелпер для вьюх
test/unit/helpers/comments_helper_test.rb	Юнит-тесты для хелпера
app/assets/javascripts/comment.js.coffee	CoffeeScript для контроллера
app/assets/stylesheets/comment.css.scss	Каскадная таблица стилей для контроллера

Как и в любом другом блоге, наши читатели будут создавать свои комментарии сразу после прочтения публикации, и после добавления комментария они будут направляться обратно на страницу отображения публикации и видеть, что их комментарий уже отражен. В связи с этим, наш CommentsController служит как средство создания комментариев и удаления спама, если будет.

Сначала мы расширим шаблон Post show (/app/views/posts/show.html.erb), чтобы он позволял добавить новый комментарий:

```
<p class="notice"><%= notice %></p>

<p>
  <b>Name:</b>
  <%= @post.name %>
</p>

<p>
  <b>Title:</b>
  <%= @post.title %>
</p>

<p>
  <b>Content:</b>
  <%= @post.content %>
</p>

<h2>Add a comment:</h2>
<%= form_for([@post, @post.comments.build]) do |f| %>
  <div class="field">
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </div>
  <div class="field">
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>

<%= link_to 'Edit Post', edit_post_path(@post) %> |
<%= link_to 'Back to Posts', posts_path %> |
```

Это добавит форму на страницу отображения публикации, создающую новый комментарий при вызове экшна create в CommentsController. Давайте напишем его:

```
class CommentsController < ApplicationController
  def create
    @post = Post.find(params[:post_id])
    @comment = @post.comments.create(params[:comment])
    redirect_to post_path(@post)
  end
end
```

Тут все немного сложнее, чем вы видели в контроллере для публикаций. Это побочный эффект вложения, которое вы настроили. Каждый запрос к комментарию отслеживает публикацию, к которой комментарий присоединен, таким образом сначала решаем вопрос с получением публикации, вызвав find на модели Post.

Кроме того, код пользуется преимуществом некоторых методов, доступных для связей. Мы используем метод create на @post.comments, чтобы создать и сохранить комментарий. Это автоматически связывает комментарий так, что он принадлежит к определенной публикации.

Как только мы создали новый комментарий, мы возвращаем пользователя обратно на оригинальную публикацию, используя хелпер post_path(@post). Как мы уже видели, он вызывает экшн show в PostsController, который, в свою очередь, рендерит шаблон show.html.erb. В этом месте мы хотим отображать комментарии, поэтому давайте добавим следующее в app/views/posts/show.html.erb.

```
<p class="notice"><%= notice %></p>

<p>
  <b>Name:</b>
  <%= @post.name %>
</p>

<p>
  <b>Title:</b>
  <%= @post.title %>
</p>

<p>
  <b>Content:</b>
  <%= @post.content %>
</p>

<h2>Comments</h2>
<% @post.comments.each do |comment| %>
  <p>
    <b>Commenter:</b>
    <%= comment.commenter %>

    <p>
      <b>Comment:</b>
      <%= comment.body %>
    </p>
  <% end %>

<h2>Add a comment:</h2>
<%= form_for([@post, @post.comments.build]) do |f| %>
  <div class="field">
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </div>
  <div class="field">
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>

<br />

<%= link_to 'Edit Post', edit_post_path(@post) %> |
<%= link_to 'Back to Posts', posts_path %> |
```

Теперь в вашем блоге можно добавлять публикации и комментарии и отображать их в нужных местах.

Рефакторинг

Теперь, когда у нас есть работающие публикации и комментарии, взглянем на шаблон `app/views/posts/show.html.erb`. Он стал длинным и неудобным. Давайте воспользуемся партиялами, чтобы разгрузить его.

