

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 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.rdoc</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 получиться 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 *NewPost* 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 comments в вашей базе данных (ваше имя файла будет включать другую временную метку) |
| app/models/comment.rb | Модель Comment |
| 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
```

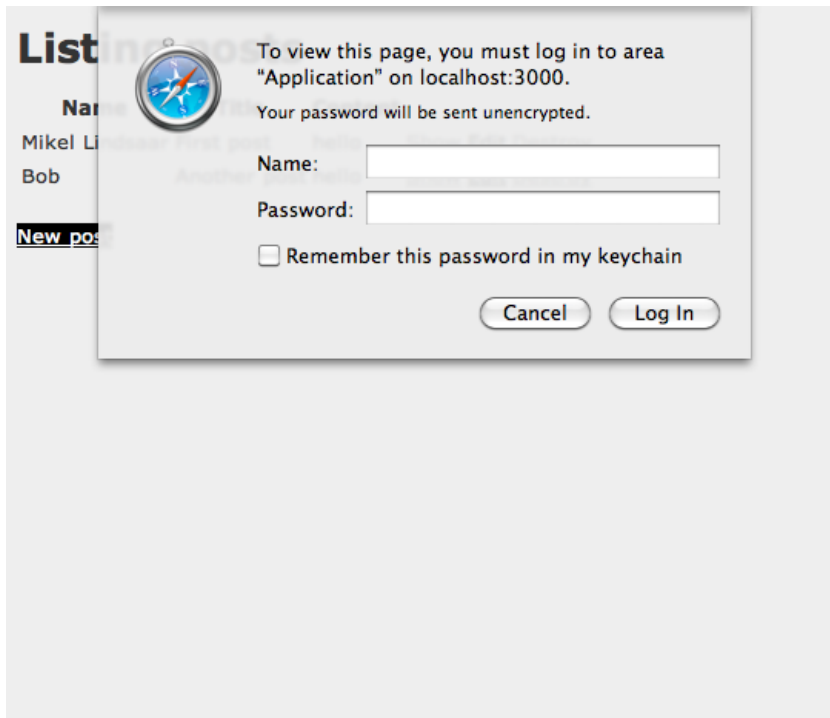
Мы также хотим позволить только аутентифицированным пользователям удалять комментарии, поэтому в CommentsController мы напомним:

```
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.

Миграции базы данных Rails

Миграции это удобный способ привести вашу базу данных к структурированной и организованной основе. Можно вручную править фрагменты SQL, но вы тогда ответственны сказать другим разработчикам, что они должны тоже выполнить это. Вам также нужно будет вести список изменений, которые нужно вносить каждый раз, когда проект развертывается на новой машине или реальном сервере.

Active Record отслеживает, какие миграции уже были выполнены, поэтому все, что нужно сделать, это обновить свой исходный код и запустить `rake db:migrate`. Active Record сам определит, какие миграции нужно запустить. Он также обновит ваш файл `db/schema.rb` в соответствии с новой структурой вашей базы данных.

Миграции также позволяют вам описать эти изменения на Ruby. Очень хорошо, что это (как и большая часть функциональности Active Record) полностью не зависит от базы данных: вам не нужно беспокоиться о точном синтаксисе `CREATE TABLE` или особенностей `SELECT *` (как в случае, если вы пишете на чистом SQL, когда надо учитывать особенности разных баз данных). Например, вы можете использовать SQLite3 при разработке, а на рабочем приложении MySQL.

Из этого руководства вы узнаете все о миграциях, включая:

- Генераторы, используемые для их создания
- Методы Active Record для воздействия с Вашей базой данных
- Задачи Rake, воздействующие на них

- Как они связаны со schema.rb

Анатомия миграции

Прежде чем погрузиться в подробности о миграциях, вот небольшие примеры того, что вы сможете сделать:

```
class CreateProducts < ActiveRecord::Migration
  def up
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end

  def down
    drop_table :products
  end
end
```

Эта миграция добавляет таблицу products со строковым столбцом name и текстовым столбцом description. Первичный ключ, названный id, также будет добавлен по умолчанию, поэтому его специально не нужно указывать. Столбцы временных меток created_at и updated_at, которые Active Record заполняет автоматически, также будут добавлены. Откат этой миграции очень прост, это удаление таблицы.

Миграции не ограничены изменением схемы. Можно использовать их для исправления плохих данных в базе данных или заполнения новых полей:

```
class AddReceiveNewsletterToUsers < ActiveRecord::Migration
  def up
    change_table :users do |t|
      t.boolean :receive_newsletter, :default => false
    end
    User.update_all ["receive_newsletter = ?", true]
  end

  def down
    remove_column :users, :receive_newsletter
  end
end
```

Есть некоторые [оговорки](#) в использовании моделей в ваших миграциях.

Эта миграция добавляет столбец receive_newsletter (получать письма) в таблице users. Мы хотим установить значение по умолчанию false для новых пользователей, но для существующих пользователей полагаем, что они выбрали этот вариант, поэтому мы используем модель User, чтобы установить значение true для существующих пользователей.

Rails 3.1 сделал миграции разумнее, предоставив новый метод change. Этот метод предпочтителен для написания конструирующих миграций (добавление столбцов или таблиц). Миграция знает, как мигрировать вашу базу данных и обратить ее, когда миграция откатывается, без необходимости писать отдельный метод down method.

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

Миграции это классы

Миграция это subclass ActiveRecord::Migration, который имеет два метода класса: up (выполнение требуемых изменений) и down (их откат).

Active Record предоставляет методы, которые выполняют общие задачи по определению данных способом, независимым от типа базы данных (подробнее об этом будет написано позже):

- add_column
- add_index
- change_column
- change_table
- create_table
- drop_table

- `remove_column`
- `remove_index`
- `rename_column`

Если вам нужно выполнить специфичную для вашей базы данных задачу (например, создать [внешний ключ](#) как ограничение ссылочной целостности), то функция `execute` позволит вам запустить произвольный SQL. Миграция – всего лишь обычный класс Ruby, так что вы не ограничены этими функциями. Например, после добавления столбца можно написать код, устанавливающий значения этого столбца для существующих записей (если необходимо, используя ваши модели).

В базах данных, поддерживающих транзакции с выражениями, изменяющими схему (такие как PostgreSQL или SQLite3), миграции упаковываются в транзакции. Если база данных не поддерживает это (например, MySQL), то, если миграция проходит неудачно, те части, которые прошли, не откатываются. Вам нужно будет откатить внесенные изменения вручную.

Имена

Миграции хранятся как файлы в директории `db/migrate`, один файл на каждый класс. Имя файла имеет вид `YYYYMMDDHHMMSS_create_products.rb`, это означает, что временная метка UTC идентифицирует миграцию, затем идет знак подчеркивания, затем идет имя миграции, где слова разделены подчеркиваниями. Имя класса миграции содержит буквенную часть названия файла, но уже в формате CamelCase (т.е. слова пишутся слитно, каждое слово начинается с большой буквы). Например, `20080906120000_create_products.rb` должен определять класс `CreateProducts`, а `20080906120001_add_details_to_products.rb` должен определять `AddDetailsToProducts`. Если вы вдруг захотите переименовать файл, вы *обязаны* изменить имя класса внутри, иначе Rails сообщит об отсутствующем классе.

Внутри Rails используются только номера миграций (временные метки) для их идентификации. До Rails 2.1 нумерация миграций начиналась с 1 и увеличивалась каждый раз, когда создавалась новая миграция. Когда работало несколько разработчиков были часты коллизии, когда требовалось перенумеровывать миграции. В Rails 2.1 этого в большей степени смогли избежать, используя время создания миграции для идентификации. Старую схему нумерации можно вернуть, добавив следующую строку в `config/application.rb`.

```
config.active_record.timestamped_migrations = false
```

Комбинация временной метки и записи, какие миграции были выполнены, позволяет Rails регулировать ситуации, которые могут произойти в случае с несколькими разработчиками.

Например, Алиса добавила миграции 20080906120000 и 20080906123000, а Боб добавил 20080906124500 и запустил ее. Алиса закончила свои изменения и отразила их в своих миграциях, а Боб откатывает последние изменения. Когда Боб запускает `rake db:migrate`, Rails знает, что он не запускал две миграции Алисы, таким образом он запускает метод `up` для каждой миграции.

Конечно, это не замена общения внутри группы. Например, если миграция Алисы убирает таблицу, которую миграция Боба предполагает существующей, возникнут проблемы.

Изменение миграций

Иногда вы можете допустить ошибку, когда пишете миграцию. Если вы уже запустили эту миграцию, то не можете просто отредактировать ее и запустить снова: Rails считает, что эта миграция уже запускалась, и ничего не будет делать, когда вы запустите `rake db:migrate`. Вы должны откатить миграцию (например, командой `rake db:rollback`), отредактировать миграцию и затем запустить `rake db:migrate` чтобы выполнить скорректированную версию.

В целом, редактирование существующих миграций это не хорошая идея: вы создаете дополнительную работу для себя и своих коллег, и вызываете большую проблему, если существующая версия уже работает в режиме `production`. Вместо этого вы можете написать новую миграцию, которая выполнит требуемые вами изменения. Редактирование только что созданной миграции, которую еще не передали в систему управлениями версий (то есть которая есть только на вашей машине) относительно безвредно.

Поддерживаемые типы

Active Record поддерживает следующие типы столбцов базы данных:

- `:binary`
- `:boolean`
- `:date`
- `:datetime`
- `:decimal`
- `:float`
- `:integer`
- `:primary_key`
- `:string`
- `:text`
- `:time`
- `:timestamp`

Они отображаются в наиболее подходящем типе базы данных, например, в MySQL тип `:string` отображается как `VARCHAR(255)`. Вы можете создать столбцы типов, не поддерживаемых Active Record, если будете использовать не сексинтаксис, например

```
create_table :products do |t|
  t.column :name, 'polygon', :null => false
end
```

Этот способ, однако препятствует переходу на другие базы данных.

Создание миграции

Создание модели

Генераторы модели и скаффолда создают соответствующие миграции для добавления новой модели. Эта миграция уже содержит инструкции для создания соответствующей таблицы. Если вы сообщите Rails какие столбцы вам нужны, выражения для создания этих столбцов также будут добавлены. Например, запуск

```
$ rails generate model Product name:string description:text
```

создаст подобную миграцию

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

Вы можете указать столько пар имя-столбца/тип, сколько хотите. По умолчанию сгенерированная миграция будет включать `t.timestamps` (что создает столбцы `updated_at` и `created_at`, автоматически заполняемые Active Record).

Создание автономной миграции

Если хотите создать миграцию для других целей (например, добавить столбец в существующую таблицу), также возможно использовать генератор миграции:

```
$ rails generate migration AddPartNumberToProducts
```

Это создаст пустую, но правильно названную миграцию:

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
  end
end
```

Если имя миграции имеет форму `"AddXXXToYYY"` или `"RemoveXXXFromYYY"` и далее следует перечень имен столбцов и их типов, то в миграции будут созданы соответствующие выражения `add_column` и `remove_column`.

```
$ rails generate migration AddPartNumberToProducts part_number:string
```

создаст

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
  end
end
```

Аналогично,

```
$ rails generate migration RemovePartNumberFromProducts part_number:string
```

создаст

```
class RemovePartNumberFromProducts < ActiveRecord::Migration
  def up
    remove_column :products, :part_number
  end

  def down
    add_column :products, :part_number, :string
  end
end
```

Вы не ограничены одним создаваемым столбцом, например

```
$ rails generate migration AddDetailsToProducts part_number:string price:decimal
```

создаст

```
class AddDetailsToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
    add_column :products, :price, :decimal
  end
end
```

Как всегда, то, что было сгенерировано, является всего лишь стартовой точкой. Вы можете добавлять и убирать строки, как считаете нужным, отредактировав файл db/migrate/YYYYMMDDHHMMSS_add_details_to_products.rb.

Генерируемый файл миграции для деструктивных миграций будет все еще по-старому использовать методы up и down. Это так, потому что Rails не может знать оригинальные типы данных, которые вы создали когда-то ранее.

Написание миграции

Как только вы создали свою миграцию, используя один из генераторов, пришло время поработать!

Создание таблицы

Метод `create_table` миграции будет одной из ваших рабочих лошадей. Обычное использование такое

```
create_table :products do |t|
  t.string :name
end
```

Это создаст таблицу `products` со столбцом `name` (и, как обсуждалось выше, подразумеваемым столбцом `id`).

Объект, переданный в блок, позволяет вам создавать столбцы в таблице. Есть два способа сделать это. Первая (традиционная) форма выглядит так

```
create_table :products do |t|
  t.column :name, :string, :null => false
end
```

Вторая форма, так называемая “секси” миграция, опускает несколько избыточный метод `column`. Вместо этого, методы `string`, `integer`, и т.д. создают столбцы этого типа. Дополнительные параметры те же самые.

```
create_table :products do |t|
  t.string :name, :null => false
end
```

По умолчанию `create_table` создаст первичный ключ, названный `id`. Вы можете изменить имя первичного ключа с помощью опции `:primary_key` (не забудьте также обновить соответствующую модель), или, если вы вообще не хотите первичный ключ (например, соединительная таблица для связи *многие ко многим*), можно указать опцию `:id => false`. Если нужно передать базе данных специфичные опции, вы можете поместить фрагмент SQL в опцию `:options`. Например,

```
create_table :products, :options => "ENGINE=BLACKHOLE" do |t|
  t.string :name, :null => false
end
```

добавит `ENGINE=BLACKHOLE` к выражению SQL, используемому для создания таблицы (при использовании MySQL по умолчанию передается `ENGINE=InnoDB`).

Изменение таблиц

Близкий родственник `create_table` это `change_table`, используемый для изменения существующих таблиц. Он используется подобно `create_table`, но у объекта, передаваемого в блок, больше методов. Например

```
change_table :products do |t|
  t.remove :description, :name
  t.string :part_number
  t.index :part_number
  t.rename :upccode, :upc_code
end
```

удаляет столбцы `description` и `name`, создает строковый столбец `part_number` и добавляет индекс на него. Наконец, он переименовывает столбец `upccode`.

Специальные хелперы

Active Record предоставляет некоторые ярлыки для обычной функциональности. Вот, например, обычно добавляются два столбца `created_at` и `updated_at`, поэтому есть метод, который делает непосредственно это:

```
create_table :products do |t|
  t.timestamps
end
```

создает новую таблицу `products` с этими двумя столбцами (плюс столбец `id`), в то время как

```
change_table :products do |t|
  t.timestamps
end
```

добавляет эти столбцы в существующую таблицу.

Другой хелпер называется `references` (также доступен как `belongs_to`). В простой форме он только добавляет немного читаемости кода

```
create_table :products do |t|
  t.references :category
end
```

создаст столбец `category_id` подходящего типа. Отметьте, что вы указали имя модели, а не имя столбца. Active Record добавил `_id` за вас. Если у вас есть полиморфные связи `belongs_to`, то `references` создаст оба требуемых столбца:

```
create_table :products do |t|
  t.references :attachment, :polymorphic => {:default => 'Photo'}
end
```

добавит столбец `attachment_id` и строковый столбец `attachment_type` со значением по умолчанию "Photo".

Хелпер `references` фактически не создает для вас внешний ключ как ограничение ссылочной целостности. Нужно использовать `execute` или плагин, который предоставляет [поддержку внешних ключей](#).

Если вам недостаточно хелперов, предоставленных Active Record, можете использовать функцию `execute` для запуска произвольного SQL.

Больше подробностей и примеров отдельных методов содержится в документации по API, в частности, документация для [ActiveRecord::ConnectionAdapters::SchemaStatements](#) (который обеспечивает методы, доступные в методах `up` и `down`), [ActiveRecord::ConnectionAdapters::TableDefinition](#) (который обеспечивает методы, доступные у объекта, переданного методом `create_table`) и [ActiveRecord::ConnectionAdapters::Table](#) (который обеспечивает методы, доступные у объекта, переданного методом `change_table`).

Использование метода `change`

Метод `change` устраняет необходимость писать оба метода `up` и `down` в тех случаях, когда Rails знает, как обратить изменения автоматически. На текущий момент метод `change` поддерживает только эти определения миграции:

- `add_column`
- `add_index`
- `add_timestamps`
- `create_table`
- `remove_timestamps`
- `rename_column`
- `rename_index`
- `rename_table`

Если вы нуждаетесь в использовании иных методов, следует писать методы `up` и `down` вместо метода `change`.

Использование методов `up/down`

Метод `down` вашей миграции должен вернуть назад изменения, выполненные методом `up`. Другими словами, схема базы данных не должна измениться, если вы выполните `up`, а затем `down`. Например, если вы создали таблицу в методе `up`, то должны ее удалить в методе `down`. Благоразумно откатить преобразования в порядке, полностью обратном порядку метода `up`. Например,

```
class ExampleMigration < ActiveRecord::Migration
  def up
    create_table :products do |t|
      t.references :category
    end
    #добавляем внешний ключ
    execute <<-SQL
    ALTER TABLE products
    ADD CONSTRAINT fk_products_categories
    FOREIGN KEY (category_id)
```



```
REFERENCES categories(id)
SQL
add_column :users, :home_page_url, :string
rename_column :users, :email, :email_address
end

def down
  rename_column :users, :email_address, :email
  remove_column :users, :home_page_url
  execute <<-SQL
    ALTER TABLE products
    DROP FOREIGN KEY fk_products_categories
  SQL
  drop_table :products
end
end
```

Иногда ваша миграция делает то, что невозможно отменить, например, уничтожает какую-либо информацию. В таких случаях можете вызвать `IrreversibleMigration` из вашего метода `down`. Если кто-либо попытается отменить вашу миграцию, будет отображена ошибка, что это не может быть выполнено.

Запуск миграций

Rails предоставляет ряд задач `rake` для работы с миграциями, которые сводятся к запуску определенных наборов миграций.

Самая первая команда `rake`, относящаяся к миграциям, которую вы будете использовать, это `rake db:migrate`. В своей основной форме она всего лишь запускает метод `up` или `change` для всех миграций, которые еще не были запущены. Если таких миграций нет, она выходит. Она запустит эти миграции в порядке, основанном на дате миграции.

Заметьте, что запуск `db:migrate` также вызывает задачу `db:schema:dump`, которая обновляет ваш файл `db/schema.rb` в соответствии со структурой вашей базы данных.

Если вы определите целевую версию, Active Record запустит требуемые миграции (методы `up`, `down` или `change`), пока не достигнет требуемой версии. Версия это числовой префикс у файла миграции. Например, чтобы мигрировать к версии 20080906120000, запустите

```
$ rake db:migrate VERSION=20080906120000
```

Если версия 20080906120000 больше текущей версии (т.е. миграция вперед) это запустит метод `up` для всех миграций до и включая 20080906120000, но не запустит какие-либо поздние миграции. Если миграция назад, это запустит метод `down` для всех миграций до, но не включая, 20080906120000.

Откат

Обычная задача это откатить последнюю миграцию, например, вы сделали ошибку и хотите исправить ее. Можно отследить версию предыдущей миграции и произвести миграцию до нее, но можно поступить проще, запустив

```
$ rake db:rollback
```

Это запустит метод `down` последней миграции. Если нужно отменить несколько миграций, можно указать параметр `STEP`:

```
$ rake db:rollback STEP=3
```

это запустит метод `down` у 3 последних миграций.

Задача `db:migrate:redo` это ярлык для выполнения отката, а затем снова запуска миграции. Так же, как и с задачей `db:rollback` можно указать параметр `STEP`, если нужно работать более чем с одной версией, например

```
$ rake db:migrate:redo STEP=3
```

Ни одна из этих команд `Rake` не может сделать ничего такого, чего нельзя было бы сделать с `db:migrate`. Они просто более удобны, так как вам не нужно явно указывать версию миграции, к которой нужно мигрировать.