Рендеринг коллекций партиялов

Сначала сделаем партиял для комментариев, показывающий все комментарии для публикации. Создайте файл `app/views/comments/_comment.html.erb` и поместите в него следующее:

```
<p>
  <b>Commenter:</b>
  <%= comment.commenter %>
</p>

<p>
  <b>Comment:</b>
  <%= comment.body %>
</p>
```

Затем можно изменить `app/views/posts/show.html.erb` вот так:

```
<p class="notice"><%= notice %></p>

<p>
  <b>Name:</b>
  <%= @post.name %>
</p>
```

```
<p>
  <b>Title:</b>
  <%= @post.title %>
</p>

<p>
  <b>Content:</b>
  <%= @post.content %>
</p>

<h2>Comments</h2>
<%= render @post.comments %>

<h2>Add a comment:</h2>
<%= form_for([@post, @post.comments.build]) do |f| %>
  <div class="field">
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </div>
  <div class="field">
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>

<br />

<%= link_to 'Edit Post', edit_post_path(@post) %> |
<%= link_to 'Back to Posts', posts_path %> |
```

Теперь это отрендерит парциал `app/views/comments/_comment.html.erb` по разу для каждого комментария в коллекции `@post.comments`. Так как метод `render` перебирает коллекцию `@post.comments`, он назначает каждый комментарий локальной переменной с именем, как у парциала, в нашем случае `comment`, которая нам доступна в парциале для отображения.

Рендеринг частичной формы

Давайте также переместим раздел нового коммента в свой парциал. Опять же, создайте файл `app/views/comments/_form.html.erb`, содержащий:

```
<%= form_for([@post, @post.comments.build]) do |f| %>
  <div class="field">
    <%= f.label :commenter %><br />
    <%= f.text_field :commenter %>
  </div>
  <div class="field">
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

Затем измените `app/views/posts/show.html.erb` следующим образом:

```
<p class="notice"><%= notice %></p>

<p>
  <b>Name:</b>
  <%= @post.name %>
</p>

<p>
  <b>Title:</b>
  <%= @post.title %>
</p>

<p>
  <b>Content:</b>
  <%= @post.content %>
</p>

<h2>Comments</h2>
<%= render @post.comments %>

<h2>Add a comment:</h2>
<%= render "comments/form" %>

<br />
```

```
<%= link_to 'Edit Post', edit_post_path(@post) %> |  
<%= link_to 'Back to Posts', posts_path %> |
```

Второй render всего лишь определяет шаблон партиала, который мы хотим рендерить, comments/form. Rails достаточно сообразительный, чтобы подставить подчеркивание в эту строку и понять, что Вы хотели рендерить файл _form.html.erb в директории app/views/comments.

Объект @post доступен в любых партиалах, рендеримых во вьюхе, так как мы определили его как переменную экземпляра.

Удаление комментариев

Другой важной особенностью блога является возможность удаления спама. Чтобы сделать это, нужно вставить некоторую ссылку во вьюхе и экшн DELETE в CommentsController.

Поэтому сначала добавим ссылку для удаления в партиал app/views/comments/_comment.html.erb:

```
<p>  
  <b>Commenter:</b>  
  <%= comment.commenter %>  
</p>  
  
<p>  
  <b>Comment:</b>  
  <%= comment.body %>  
</p>  
  
<p>  
  <%= link_to 'Destroy Comment', [comment.post, comment],  
    :confirm => 'Are you sure?',  
    :method => :delete %>  
</p>
```

Нажатие этой новой ссылки “Destroy Comment” запустит DELETE /posts/:id/comments/:id в нашем CommentsController, который затем будет использоваться для нахождения комментария, который мы хотим удалить, поэтому давайте добавим экшн destroy в наш контроллер:

```
class CommentsController < ApplicationController  
  
  def create  
    @post = Post.find(params[:post_id])  
    @comment = @post.comments.create(params[:comment])  
    redirect_to post_path(@post)  
  end  
  
  def destroy  
    @post = Post.find(params[:post_id])  
    @comment = @post.comments.find(params[:id])  
    @comment.destroy  
    redirect_to post_path(@post)  
  end  
  
end
```

Экшн destroy найдет публикацию, которую мы просматриваем, обнаружит комментарий в коллекции @post.comments и затем уберет его из базы данных и вернет нас обратно на просмотр публикации.

Удаление связанных объектов

Если удаляете публикацию, связанные с ней комментарии также должны быть удалены. В ином случае они будут просто занимать место в базе данных. Rails позволяет использовать опцию dependent на связи для достижения этого. Измените модель Post, app/models/post.rb, следующим образом:

```
class Post < ActiveRecord::Base  
  validates :name, :presence => true  
  validates :title, :presence => true,  
    :length => { :minimum => 5 }  
  has_many :comments, :dependent => :destroy  
end
```

Безопасность

Если вы опубликуете свой блог онлайн, любой сможет добавлять, редактировать и удалять публикации или удалять комментарии.

Rails предоставляет очень простую аутентификационную систему HTTP, которая хорошо работает в этой ситуации.

В PostsController нам нужен способ блокировать доступ к различным экшнам, если пользователь не аутентифицирован, тут мы можем использовать метод Rails `http_basic_authenticate_with`, разрешающий доступ к требуемым экшнам, если метод позволит это.

Чтобы использовать систему аутентификации, мы определим ее вверху нашего PostsController, в нашем случае, мы хотим, чтобы пользователь был аутентифицирован для каждого экшна, кроме index и show, поэтому напомним так:

```
class PostsController < ApplicationController

  http_basic_authenticate_with :name => "dhh", :password => "secret", :except => [:index, :show]

  # GET /posts
  # GET /posts.json
  def index
    @posts = Post.all
    respond_to do |format|
      # пропущено для краткости
    end
  end
end
```

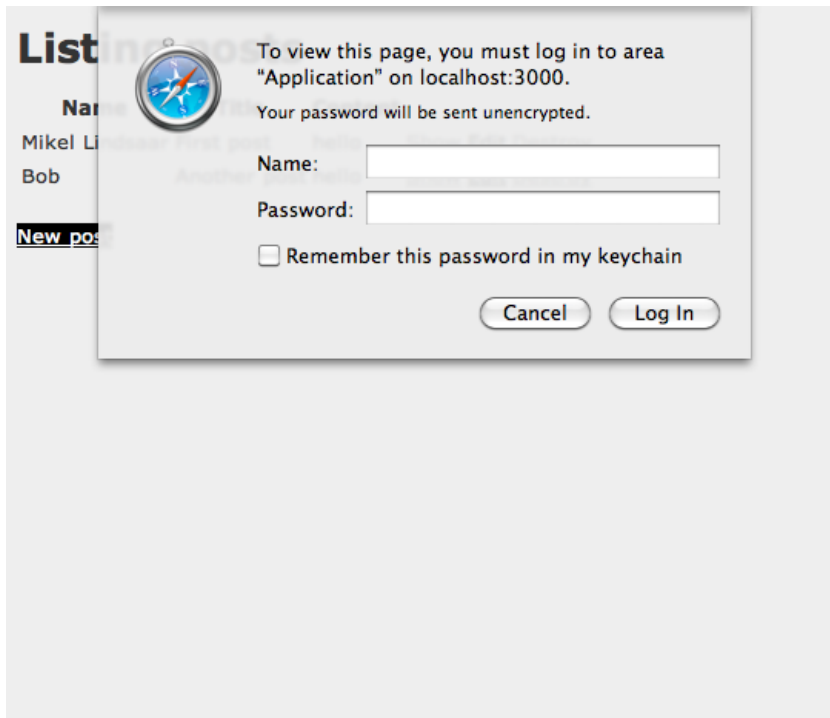
Мы также хотим позволить только аутентифицированным пользователям удалять комментарии, поэтому в CommentsController мы напомним:

```
class CommentsController < ApplicationController

  http_basic_authenticate_with :name => "dhh", :password => "secret", :only => :destroy

  def create
    @post = Post.find(params[:post_id])
    # пропущено для краткости
  end
end
```

Теперь, если попытаетесь создать новую публикацию, то встретитесь с простым вызовом аутентификации HTTP



Создаем мульти-модельную форму

Другой особенностью обычного блога является возможность метки (тегирования) публикаций. Чтобы применить эту особенность, вашему приложению нужно взаимодействовать более чем с одной моделью в одной форме. Rails предлагает поддержку вложенных форм.

Чтобы это продемонстрировать, добавим поддержку для присвоения каждой публикации множественных тегов непосредственно в форме, где вы создаете публикации. Сначала создадим новую модель для хранения тегов:

```
$ rails generate model tag name:string post:references
```

Снова запустим миграцию для создания таблицы в базе данных:

```
$ rake db:migrate
```

Далее отредактируем файл `post.rb`, создав другую сторону связи, и сообщим Rails (с помощью макроса `accepts_nested_attributes_for`), что намереваемся редактировать теги непосредственно в публикациях:

```
class Post < ActiveRecord::Base
  validates :name, :presence => true
  validates :title, :presence => true,
            :length => { :minimum => 5 }

  has_many :comments, :dependent => :destroy
  has_many :tags

  accepts_nested_attributes_for :tags, :allow_destroy => :true,
  :reject_if => proc { |attrs| attrs.all? { |k, v| v.blank? } }
end
```

Опция `:allow_destroy` на объявлении вложенного атрибута говорит Rails отображать чекбокс “remove” во вьюхе, которую вы скоро создадите. Опция `:reject_if` предотвращает сохранение новых тегов, не имеющих каких-либо заполненных атрибутов.

Изменим `views/posts/_form.html.erb`, чтобы рендерить партиал, создающий теги:

```
<% @post.tags.build %>
<%= form_for(@post) do |post_form| %>
  <% if @post.errors.any? %>
    <div id="errorExplanation">
      <h2><%= pluralize(@post.errors.count, "error") %> prohibited this post from being saved:</h2>
      <ul>
        <% @post.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= post_form.label :name %><br />
    <%= post_form.text_field :name %>
  </div>
  <div class="field">
    <%= post_form.label :title %><br />
    <%= post_form.text_field :title %>
  </div>
  <div class="field">
    <%= post_form.label :content %><br />
    <%= post_form.text_area :content %>
  </div>
  <h2>Tags</h2>
  <%= render :partial => 'tags/form',
            :locals => { :form => post_form } %>
  <div class="actions">
    <%= post_form.submit %>
  </div>
<% end %>
```

Отметьте, что мы изменили `f` в `form_for(@post) do |f|` на `post_form`, чтобы было проще понять, что происходит.

Этот пример показывает другую опцию хелпера `render`, способную передавать локальные переменные, в нашем случае мы хотим передать переменную `form` в партиал, относящуюся к объекту `post_form`.

Мы также добавили `@post.tags.build` вверху формы. Это сделано для того, чтобы убедиться, что имеется новый тег, готовый к указанию его имени пользователем. Если не создаете новый тег, то форма не появится, так как не будет доступного для создания нового объекта `Tag`.

Теперь создайте папку `app/views/tags` и создайте файл с именем `_form.html.erb`, содержащий форму для тега:

```
<%= form.fields_for :tags do |tag_form| %>
  <div class="field">
    <%= tag_form.label :name, 'Tag:' %>
    <%= tag_form.text_field :name %>
  </div>
  <% unless tag_form.object.nil? || tag_form.object.new_record? %>
    <div class="field">
      <%= tag_form.label :_destroy, 'Remove:' %>
      <%= tag_form.check_box :_destroy %>
    </div>
  <% end %>
<% end %>
```

Наконец, отредактируйте шаблон `app/views/posts/show.html.erb`, чтобы отображались наши теги.

```
<p class="notice"><%= notice %></p>

<p>
  <b>Name:</b>
  <%= @post.name %>
</p>
```



```
<p>
  <b>Title:</b>
  <%= @post.title %>
</p>

<p>
  <b>Content:</b>
  <%= @post.content %>
</p>

<p>
  <b>Tags:</b>
  <%= @post.tags.map { |t| t.name }.join(", ") %>
</p>

<h2>Comments</h2>
<%= render @post.comments %>

<h2>Add a comment:</h2>
<%= render "comments/form" %>

<%= link_to 'Edit Post', edit_post_path(@post) %> |
<%= link_to 'Back to Posts', posts_path %> |
```

После этих изменений, вы сможете редактировать публикацию и ее теги в одной и той же вьюхе.

Однако, такой вызов метода `@post.tags.map { |t| t.name }.join(", ")` неуклюж, мы можем исправить это, создав метод хелпера.

Хелперы вьюхи

Хелперы вьюхи обитают в `app/helpers` и представляют небольшие фрагменты повторно используемого кода для вьюх. В нашем случае, мы хотим метод, который заносит в строку несколько объектов вместе, используя их атрибуты имени и соединяя их запятыми. Так как это нужно для шаблона `Post show`, мы поместим код в `PostsHelper`.

Откройте `app/helpers/posts_helper.rb` и добавьте следующее:

```
module PostsHelper
  def join_tags(post)
    post.tags.map { |t| t.name }.join(", ")
  end
end
```

Теперь можно отредактировать вьюху в `app/views/posts/show.html.erb`, чтобы она выглядела так:

```
<p class="notice"><%= notice %></p>

<p>
  <b>Name:</b>
  <%= @post.name %>
</p>

<p>
  <b>Title:</b>
  <%= @post.title %>
</p>

<p>
  <b>Content:</b>
  <%= @post.content %>
</p>

<p>
  <b>Tags:</b>
  <%= join_tags(@post) %>
</p>

<h2>Comments</h2>
<%= render @post.comments %>

<h2>Add a comment:</h2>
<%= render "comments/form" %>

<%= link_to 'Edit Post', edit_post_path(@post) %> |
<%= link_to 'Back to Posts', posts_path %> |
```

Что дальше?

После того, как вы создали свое первое приложение на Rails, можете свободно его модифицировать и экспериментировать на свое усмотрение. Но без посторонней помощи, вы, скорее всего, ничего не сможете сделать. Так же, как вы обращались к этому руководству “Rails для начинающих”, далее можете так же свободно пользоваться этими ресурсами:

- The [Ruby on Rails guides](#)
- The [Ruby on Rails Tutorial](#)
- The [Ruby on Rails mailing list](#)
- The [#rubyonrails](#) channel on irc.freenode.net
- The [Rails Wiki](#)

Отдельно хотелось бы выделить и поддержать следующие хорошие русскоязычные ресурсы по Ruby on rails:

- [Ruby on Rails по-русски](#)
- [Изучение Rails на примерах](#)
- [Блог ‘Ruby on Rails с нуля!’](#)
- [Railsclub – организация конференций](#)

Rails также поставляется со встроенной помощью, которую Вы можете вызвать, используя командную утилиту rake:

- Запуск `rake doc:guides` выложит полную копию Rails Guides в папку `/doc/guides` вашего приложения. Откройте `/doc/guides/index.html` в веб-браузере, для обзора руководства.
- Запуск `rake doc:rails` выложит полную копию документации по API для Rails в папку `/doc/api` вашего приложения. Откройте `/doc/api/index.html` в веб-браузере, для обзора документации по API.

Ошибки конфигурации

Простейший способ работы с Rails заключается в хранении всех внешних данных в UTF-8. Если не так, библиотеки Ruby и Rails часто будут способны конвертировать ваши родные данные в UTF-8, но это не всегда надежно работает, поэтому лучше быть уверенным, что все внешние данные являются UTF-8.

Если вы допускаете ошибку в этой области, наиболее обычным симптомом является черный ромбик со знаком вопроса внутри, появляющийся в браузере. Другим обычным симптомом являются символы, такие как “Ã¼” появляющиеся вместо “ü”. Rails предпринимает ряд внутренних шагов для смягчения общих случаев тех проблем, которые могут быть автоматически обнаружены и исправлены. Однако, если имеются внешние данные, не хранящиеся в UTF-8, это может привести к такого рода проблемам, которые не могут быть автоматически обнаружены Rails и исправлены.

Два наиболее обычных источника данных, которые не в UTF-8:

- Ваш текстовый редактор: Большинство текстовых редакторов (такие как Textmate), по умолчанию сохраняют файлы как UTF-8. Если ваш текстовый редактор так не делает, это может привести к тому, что специальные символы, введенные в ваши шаблоны (такие как ё) появятся как ромбик с вопросительным знаком в браузере. Это также касается ваших файлов перевода I18N. Большинство редакторов, не устанавливающие по умолчанию UTF-8 (такие как некоторые версии Dreamweaver) предлагают способ изменить умолчания на UTF-8. Сделайте так.
- Ваша база данных. Rails по умолчанию преобразует данные из вашей базы данных в UTF-8 на границе. Однако, если ваша база данных не использует внутри UTF-8, она может не быть способной хранить все символы, которые введет ваш пользователь. Например, если ваша база данных внутри использует Latin-1, и ваш пользователь вводит русские, ивритские или японские символы, данные будут потеряны как только попадут в базу данных. Если возможно, используйте UTF-8 как внутреннее хранилище в своей базе данных.