Сброс базы данных

Задача `db:reset` удаляет базу данных, пересоздает ее и загружает в нее текущую схему.

Это не то же самое, что запуск всех миграций, смотрите раздел про [schema.rb](#).

Запуск определенных миграций

Если вам нужно запустить определенную миграцию (`up` или `down`), задачи `db:migrate:up` и `db:migrate:down` сделают это. Просто определите подходящий вариант и у соответствующей миграция будет вызван метод `up` или `down`, например

```
$ rake db:migrate:up VERSION=20080906120000
```

запустит метод `up` у миграции `20080906120000`. Эти задачи все еще проверяют, были ли миграции уже запущены, так, например, `db:migrate:up VERSION=20080906120000` ничего делать не будет, если Active Record считает, что `20080906120000` уже была запущена.

Изменение вывод результата запущенных миграций

По умолчанию миграции говорят нам только то, что они делают, и сколько времени это заняло. Миграция, создающая таблицу и добавляющая индекс, выдаст что-то наподобие этого

```
== CreateProducts: migrating =====
-- create_table(:products)
--> 0.0028s
== CreateProducts: migrated (0.0028s) =====
```

Некоторые методы в миграциях позволяют вам все это контролировать:

| Метод | Назначение |
|--------------------------------|--|
| <code>suppress_messages</code> | Принимает блок как аргумент и запрещает любой вывод, сгенерированный этим блоком. |
| <code>say</code> | Принимает сообщение как аргумент и выводит его как есть. Может быть передан второй булевый аргумент для указания, нужен отступ или нет. |
| <code>say_with_time</code> | Выводит текст вместе с продолжительностью выполнения блока. Если блок возвращает число, предполагается, что это количество затронутых строк. |

Например, эта миграция

```
class CreateProducts < ActiveRecord::Migration
  def change
    suppress_messages do
      create_table :products do |t|
        t.string :name
        t.text :description
        t.timestamps
      end
    end
    say "Created a table"
    suppress_messages {add_index :products, :name}
    say "and an index!", true
    say_with_time 'Waiting for a while' do
      sleep 10
      250
    end
  end
end
```

сгенерирует следующий результат

```
== CreateProducts: migrating =====
-- Created a table
--> and an index!
-- Waiting for a while
--> 10.0013s
--> 250 rows
== CreateProducts: migrated (10.0054s) =====
```

Если хотите, чтобы Active Record ничего не выводил, запуск `rake db:migrate VERBOSE=false` запретит любой вывод.

Использование моделей в ваших миграциях

При создании или обновлении данных зачастую хочется использовать одну из ваших моделей. Ведь они же существуют, чтобы облегчить доступ к лежащим в их основе данным. Это осуществимо, но с некоторыми предостережениями.

Например, проблемы происходят, когда модель использует столбцы базы данных, которые (1) в текущий момент отсутствуют в базе данных и (2) будут созданы в этой или последующих миграциях.

Рассмотрим пример, когда Алиса и Боб работают над одним и тем же участком кода, содержащим модель `Product`

Боб ушел в отпуск.

Алиса создала миграцию для таблицы `products`, добавляющую новый столбец, и инициализировала его. Она также добавила в модели `Product` валидацию на новый столбец.

```
# db/migrate/20100513121110_add_flag_to_product.rb

class AddFlagToProduct < ActiveRecord::Migration
  def change
    add column :products, :flag, :boolean
```

```
    Product.all.each do |product|
      product.update_attributes!(:flag => 'false')
    end
  end
end

# app/model/product.rb

class Product < ActiveRecord::Base
  validates :flag, :presence => true
end
```

Алиса добавила вторую миграцию, добавляющую и инициализирующую другой столбец в таблице products, и снова добавила в модели Product валидацию на новый столбец.

```
# db/migrate/20100515121110_add_fuzz_to_product.rb

class AddFuzzToProduct < ActiveRecord::Migration
  def change
    add_column :products, :fuzz, :string
    Product.all.each do |product|
      product.update_attributes! :fuzz => 'fuzzy'
    end
  end
end

# app/model/product.rb

class Product < ActiveRecord::Base
  validates :flag, :fuzz, :presence => true
end
```

Обе миграции работают для Алисы.

Боб вернулся с отпуска, и:

1. Обновил исходники – содержащие обе миграции и последнюю версию модели Product.
2. Запустил невыполненные миграции с помощью `rake db:migrate`, включая обновляющие модель Product.

Миграции не выполняются, так как при попытке сохранения модели, она попытается валидировать второй добавленный столбец, отсутствующий в базе данных на момент запуска *первой* миграции.

```
rake aborted!
An error has occurred, this and all later migrations canceled:
```

```
undefined method `fuzz' for #<Product:0x000001049b14a0>
```

Это исправляется путем создания локальной модели внутри миграции. Это предохраняет rails от запуска валидаций, поэтому миграции проходят.

При использовании искусственной модели неплохо бы вызвать `Product.reset_column_information` для обновления кэша ActiveRecord для модели Product до обновления данных в базе данных.

Если бы Алиса сделала бы так, проблем бы не было:

```
# db/migrate/20100513121110_add_flag_to_product.rb

class AddFlagToProduct < ActiveRecord::Migration
  class Product < ActiveRecord::Base
    end

  def change
    add_column :products, :flag, :integer
    Product.reset_column_information
    Product.all.each do |product|
      product.update_attributes!(:flag => false)
    end
  end
end

# db/migrate/20100515121110_add_fuzz_to_product.rb

class AddFuzzToProduct < ActiveRecord::Migration
  class Product < ActiveRecord::Base
    end

  def change
    add_column :products, :fuzz, :string
    Product.reset_column_information
    Product.all.each do |product|
      product.update_attributes!(:fuzz => 'fuzzy')
    end
  end
end
```

end

Экспорт схемы

Для чего нужны файлы схемы?

Миграции, какими бы не были они мощными, не являются авторитетным источником для вашей схемы базы данных. Это роль достается или файлу `db/schema.rb`, или файлу SQL, которые генерирует Active Record при исследовании базы данных. Они разработаны не для редактирования, они всего лишь отражают текущее состояние базы данных.

Не нужно (это может привести к ошибке) развертывать новый экземпляр приложения, применяя всю историю миграций. Намного проще и быстрее загрузить в базу данных описание текущей схемы.

Например, как создается тестовая база данных: текущая рабочая база данных выгружается (или в `db/schema.rb`, или в `db/structure.sql`), а затем загружается в тестовую базу данных.

Файлы схемы также полезны, если хотите быстро взглянуть, какие атрибуты есть у объекта Active Record. Эта информация не содержится в коде модели и часто размазана по нескольким миграциям, но собрана воедино в файле схемы. Имеется гем [annotate_models](#) автоматически добавляет и обновляет комментарии в начале каждой из моделей, составляющих схему, если хотите такую функциональность.

Типы выгрузок схемы

Есть два способа выгрузить схему. Они устанавливаются в `config/environment.rb` в свойстве `config.active_record.schema_format`, которое может быть или `:sql`, или `:ruby`.

Если выбрано `:ruby`, тогда схема храниться в `db/schema.rb`. Посмотрев в этот файл, можно увидеть, что он очень похож на одну большую миграцию:

```
ActiveRecord::Schema.define(:version => 20080906171750) do
  create_table "authors", :force => true do |t|
    t.string   "name"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "products", :force => true do |t|
    t.string   "name"
    t.text     "description"
    t.datetime "created_at"
    t.datetime "updated_at"
    t.string   "part_number"
  end
end
```

Во многих случаях этого достаточно. Этот файл создается с помощью проверки базы данных и описывает свою структуру, используя `create_table`, `add_index` и так далее. Так как он не зависит от типа базы данных, он может быть загружен в любую базу данных, поддерживаемую Active Record. Это очень полезно, если Вы распространяете приложение, которое может быть запущено на разных базах данных.

Однако, тут есть компромисс: `db/schema.rb` не может описать специфичные элементы базы данных, такие как внешний ключ (как ограничитель ссылочной целостности), триггеры или хранимые процедуры. В то время как в миграции вы можете выполнить произвольное выражение SQL, выгрузчик схемы не может воспроизвести эти выражения из базы данных. Если Вы используете подобные функции, нужно установить формат схемы `:sql`.

Вместо использования выгрузчика схемы Active Records, структура базы данных будет выгружена с помощью инструмента, предназначенного для этой базы данных (с помощью задачи `db:structure:dump` Rake) в `db/structure.sql`. Например, СУБД PostgreSQL использует утилиту `pg_dump`. Для MySQL этот файл будет содержать результат `SHOW CREATE TABLE` для разных таблиц. Загрузка таких схем это просто запуск содержащихся в них выражений SQL. По определению создается точная копия структуры базы данных. Использование формата `:sql` схемы, однако, предотвращает загрузку схемы в СУБД иную, чем использовалась при ее создании.

Выгрузки схем и контроль исходного кода

Поскольку выгрузки схем это авторитетный источник для вашей схемы базы данных, очень рекомендовано включать их в контроль исходного кода.

Active Record и ссылочная целостность

Способ Active Record требует, чтобы логика была в моделях, а не в базе данных. По большому счету, функции, такие как триггеры или внешние ключи как ограничители ссылочной целостности, которые переносят часть логики обратно в базу данных, не используются активно.

Валидации, такие как `validates_uniqueness_of`, это один из способов, которым ваши модели могут соблюдать ссылочную целостность. Опция `:dependent` в связях позволяет моделям автоматически уничтожать дочерние объекты при уничтожении родителя. Подобно всему, что работает на уровне приложения, это не может гарантировать ссылочной целостности, таким образом кто-то может добавить еще и внешние ключи как ограничители ссылочной целостности в базе данных.

Хотя Active Record не предоставляет каких-либо инструментов для работы напрямую с этими функциями, можно использовать метод `execute` для запуска произвольного SQL. Можно использовать плагины, такие как [foreigner](#), добавляющие поддержку внешних ключей в Active Record (включая поддержку выгрузки внешних ключей в `db/schema.rb`).

Валидации и колбэки Active Record

Это руководство научит, как вмешиваться в жизненный цикл ваших объектов Active Record. Вы научитесь, как осуществлять валидацию состояния объектов до того, как они будут направлены в базу данных, и как выполнять произвольные операции на определенных этапах жизненного цикла объекта.

После прочтения руководства и опробования этих концепций, надеюсь, что вы сможете:

- Понимать жизненный цикл объектов Active Record
- Использовать встроенные хелперы валидации Active Record
- Создавать свои собственные методы валидации
- Работать с сообщениями об ошибках, возникающими в процессе валидации
- Создавать методы обратного вызова (колбэки), реагирующие на события в жизненном цикле объекта
- Создавать специальные классы, инкапсулирующие общее поведение ваших колбэков
- Создавать обозреватели (обсерверы), реагирующие на события в жизненном цикле извне оригинального класса

Жизненный цикл объекта

В результате обычных операций приложения на Rails, объекты могут быть созданы, обновлены и уничтожены. Active Record дает возможность вмешаться в этот жизненный цикл объекта, таким образом, вы можете контролировать свое приложение и его данные.

Валидации позволяют вам быть уверенными, что только валидные данные хранятся в вашей базе данных. Колбэки и обозреватели позволяют вам переключать логику до или после изменения состояния объекта.

Обзор валидаций

Прежде чем погрузиться в подробности о валидациях в Rails, нужно немного понять, как валидации вписываются в общую картину.

Зачем использовать валидации?

Валидации используются, чтобы быть уверенными, что только валидные данные сохраняются в вашу базу данных. Например, для вашего приложения может быть важно, что каждый пользователь предоставил валидный электронный и почтовый адреса.

Есть несколько способов валидации данных, прежде чем они будут сохранены в вашу базу данных, включая ограничения, встроенные в базу данных, валидации на клиентской части, валидации на уровне контроллера и валидации на уровне модели:

- Ограничения базы данных и/или хранимые процедуры делают механизмы валидации зависимыми от базы данных, что делает тестирование и поддержку более трудными. Однако, если ваша база данных используется другими приложениями, валидация на уровне базы данных может безопасно обрабатывать некоторые вещи (такие как уникальность в нагруженных таблицах), которые затруднительно выполнять по другому.
- Валидации на клиентской части могут быть очень полезны, но в целом ненадежны, если используются в одиночку. Если они используют JavaScript, они могут быть пропущены, если JavaScript отключен в клиентском браузере. Однако, если этот способ комбинировать с другими, валидации на клиентской части могут быть удобным способом предоставить пользователям немедленную обратную связь при использовании вашего сайта.
- Валидации на уровне контроллера заманчиво делать, но это часто приводит к громоздкости и трудности тестирования и поддержки. Во всех случаях, когда это возможно, [держите свои контроллеры 'тощими'](#), тогда с вашим приложением будет приятно работать в долгосрочной перспективе.
- Валидации на уровне модели – лучший способ быть уверенным, что только валидные данные сохраняются в вашу базу данных. Они безразличны к базе данных, не могут быть обойдены конечным пользователем и удобны для тестирования и поддержки. Rails делает их простыми в использовании, предоставляет встроенные хелперы для общих нужд и позволяет вам создавать свои собственные валидационные методы.

Когда происходит валидация?

Есть два типа объектов Active Record: те, которые соответствуют строке в вашей базе данных, и те, которые нет. Когда создаете новый объект, например, используя метод `new`, этот объект еще не привязан к базе данных. Как только вы

вызове `save`. Этот объект будет сохранен в подходящую таблицу базы данных. Active Record использует метод экземпляра `new_record?` для определения, есть ли уже объект в базе данных или нет. Рассмотрим следующий простой класс Active Record:

```
class Person < ActiveRecord::Base
end
```

Можно увидеть, как он работает, взглянув на результат rails console:

```
>> p = Person.new(:name => "John Doe")
=> #<Person id: nil, name: "John Doe", created_at: nil, :updated_at: nil>
>> p.new_record?
=> true
>> p.save
=> true
>> p.new_record?
=> false
```

Создание и сохранение новой записи посылает операцию SQL INSERT базе данных. Обновление существующей записи вместо этого посылает операцию SQL UPDATE. Валидации обычно запускаются до того, как эти команды посылаются базе данных. Если любая из валидаций проваливается, объект помечается как недействительный и Active Record не выполняет операцию INSERT или UPDATE. Это помогает избежать хранения невалидного объекта в базе данных. Можно выбирать запуск специфичных валидаций, когда объект создается, сохраняется или обновляется.

Есть разные методы изменения состояния объекта в базе данных. Некоторые методы вызывают валидации, некоторые нет. Это означает, что возможно сохранить в базу данных объект с недействительным статусом, если вы будете не внимательны.

Следующие методы вызывают валидацию, и сохраняют объект в базу данных только если он валиден:

- `create`
- `create!`
- `save`
- `save!`
- `update`
- `update_attributes`
- `update_attributes!`

Версии с восклицательным знаком (т.е. `save!`) вызывают исключение, если запись недействительна. Невосклицательные версии не вызывают: `save` и `update_attributes` возвращают `false`, `create` и `update` возвращают объекты.

Пропуск валидаций

Следующие методы пропускают валидации, и сохраняют объект в базу данных, независимо от его валидности. Их нужно использовать осторожно.

- `decrement!`
- `decrement_counter`
- `increment!`
- `increment_counter`
- `toggle!`
- `touch`
- `update_all`
- `update_attribute`
- `update_column`
- `update_counters`

Заметьте, что `save` также имеет способность пропустить валидации, если как передать `:validate => false` как аргумент. Этот способ нужно использовать осторожно.

- `save(:validate => false)`

valid? или invalid?

Чтобы определить, валиден объект или нет, Rails использует метод `valid?`. Вы также можете его использовать для себя. `valid?` вызывает ваши валидации и возвращает `true`, если ни одной ошибки не было найдено у объекта, иначе `false`.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true
end
```

```
Person.create(:name => "John Doe").valid? # => true
Person.create(:name => nil).valid? # => false
```

После того, как Active Record выполнит валидации, все найденные ошибки будут доступны в методе экземпляра `errors`, возвращающем коллекцию ошибок. По определению объект валиден, если эта коллекция будет пуста после запуска

валидаций.

Заметьте, что объект, созданный с помощью `new` не сообщает об ошибках, даже если технически невалиден, поскольку валидации не запускаются при использовании `new`.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true
end

>> p = Person.new
=> #<Person id: nil, name: nil>
>> p.errors
=> {}

>> p.valid?
=> false
>> p.errors
=> {:name=>["can't be blank"]}

>> p = Person.create
=> #<Person id: nil, name: nil>
>> p.errors
=> {:name=>["can't be blank"]}

>> p.save
=> false

>> p.save!
=> ActiveRecord::RecordInvalid: Validation failed: Name can't be blank

>> Person.create!
=> ActiveRecord::RecordInvalid: Validation failed: Name can't be blank
```

`invalid?` это просто антипод `valid?`. `invalid?` запускает ваши валидации, возвращая `true`, если для объекта были добавлены ошибки, и `false` в противном случае.

errors[]

Чтобы проверить, является или нет конкретный атрибут объекта валидным, можно использовать `errors[:attribute]`, который возвращает массив со всеми ошибками атрибута, когда нет ошибок по определенному атрибуту, возвращается пустой массив.

Этот метод полезен только *после того*, как валидации были запущены, так как он всего лишь исследует коллекцию `errors`, но сам не вызывает валидации. Он отличается от метода `ActiveRecord::Base#invalid?`, описанного выше, тем, что не проверяет валидность объекта в целом. Он всего лишь проверяет, какие ошибки были найдены для отдельного атрибута объекта.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true
end

>> Person.new.errors[:name].any? # => false
>> Person.create.errors[:name].any? # => true
```

Мы рассмотрим ошибки валидации подробнее в разделе [Работаем с ошибками валидации](#). А сейчас обратимся к встроенным валидационным хелперам, предоставленным Rails по умолчанию.

Валидационные хелперы

Active Record предлагает множество предопределенных валидационных хелперов, которые Вы можете использовать прямо внутри Ваших определений класса. Эти хелперы предоставляют общие правила валидации. Каждый раз, когда валидация проваливается, сообщение об ошибке добавляется в коллекцию `errors` объекта, и это сообщение связывается с атрибутом, который подлежал валидации.

Каждый хелпер принимает произвольное количество имен атрибутов, поэтому в одной строке кода можно добавить валидации одинакового вида для нескольких атрибутов.

Они все принимают опции `:on` и `:message`, которые определяют, когда валидация должна быть запущена, и какое сообщение должно быть добавлено в коллекцию `errors`, если она провалится. Опция `:on` принимает одно из значений `:save` (по умолчанию), `:create` или `:update`. Для каждого валидационного хелпера есть свое сообщение об ошибке по умолчанию. Эти сообщения используются, если не определена опция `:message`. Давайте рассмотрим каждый из доступных хелперов.

acceptance

Проверяет, что чекбокс в пользовательском интерфейсе был нажат, когда форма была подтверждена. Обычно используется, когда пользователю нужно согласиться с условиями использования Вашего приложения, подтвердить прочтение некоторого текста или выполнить любое подобное действие. Валидация очень специфична для веб приложений, и

ее принятие не нужно записывать куда-либо в базу данных (если у Вас нет поля для него, хелпер всего лишь создаст виртуальный атрибут).

```
class Person < ActiveRecord::Base
  validates :terms_of_service, :acceptance => true
end
```

Для этого хелпера сообщение об ошибке по умолчанию следующее *“must be accepted”*.

Он может получать опцию `:accept`, которая определяет значение, которое должно считаться принятым. По умолчанию это `“1”`, но его можно изменить.

```
class Person < ActiveRecord::Base
  validates :terms_of_service, :acceptance => { :accept => 'yes' }
end
```

validates_associated

Этот хелпер можно использовать, когда у вашей модели есть связи с другими моделями, и их также нужно проверить на валидность. Когда вы пытаетесь сохранить свой объект, будет вызван метод `valid?` для каждого из связанных объектов.

```
class Library < ActiveRecord::Base
  has_many :books
  validates_associated :books
end
```

Эта валидация работает со всеми типами связей.

Не используйте `validates_associated` на обоих концах ваших связей, они будут вызывать друг друга в бесконечном цикле.

Для `validates_associated` сообщение об ошибке по умолчанию следующее *“is invalid”*. Заметьте, что каждый связанный объект имеет свою собственную коллекцию `errors`; ошибки не добавляются к вызывающей модели.

confirmation

Этот хелпер можно использовать, если у вас есть два текстовых поля, из которых нужно получить полностью идентичное содержание. Например, вы хотите подтверждение адреса электронной почты или пароля. Эта валидация создает виртуальный атрибут, имя которого равно имени подтверждаемого поля с добавлением `“_confirmation”`.

```
class Person < ActiveRecord::Base
  validates :email, :confirmation => true
end
```

В вашем шаблоне вьюхи нужно использовать что-то вроде этого

```
<%= text_field :person, :email %>
<%= text_field :person, :email_confirmation %>
```

Эта проверка выполняется, только если `email_confirmation` не равно `nil`. Чтобы требовать подтверждение, нужно добавить еще проверку на существование проверяемого атрибута (мы рассмотрим `presence` чуть позже):

```
class Person < ActiveRecord::Base
  validates :email, :confirmation => true
  validates :email_confirmation, :presence => true
end
```

По умолчанию сообщение об ошибке для этого хелпера такое *“doesn't match confirmation”*.

exclusion

Этот хелпер проводит валидацию того, что значения атрибутов не включены в указанный набор. Фактически, этот набор может быть любым перечисляемым объектом.

```
class Account < ActiveRecord::Base
  validates :subdomain, :exclusion => { :in => %w(www us ca jp),
    :message => "Subdomain %{value} is reserved." }
end
```

Хелпер `exclusion` имеет опцию `:in`, которая получает набор значений, которые не должны приниматься проверяемыми атрибутами. Опция `:in` имеет псевдоним `:within`, который используется для тех же целей. Этот пример использует опцию `:message`, чтобы показать вам, как можно включать значение атрибута.

Значение сообщения об ошибке по умолчанию *“is reserved”*.

format

Этот хелпер проводит валидацию значений атрибутов, тестируя их на соответствие указанному регулярному выражению, которое определяется с помощью опции `:with`.

```
class Product < ActiveRecord::Base
  validates :legacy_code, :format => { :with => /\A[a-zA-Z]+\z/,
    :message => "Only letters allowed" }
end
```

Значение сообщения об ошибке по умолчанию *“is invalid”*.

inclusion

Этот хелпер проводит валидацию значений атрибутов на включение в указанный набор. Фактически этот набор может быть любым перечисляемым объектом.

```
class Coffee < ActiveRecord::Base
  validates :size, :inclusion => { :in => %w(small medium large),
    :message => "%{value} is not a valid size" }
end
```

Хелпер `inclusion` имеет опцию `:in`, которая получает набор значений, которые должны быть приняты. Опция `:in` имеет псевдоним `:within`, который используется для тех же целей. Предыдущий пример использует опцию `:message`, чтобы показать вам, как можно включать значение атрибута.

Значение сообщения об ошибке по умолчанию для этого хелпера такое *“is not included in the list”*.

length

Этот хелпер проводит валидацию длины значений атрибутов. Он предлагает ряд опций, с помощью которых вы можете определить ограничения по длине разными способами:

```
class Person < ActiveRecord::Base
  validates :name, :length => { :minimum => 2 }
  validates :bio, :length => { :maximum => 500 }
  validates :password, :length => { :in => 6..20 }
  validates :registration_number, :length => { :is => 6 }
end
```

Возможные опции ограничения длины такие:

- `:minimum` – атрибут не может быть меньше определенной длины.
- `:maximum` – атрибут не может быть больше определенной длины.
- `:in` (или `:within`) – длина атрибута должна находиться в указанном интервале. Значение этой опции должно быть интервалом.
- `:is` – длина атрибута должна быть равной указанному значению.

Значение сообщения об ошибке по умолчанию зависит от типа выполняемой валидации длины. Можно переопределить эти сообщения, используя опции `:wrong_length`, `:too_long` и `:too_short`, и `%{count}` как место для вставки числа, соответствующего длине используемого ограничения. Можете использовать опцию `:message` для определения сообщения об ошибке.

```
class Person < ActiveRecord::Base
  validates :bio, :length => { :maximum => 1000,
    :too_long => "%{count} characters is the maximum allowed" }
end
```

По умолчанию этот хелпер считает символы, но вы можете разбить значение иным способом используя опцию `:tokenizer`:

```
class Essay < ActiveRecord::Base
  validates :content, :length => {
    :minimum => 300,
    :maximum => 400,
    :tokenizer => lambda { |str| str.scan(/\w+/) },
    :too_short => "must have at least %{count} words",
    :too_long => "must have at most %{count} words"
  }
end
```

Отметьте, что сообщения об ошибке по умолчанию во множественном числе (т.е., *“is too short (minimum is %{count} characters)”*). По этой причине, когда `:minimum` равно 1, следует предоставить собственное сообщение или использовать вместо него `validates_presence_of`. Когда `:in` или `:within` имеют как нижнюю границу 1, следует или предоставить собственное сообщение, или вызвать `presence` перед `length`.

Хелпер `size` это псевдоним для `length`.

numericality

Этот хелпер проводит валидацию того, что ваши атрибуты имеют только числовые значения. По умолчанию, этому будет

соответствовать возможный знак первым символом, и следующее за ним целочисленное или с плавающей запятой число. Чтобы определить, что допустимы только целочисленные значения, установите `:only_integer` в `true`.

Если установить `:only_integer` в `true`, тогда будет использоваться регулярное выражение

```
/\A[+-]?[0-9]+\Z/
```

для проведения валидации значения атрибута. В противном случае, он будет пытаться конвертировать значение в число, используя `Float`.

Отметьте, что вышеописанное регулярное выражение позволяет завершающий символ перевода строки

```
class Player < ActiveRecord::Base
  validates :points, :numericality => true
  validates :games_played, :numericality => { :only_integer => true }
end
```

Кроме `:only_integer`, хелпер `validates_numericality_of` также принимает следующие опции для добавления ограничений к приемлемым значениям:

- `:greater_than` – определяет, что значение должно быть больше, чем значение опции. По умолчанию сообщение об ошибке для этой опции такое *“must be greater than %{count}”*.
- `:greater_than_or_equal_to` – определяет, что значение должно быть больше или равно значению опции. По умолчанию сообщение об ошибке для этой опции такое *“must be greater than or equal to %{count}”*.
- `:equal_to` – определяет, что значение должно быть равно значению опции. По умолчанию сообщение об ошибке для этой опции такое *“must be equal to %{count}”*.
- `:less_than` – определяет, что значение должно быть меньше, чем значение опции. По умолчанию сообщение об ошибке для этой опции такое *“must be less than %{count}”*.
- `:less_than_or_equal_to` – определяет, что значение должно быть меньше или равно значению опции. По умолчанию сообщение об ошибке для этой опции такое *“must be less than or equal to %{count}”*.
- `:odd` – определяет, что значение должно быть нечетным, если установлено `true`. По умолчанию сообщение об ошибке для этой опции такое *“must be odd”*.
- `:even` – определяет, что значение должно быть четным, если установлено `true`. По умолчанию сообщение об ошибке для этой опции такое *“must be even”*.

По умолчанию сообщение об ошибке *“is not a number”*.

presence

Этот хелпер проводит валидацию того, что определенные атрибуты не пустые. Он использует метод `blank?` для проверки того, является ли значение или `nil`, или пустой строкой (это строка, которая или пуста, или содержит пробелы).

```
class Person < ActiveRecord::Base
  validates :name, :login, :email, :presence => true
end
```

Если хотите быть уверенным, что связь существует, нужно проверить, существует ли внешний ключ, используемый для связи, но не сам связанный объект.

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  validates :order_id, :presence => true
end
```

Так как `false.blank?` это `true`, если хотите провести валидацию существования булева поля, нужно использовать `validates :field_name, :inclusion => { :in => [true, false] }`.

По умолчанию сообщение об ошибке *“can't be empty”*.

uniqueness

Этот хелпер проводит валидацию того, что значение атрибута уникально, перед тем, как объект будет сохранен. Он не создает условие уникальности в базе данных, следовательно, может произойти так, что два разных подключения к базе данных создадут две записи с одинаковым значением для столбца, который вы подразумеваете уникальным. Чтобы этого избежать, нужно создать индекс `unique` в вашей базе данных.

```
class Account < ActiveRecord::Base
  validates :email, :uniqueness => true
end
```

Валидация производится путем SQL запроса в таблицу модели, поиска существующей записи с тем же значением атрибута.

Имеется опция `:score`, которую можно использовать для определения других атрибутов, используемых для ограничения проверки уникальности:

```
class Holiday < ActiveRecord::Base
```

```
validates :name, :uniqueness => { :scope => :year,  
  :message => "should happen once per year" }  
end
```

Также имеется опция `:case_sensitive`, которой можно определить, будет ли ограничение уникальности чувствительно к регистру или нет. Опция по умолчанию равна `true`.

```
class Person < ActiveRecord::Base  
  validates :name, :uniqueness => { :case_sensitive => false }  
end
```

Отметьте, что некоторые базы данных настроены на выполнение чувствительного к регистру поиска в любом случае.

По умолчанию сообщение об ошибке *“has already been taken”*.

validates_with

Этот хелпер передает запись в отдельный класс для валидации.

```
class Person < ActiveRecord::Base  
  validates_with GoodnessValidator  
end  
  
class GoodnessValidator < ActiveModel::Validator  
  def validate(record)  
    if record.first_name == "Evil"  
      record.errors[:base] << "This person is evil"  
    end  
  end  
end
```

Ошибки, добавляемые в `record.errors[:base]` относятся к состоянию записи в целом, а не к определенному атрибуту.

Хелпер `validates_with` принимает класс или список классов для использования в валидации. Для `validates_with` нет сообщения об ошибке по умолчанию. Следует вручную добавлять ошибки в коллекцию `errors` записи в классе валидатора.

Для применения метода `validate`, необходимо иметь определенным параметр `record`, который является записью, проходящей валидацию.

Подобно всем другим валидациям, `validates_with` принимает опции `:if`, `:unless` и `:on`. Если передадите любые другие опции, они будут пересланы в класс валидатора как `options`:

```
class Person < ActiveRecord::Base  
  validates_with GoodnessValidator, :fields => [:first_name, :last_name]  
end  
  
class GoodnessValidator < ActiveModel::Validator  
  def validate(record)  
    if options[:fields].any?{|field| record.send(field) == "Evil" }  
      record.errors[:base] << "This person is evil"  
    end  
  end  
end
```

validates_each

Этот хелпер помогает провести валидацию атрибутов с помощью блока кода. Он не имеет предопределенной валидационной функции. Вы должны создать ее, используя блок, и каждый атрибут, указанный в `validates_each`, будет протестирован в нем. В следующем примере нам не нужны имена и фамилии, начинающиеся с маленькой буквы.

```
class Person < ActiveRecord::Base  
  validates_each :name, :surname do |record, attr, value|  
    record.errors.add(attr, 'must start with upper case') if value =~ /\A[a-z]/  
  end  
end
```

Блок получает запись, имя атрибута и значение атрибута. Вы можете делать что угодно для проверки валидности данных внутри блока. Если валидация проваливается, следует добавить сообщение об ошибке в модель, которое делает ее невалидной.

Общие опции валидаций

Есть несколько общих опций валидаций:

:allow_nil

Опция `:allow_nil` пропускает валидацию, когда проверяемое значение равно `nil`.

```
class Coffee < ActiveRecord::Base
  validates :size, :inclusion => { :in => %w(small medium large),
    :message => "%{value} is not a valid size" }, :allow_nil => true
end
```

`:allow_nil` игнорируется валидатором `presence`.

`:allow_blank`

Опция `:allow_blank` подобна опции `:allow_nil`. Эта опция пропускает валидацию, если значение атрибута `blank?`, например `nil` или пустая строка.

```
class Topic < ActiveRecord::Base
  validates :title, :length => { :is => 5 }, :allow_blank => true
end
```

```
Topic.create("title" => "").valid? # => true
Topic.create("title" => nil).valid? # => true
```

`:allow_blank` игнорируется валидатором `presence`.

`:message`

Как мы уже видели, опция `:message` позволяет определить сообщение, которое будет добавлено в коллекцию `errors`, когда валидация проваливается. Если эта опция не используется, Active Record будет использовать соответственные сообщения об ошибках по умолчанию для каждого валидационного хелпера.

`:on`

Опция `:on` позволяет определить, когда должна произойти валидация. Стандартное поведение для всех встроенных валидационных хелперов это запускаться при сохранении (и когда создается новая запись, и когда она обновляется). Если хотите изменить это, используйте `:on => :create`, для запуска валидации только когда создается новая запись, или `:on => :update`, для запуска валидации когда запись обновляется.

```
class Person < ActiveRecord::Base
  # будет возможно обновить email с дублирующим значением
  validates :email, :uniqueness => true, :on => :create

  # будет возможно создать запись с нечисловым возрастом
  validates :age, :numericality => true, :on => :update

  # по умолчанию (проверяет и при создании, и при обновлении)
  validates :name, :presence => true, :on => :save
end
```

Условная валидация

Иногда имеет смысл проводить валидацию объекта только при выполнении заданного условия. Это можно сделать, используя опции `:if` и `:unless`, которые принимают символ, строку или Proc. Опцию `:if` можно использовать, если вы хотите определить, когда валидация **должна** произойти. Если вы хотите определить, когда валидация **не должна** произойти, воспользуйтесь опцией `:unless`.

Использование символа с `:if` и `:unless`

Вы можете связать опции `:if` и `:unless` с символом, соответствующим имени метода, который будет вызван перед валидацией. Это наиболее часто используемый вариант.

```
class Order < ActiveRecord::Base
  validates :card_number, :presence => true, :if => :paid_with_card?

  def paid_with_card?
    payment_type == "card"
  end
end
```

Использование строки с `:if` и `:unless`

Также можно использовать строку, которая будет вычислена с использованием `eval`, и должна содержать валидный код Ruby. Этот вариант следует использовать, если строка содержит действительно короткое условие.

```
class Person < ActiveRecord::Base
  validates :surname, :presence => true, :if => "name.nil?"
end
```

Использование Proc с :if и :unless

Наконец, можно связать :if и :unless с объектом Proc, который будет вызван. Использование объекта Proc дает возможность написать встроенное условие вместо отдельного метода. Этот вариант лучше всего подходит для однострочного кода.

```
class Account < ActiveRecord::Base
  validates :password, :confirmation => true,
    :unless => Proc.new { |a| a.password.blank? }
end
```

Группировка условных валидаций

Иногда полезно иметь несколько валидаций с одним условием, это легко достигается с использованием with_options.

```
class User < ActiveRecord::Base
  with_options :if => :is_admin? do |admin|
    admin.validates :password, :length => { :minimum => 10 }
    admin.validates :email, :presence => true
  end
end
```

Во все валидации внутри with_options будет автоматически передано условие :if => :is_admin?.

Выполнение собственных валидаций

Когда встроенных валидационных хелперов недостаточно для ваших нужд, можете написать свои собственные валидаторы или методы валидации.

Собственные валидаторы

Собственные валидаторы это классы, расширяющие ActiveRecord::Validator. Эти классы должны реализовать метод validate, принимающий запись как аргумент и выполняющий валидацию на ней. Собственный валидатор вызывается с использованием метода validates_with.

```
class MyValidator < ActiveRecord::Validator
  def validate(record)
    if record.name.starts_with? 'X'
      record.errors[:name] << 'Need a name starting with X please!'
    end
  end
end

class Person
  include ActiveRecord::Validations
  validates_with MyValidator
end
```

Простейшим способом добавить собственные валидаторы для валидации отдельных атрибутов является наследуемость от ActiveRecord::EachValidator. В этом случае класс собственного валидатора должен реализовать метод validate_each, принимающий три аргумента: запись, атрибут и значение, соответствующее экземпляру, соответственно атрибут тот, который будет проверяться и значение в переданном экземпляре:

```
class EmailValidator < ActiveRecord::EachValidator
  def validate_each(record, attribute, value)
    unless value =~ /\A([^\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
      record.errors[attribute] << (options[:message] || "is not an email")
    end
  end
end

class Person < ActiveRecord::Base
  validates :email, :presence => true, :email => true
end
```

Как показано в примере, можно объединять стандартные валидации со своими произвольными валидаторами.

Собственные методы

Также возможно создать методы, проверяющие состояние ваших моделей и добавляющие сообщения в коллекцию errors, когда они невалидны. Затем эти методы следует зарегистрировать, используя один или более из методов класса validate, validate_on_create или validate_on_update, передав символьные имена валидационных методов.

Можно передать более одного символа для каждого метода класса, и соответствующие валидации будут запущены в том порядке, в котором они зарегистрированы.

```
class Invoice < ActiveRecord::Base
  validate :expiration_date_cannot_be_in_the_past,
    :discount_cannot_be_greater_than_total_value

  def expiration_date_cannot_be_in_the_past
    errors.add(:expiration_date, "can't be in the past") if
      !expiration_date.blank? and expiration_date < Date.today
  end

  def discount_cannot_be_greater_than_total_value
    errors.add(:discount, "can't be greater than total value") if
      discount > total_value
  end
end
```

Можно даже создать собственные валидационные хелперы и использовать их в нескольких различных моделях. Для примера, в приложении, управляющим исследованиями, может быть полезным указать, что определенное поле соответствует ряду значений:

```
ActiveRecord::Base.class_eval do
  def self.validates_as_choice(attr_name, n, options={})
    validates attr_name, :inclusion => { :in => 1..n }.merge(options) }
  end
end
```

Просто переоткройте ActiveRecord::Base и определите подобный этому метод класса. Такой код обычно располагают где-нибудь в config/initializers. Теперь Вы можете использовать этот хелпер таким образом:

```
class Movie < ActiveRecord::Base
  validates_as_choice :rating, 5
end
```

Работаем с ошибками валидации

В дополнение к методам valid? и invalid?, раскрытым ранее, Rails предоставляет ряд методов для работы с коллекцией errors и исследования валидности объектов.

Предлагаем список наиболее часто используемых методов. Если хотите увидеть список всех доступных методов, обратитесь к документации по ActiveRecord::Errors.

errors

Возвращает экземпляр класса ActiveRecord::Errors (который ведет себя как OrderedHash), содержащий все ошибки. Каждый ключ это имя атрибута и значение это массив строк со всеми ошибками.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true, :length => { :minimum => 3 }
end

person = Person.new
person.valid? # => false
person.errors
# => {:name => ["can't be blank", "is too short (minimum is 3 characters)"]}

person = Person.new(:name => "John Doe")
person.valid? # => true
person.errors # => []
```

errors[]

errors[] используется, когда вы хотите проверить сообщения об ошибке для определенного атрибута. Он возвращает массив строк со всеми сообщениями об ошибке для заданного атрибута, каждая строка с одним сообщением об ошибке. Если нет ошибок, относящихся к атрибуту, возвратится пустой массив.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true, :length => { :minimum => 3 }
end

person = Person.new(:name => "John Doe")
person.valid? # => true
person.errors[:name] # => []

person = Person.new(:name => "JD")
person.valid? # => false
person.errors[:name] # => ["is too short (minimum is 3 characters)"]

person = Person.new
person.valid? # => false
```

```
person.errors[:name]
# => ["can't be blank", "is too short (minimum is 3 characters)"]
```

errors.add

Метод `add` позволяет вручную добавлять сообщения, которые относятся к определенным атрибутам. Можно использовать методы `errors.full_messages` или `errors.to_a` для просмотра сообщения в форме, в которой они отображаются пользователю. Эти определенные сообщения получают предшествующим (и с прописной буквы) имя атрибута. `add` получает имя атрибута, к которому вы хотите добавить сообщение, и само сообщение.

```
class Person < ActiveRecord::Base
  def a_method_used_for_validation_purposes
    errors.add(:name, "cannot contain the characters !@#%*()_+=")
  end
end
```

```
person = Person.create(:name => "!@#")
```

```
person.errors[:name]
# => ["cannot contain the characters !@#%*()_+="]
```

```
person.errors.full_messages
# => ["Name cannot contain the characters !@#%*()_+="]
```

Другой способ использования заключается в установлении `[]=`

```
class Person < ActiveRecord::Base
  def a_method_used_for_validation_purposes
    errors[:name] = "cannot contain the characters !@#%*()_+="
```

```
  end
```

```
end
```

```
person = Person.create(:name => "!@#")
```

```
person.errors[:name]
# => ["cannot contain the characters !@#%*()_+="]
```

```
person.errors.to_a
# => ["Name cannot contain the characters !@#%*()_+="]
```

errors[:base]

Можете добавлять сообщения об ошибках, которые относятся к состоянию объекта в целом, а не к отдельному атрибуту. Этот метод можно использовать, если вы хотите сказать, что объект невалиден, независимо от значений его атрибутов. Поскольку `errors[:base]` массив, можете просто добавить строку к нему, и она будет использована как сообщение об ошибке.

```
class Person < ActiveRecord::Base
  def a_method_used_for_validation_purposes
    errors[:base] << "This person is invalid because ..."
```

```
  end
```

```
end
```

errors.clear

Метод `clear` используется, когда вы намеренно хотите очистить все сообщения в коллекции `errors`. Естественно, вызов `errors.clear` для невалидного объекта фактически не сделает его валидным: сейчас коллекция `errors` будет пуста, но в следующий раз, когда вы вызовете `valid?` или любой метод, который попытается сохранить этот объект в базу данных, валидации выполнятся снова. Если любая из валидаций провалится, коллекция `errors` будет заполнена снова.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true, :length => { :minimum => 3 }
end
```

```
person = Person.new
person.valid? # => false
person.errors[:name]
# => ["can't be blank", "is too short (minimum is 3 characters)"]
```

```
person.errors.clear
person.errors.empty? # => true
```

```
p.save # => false
```

```
p.errors[:name]
# => ["can't be blank", "is too short (minimum is 3 characters)"]
```

errors.size

Метод `size` возвращает количество сообщений об ошибке для объекта.


```
class Person < ActiveRecord::Base
  validates :name, :presence => true, :length => { :minimum => 3 }
end

person = Person.new
person.valid? # => false
person.errors.size # => 2

person = Person.new(:name => "Andrea", :email => "andrea@example.com")
person.valid? # => true
person.errors.size # => 0
```

Отображение ошибок валидации во вьюхе

[DynamicForm](#) представляет хелперы для отображения сообщений об ошибке ваших моделей в ваших шаблонах вьюх.

Его можно установить как гем, добавив эту строку в Gemfile:

```
gem "dynamic_form"
```

Теперь у вас есть доступ к двум методам хелпера `error_messages` и `error_messages_for` в своих шаблонах вьюх.

`error_messages` и `error_messages_for`

При создании формы с помощью хелпера `form_for`, в нем можно использовать метод `error_messages` для отображения всех сообщений о проваленных валидациях для текущего экземпляра модели.

```
class Product < ActiveRecord::Base
  validates :description, :value, :presence => true
  validates :value, :numericality => true, :allow_nil => true
end

<%= form_for(@product) do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :description %><br />
    <%= f.text_field :description %>
  </p>
  <p>
    <%= f.label :value %><br />
    <%= f.text_field :value %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>
```

Если вы подтвердите форму с пустыми полями, результат будет похож на следующее:

New product

2 errors prohibited this product from being saved

There were problems with the following fields:

- Value can't be blank
- Description can't be blank

Description

Value

Create

Появившийся сгенерированный HTML будет отличаться от показанного, если не был использован скаффолдинг. Смотрите [Настройка CSS сообщений об ошибке](#).

Также можно использовать хелпер `error_messages_for` для отображения сообщений об ошибке от переданной в шаблон

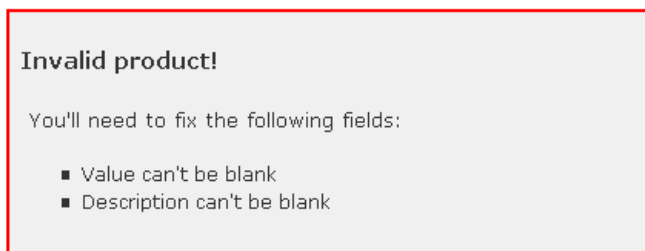
<%= error messages for :product %>

И хелпер `form.error_messages`, и хелпер `error_messages_for` принимают опции, позволяющие настроить элемент `div`, содержащий сообщения, изменить текст заголовка, сообщение после текста заголовка и определить тег, используемый для элемента заголовка.

```
<%= f.error_messages :header_message => "Invalid product!",  
      :message => "You'll need to fix the following fields:",  
      :header_tag => :h3 %>
```

приведет к

New product



Настройка CSS сообщений об ошибке

- `.field_with_errors` – стиль для полей формы и label-ов с ошибками.
- `#error_explanation` – стиль для элемента div с сообщениями об ошибках.
- `#error_explanation h2` – стиль для заголовка элемента div.
- `#error_explanation p` – стиль для параграфа, содержащего сообщение, который появляется сразу после заголовка элемента div.
- `#error_explanation ul li` – стиль для элементов списка с отдельными сообщениями об ошибках.

Имя класса и id могут быть изменены опциями :class и :id, принимаемыми обоими хелперами.

Настройка HTML сообщений об ошибке

Способ, с помощью которого обрабатываются поля формы с ошибками, определяется `ActionView::Base.field_error_proc`. Это Proc который получает два параметра:

- Строку с тегом HTML
- Экземпляр `ActionView::Helpers::InstanceTag`.

Ниже простой пример, где мы изменим поведение Rails всегда отображать сообщения об ошибках в начале каждого поля формы с ошибкой. Сообщения об ошибках будут содержаться в элементе `span` с CSS классом `validation-error`. Вокруг элемента `input` не будет никакого элемента `div`, тем самым мы избавимся от этой красной рамки вокруг текстового поля. Можете использовать CSS класс `validation-error` для стилизации, где только захотите.

```
ActionView::Base.field_error_proc = Proc.new do |html_tag, instance|
  if instance.error_message.kind_of?(Array)
    %({html_tag}<span class="validation-error">&nbsp;
      #{instance.error_message.join(',')}</span>).html_safe
  else
    %({html_tag}<span class="validation-error">&nbsp;
      #{instance.error_message}</span>).html_safe
  end
end
```

Результат будет выглядеть так:

Description

can't be blank

Колбэки

Обзор колбэков

Колбэки это методы, которые вызываются в определенные моменты жизненного цикла объекта. С колбэками возможно написать код, который будет запущен, когда объект Active Record создается, сохраняется, обновляется, удаляется, проходит валидацию или загружается из базы данных.

Регистрация колбэков

Для того, чтобы использовать доступные колбэки, их нужно зарегистрировать. Можно реализовать колбэки как обычные методы, а затем использовать макро-методы класса для их регистрации как колбэков.

```
class User < ActiveRecord::Base
  validates :login, :email, :presence => true

  before_validation :ensure_login_has_a_value

  protected
  def ensure_login_has_a_value
    if login.nil?
      self.login = email unless email.blank?
    end
  end
end
```

Макро-методы класса также могут получать блок. Это можно использовать, если код внутри блока такой короткий, что помещается в одну строку.

```
class User < ActiveRecord::Base
  validates :login, :email, :presence => true

  before_create do |user|
    user.name = user.login.capitalize if user.name.blank?
  end
end
```

Считается хорошей практикой объявлять методы колбэков как `protected` или `private`. Если их оставить `public`, они могут быть вызваны извне модели и нарушить принципы инкапсуляции объекта.

Доступные колбэки

Вот список всех доступных колбэков Active Record, перечисленных в том порядке, в котором они вызываются в течение соответствующих операций:

Создание объекта

- `before_validation`
- `after_validation`
- `before_save`
- `before_create`
- `around_create`
- `after_create`
- `after_save`

Обновление объекта

- `before_validation`
- `after_validation`
- `before_save`
- `before_update`
- `around_update`
- `after_update`
- `after_save`

Уничтожение объекта

- `before_destroy`
- `after_destroy`

- `around_destroy`

`after_save` запускается и при создании, и при обновлении, но всегда *после* более специфичных колбэков `after_create` и `after_update`, не зависимо от порядка, в котором запускаются макро-вызовы.

after_initialize и after_find

Колбэк `after_initialize` вызывается всякий раз, когда возникает экземпляр объекта Active Record, или непосредственно при использовании `new`, или когда запись загружается из базы данных. Он необходим, чтобы избежать необходимости непосредственно переопределять метод Active Record `initialize`.

Колбэк `after_find` будет вызван всякий раз, когда Active Record загружает запись из базы данных. `after_find` вызывается перед `after_initialize`, если они оба определены.

У колбэков `after_initialize` и `after_find` нет пары `before_*`, но они могут быть зарегистрированы подобно другим колбэкам Active Record.

```
class User < ActiveRecord::Base
  after_initialize do |user|
    puts "You have initialized an object!"
  end

  after_find do |user|
    puts "You have found an object!"
  end
end
```

```
>> User.new
You have initialized an object!
=> #<User id: nil>
```

```
>> User.first
You have found an object!
You have initialized an object!
=> #<User id: 1>
```

Запуск колбэков

Следующие методы запускают колбэки:

- `create`
- `create!`
- `decrement!`
- `destroy`
- `destroy_all`
- `increment!`
- `save`
- `save!`
- `save(false)`
- `toggle!`
- `update`
- `update_attribute`
- `update_attributes`
- `update_attributes!`
- `valid?`

Дополнительно, колбэк `after_find` запускается следующими поисковыми методами:

- `all`
- `first`
- `find`
- `find_all_by_attribute`
- `find_by_attribute`
- `find_by_attribute!`
- `last`

Колбэк `after_initialize` запускается всякий раз, когда инициализируется новый объект класса.

Пропуск колбэков

Подобно валидациям, также возможно пропустить колбэки. Однако, эти методы нужно использовать осторожно, поскольку важные бизнес-правила и логика приложения могут содержаться в колбэках. Пропуск их без понимания возможных последствий может привести к невалидным данным.

- `decrement`

- decrement_counter
- delete
- delete_all
- find_by_sql
- increment
- increment_counter
- toggle
- touch
- update_column
- update_all
- update_counters

Прерывание выполнения

Как только вы зарегистрировали новые колбэки в своих моделях, они будут поставлены в очередь на выполнение. Эта очередь включает все валидации вашей модели, зарегистрированные колбэки и операции с базой данных для выполнения.

Вся цепочка колбэков упаковывается в операцию. Если любой метод *before* колбэков возвращает false или вызывает исключение, выполняемая цепочка прерывается и запускается ROLLBACK; Колбэки *after* могут достичь этого, только вызвав исключение.

Вызов произвольного исключения может прервать код, который предполагает, что save и тому подобное не будут провалены подобным образом. Исключение ActiveRecord::Rollback чуть точнее сообщает Active Record, что происходит откат. Он подхватывается изнутри, но не перевызывает исключение.

Относительные колбэки

Колбэки работают с отношениями между моделями, и даже могут быть определены ими. Представим пример, где пользователь имеет много публикаций. Публикации пользователя должны быть уничтожены, если уничтожается пользователь. Давайте добавим колбэк after_destroy в модель User через ее отношения с моделью Post.

```
class User < ActiveRecord::Base
  has_many :posts, :dependent => :destroy
end

class Post < ActiveRecord::Base
  after_destroy :log_destroy_action

  def log_destroy_action
    puts 'Post destroyed'
  end
end

>> user = User.first
=> #<User id: 1>
>> user.posts.create!
=> #<Post id: 1, user_id: 1>
>> user.destroy
Post destroyed
=> #<User id: 1>
```

Условные колбэки

Как и в валидациях, возможно сделать вызов метода колбэка условным от удовлетворения заданного условия. Это осуществляется при использовании опций :if и :unless, которые могут принимать символ, строку или Proc. Опцию :if следует использовать для определения, при каких условиях колбэк **должен** быть вызван. Если вы хотите определить условия, при которых колбэк **не должен** быть вызван, используйте опцию :unless.

Использование :if и :unless с символом

Опции :if и :unless можно связать с символом, соответствующим имени метода условия, который будет вызван непосредственно перед вызовом колбэка. При использовании опции :if, колбэк не будет выполнен, если метод условия возвратит false; при использовании опции :unless, колбэк не будет выполнен, если метод условия возвратит true. Это самый распространенный вариант. При использовании такой формы регистрации, также возможно зарегистрировать несколько различных условий, которые будут вызваны для проверки, должен ли запуститься колбэк.

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, :if => :paid_with_card?
end
```

Использование :if и :unless со строкой

Также возможно использование строки, которая будет вычислена с помощью eval, и, следовательно, должна содержать валидный код Ruby. Этот вариант следует использовать только тогда, когда строка представляет действительно короткое

условие.

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, :if => "paid_with_card?"
end
```

Использование :if и :unless с Proc

Наконец, можно связать :if и :unless с объектом Proc. Этот вариант более всего подходит при написании коротких методов, обычно в одну строку.

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number,
    :if => Proc.new { |order| order.paid_with_card? }
end
```

Составные условия для колбэков

При написании условных колбэков, возможно смешивание :if и :unless в одном объявлении колбэка.

```
class Comment < ActiveRecord::Base
  after_create :send_email_to_author, :if => :author_wants_emails?,
    :unless => Proc.new { |comment| comment.post.ignore_comments? }
end
```

Классы колбэков

Иногда написанные вами методы колбэков достаточно полезны для повторного использования в других моделях. ActiveRecord делает возможным создавать классы, включающие методы колбэка, так, что становится очень легко использовать их повторно.

Вот пример, где создается класс с колбэком after_destroy для модели PictureFile:

```
class PictureFileCallbacks
  def after_destroy(picture_file)
    if File.exists?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

При объявлении внутри класса, как выше, методы колбэка получают объект модели как параметр. Теперь можем использовать класс колбэка в модели:

```
class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks.new
end
```

Заметьте, что нам нужно создать экземпляр нового объекта PictureFileCallbacks, после того, как объявили наш колбэк как отдельный метод. Это особенно полезно, если колбэки используют состояние экземпляра объекта. Часто, однако, более подходящим является иметь его как метод класса.

```
class PictureFileCallbacks
  def self.after_destroy(picture_file)
    if File.exists?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

Если метод колбэка объявляется таким образом, нет необходимости создавать экземпляр объекта PictureFileCallbacks.

```
class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks
end
```

Внутри своего колбэк-класса можно создать сколько угодно колбэков.

Обсерверы

Обсерверы похожи на колбэки, но с важными отличиями. В то время как колбэки внедряют в модель свой код, непосредственно не связанный с нею, обсерверы позволяют добавить ту же функциональность, без изменения кода модели. Например, можно утверждать, что модель User не должна включать код для рассылки писем с подтверждением регистрации. Всякий раз, когда используются колбэки с кодом, прямо не связанным с моделью, возможно, Вам захочется вместо них создать обсервер.

Создание обсерверов

Например, рассмотрим модель `User`, в которой хотим отправлять электронное письмо всякий раз, когда создается новый пользователь. Так как рассылка писем непосредственно не связана с целями нашей модели, создадим обсервер, содержащий код, реализующий эту функциональность.

```
$ rails generate observer User
```

генерирует `app/models/user_observer.rb`, содержащий класс обсервера `UserObserver`:

```
class UserObserver < ActiveRecord::Observer
end
```

Теперь можно добавить методы, вызываемые в нужных случаях:

```
class UserObserver < ActiveRecord::Observer
  def after_create(model)
    # код для рассылки подтверждений email...
  end
end
```

Как в случае с классами колбэков, методы обсервера получают рассматриваемую модель как параметр.

Регистрация обсерверов

По соглашению, обсерверы располагаются внутри директории `app/models` и регистрируются в вашем приложении в файле `config/environment.rb`. Например, `UserObserver` будет сохранен как `app/models/user_observer.rb` и зарегистрирован в `config/environment.rb` таким образом:

```
# Activate observers that should always be running.
config.active_record.observers = :user_observer
```

Естественно, настройки в `config/environments` имеют преимущество перед `config/environment.rb`. Поэтому, если вы предпочитаете не запускать обсервер для всех сред, можете его просто зарегистрировать для определенной среды.

Совместное использование обсерверов

По умолчанию Rails просто убирает “Observer” из имени обсервера для поиска модели, которую он должен рассматривать. Однако, обсерверы также могут быть использованы для добавления обращения более чем к одной модели, это возможно, если явно определить модели, который должен рассматривать обсервер.

```
class MailerObserver < ActiveRecord::Observer
  observe :registration, :user

  def after_create(model)
    # code to send confirmation email...
  end
end
```

В этом примере метод `after_create` будет вызван всякий раз, когда будет создан `Registration` или `User`. Отметьте, что этот новый `MailerObserver` также должен быть зарегистрирован в `config/environment.rb`, чтобы вступил в силу.

```
# Activate observers that should always be running.
config.active_record.observers = :mailer_observer
```

Транзакционные колбэки

Имеются два дополнительных колбэка, которые включаются по завершению транзакции базы данных: `after_commit` и `after_rollback`. Эти колбэки очень похожи на колбэк `after_save`, за исключением того, что они не запускаются пока изменения в базе данных не будут подтверждены или отменены. Они наиболее полезны, когда вашим моделям `active record` необходимо взаимодействовать с внешними системами, не являющимися частью транзакции базы данных.

Рассмотрим, к примеру, предыдущий пример, где модели `PictureFile` необходимо удалить файл после того, как запись уничтожена. Если что-либо вызовет исключение после того, как был вызван колбэк `after_destroy`, и транзакция откатывается, файл будет удален и модель останется в противоречивом состоянии. Например, предположим, что `picture_file_2` в следующем коде не валидна, и метод `save!` вызовет ошибку.

```
PictureFile.transaction do
  picture_file_1.destroy
  picture_file_2.save!
end
```

Используя колбэк `after_commit`, можно учесть этот случай.

```
class PictureFile < ActiveRecord::Base
  attr_accessor :delete_file
end
```

```
after_destroy do |picture_file|
  picture_file.delete_file = picture_file.filepath
end

after_commit do |picture_file|
  if picture_file.delete_file && File.exist?(picture_file.delete_file)
    File.delete(picture_file.delete_file)
    picture_file.delete_file = nil
  end
end
end
```

Колбэки `after_commit` и `after_rollback` гарантируют, что будут вызваны для всех созданных, обновленных или удаленных моделей внутри блока транзакции. Если какое-либо исключение вызовется в одном из этих колбэков, они будут проигнорированы, чтобы не препятствовать другим колбэкам. По сути, если код вашего колбэка может вызвать исключение, нужно для него вызвать `rescue`, и обработать его нужным образом в колбэке.