
Ruby
on
Rails

Содержание

Как читать эту книгу

Проще всего объяснить принцип — продемонстрировав пример (именно так мы и будем поступать в дальнейшем). В этой книге мы будем использовать в качестве примера интернет-магазин, над которым мы якобы работаем. В отличие от других книг, этот пример никогда не будет завершен и более того, его разработка на протяжении книги не будет последовательной. Я просто буду говорить — "в нашем интернет магазине" и дальше приводить пример задачи, которую надо решить или пример кода, который нужно написать. Мы не будем держать в памяти и учитывать все детали интернет-магазина или все те изменения, которые мы внесли в него до этого. В то же время, поскольку интернет-магазин является довольно понятным большинству пользователей (и, полагаю, всем моим читателям) приложением, его общая структура, задачи и концепции известны и понятны всем. Именно поэтому мне не нужно будет тратить время на объяснение *предметной области*[1] и логики и я смогу сосредоточиться на объяснении того, как решить конкретную известную всем задачу (скажем, добавление товара в корзину) с помощью Ruby On Rails.

Сама книга условно разделена на три части. В главах с 1-ой по 3-ю вы можете "откинуться на спинку кресла" и попытаться "впитать" теоретический материал настолько, насколько это возможно. С 3-ей по 6-ую главу я буду рассказывать о самых важных частях Rails — моделях, контроллерах и views. К сожалению, изучать их последовательно и по отдельности невозможно. Моя рекомендация будет следующей: после завершения 3-ей главы попробуйте читать 4-ую, 5-ую и 6-ую — одновременно, по немногу продвигаясь вперед. Это поможет вам не потерять из виду общую картину архитектуры MVC и окончательно не запутаться. Наконец, с 7-ой главы вы можете продолжить читать в обычном режиме.

Чтобы прочесть и понять эту книгу, вам потребуется как минимум знание языка Ruby.

[1] Предметная область — все, что имеет отношение к задаче, ради которой делается приложение. Например, если бы мы разрабатывали приложение для авиадиспетчеров, то предметная область называлась бы "регулирование воздушного движения". Если бы мы делали интернет-банк, то предметная область называлась бы "обслуживание клиентов банка". В нашем случае предметной областью будет "интернет-торговля".

Соглашения, принятые в книге

Пожалуйста обратите внимание на несколько соглашений, которые используются в книге и которые необходимо понимать:

- Ruby, html, css и другой код, а также команды для терминала могут быть вынесены в отдельный блок, например так:

```
item1.name  = "Porsche"
```

```
item1.name  = "A really fast car"
item1.price = 3000000
```

- Код также может встречаться прямо внутри предложения; тогда он может выглядеть так: `item1 = Item.last`
- Символ `#`, который используется внутри блока с кодом внутри предложения — означает не комментарий, а **инстансный метод**. Например, если речь в предложении идет о классе `Item`, то я могу написать "и затем вам нужно вызвать на объекте `Item` метод `#save`" или я могу также написать "и затем вам нужно вызвать метод `Item#save`", что будет означать ровно то же самое. Это повсеместно принятое в литературе по Ruby обозначение инстансных методов. Внутри большого блока кода, символ `#` будет означать комментарий.
- Аналогично предыдущему соглашению, точка `.` означает метод класса. Например, я могу написать "после того как вы вызовите метод `Item.create`...".
- Внутри блоков с кодом вам может встретиться подобное выражение:

  ```
  @item.id #=> 1
  ```

  Это стандартный способ, принятый в документации Ruby-библиотек для того, чтобы прямо рядом с исходным кодом показать, каков будет результат выполнения этого кода. Т.к. символ `#` в Ruby означает комментарий, то все, что следует за ним — игнорируется программой и предназначено исключительно для глаз программиста. В данном случае я хотел показать, что если вызвать на переменной `@item` метод `#id`, то он вернет значение `1`.
- Так как некоторые файлы могут состоять из большого количества кода, неразумно было бы приводить его целиком каждый раз. Я буду использовать три точки `...`, когда я подразумеваю наличие какого-либо кода в том месте, где указаны эти точки. Например:

  ```
  class Item < ActiveRecord::Base
    ...
    validates :price, numericality: { greater_than: 0 }
    ...
  end
  ```

Несколько слов перед тем, как мы приступим

Сообщения об ошибках, неточностях или опечатках большая просьба присылать на мой адрес **roman.snitko@gmail.com**

Глава 1. Архитектура веб-приложения

Преждем чем приступать к разработке своего собственного веб-приложения, я думаю, важно разобраться, что такое веб-приложения и как они обычно организованы — то есть посмотреть на картину целиком. Именно этим мы и займемся в первой главе.

Что называют веб-приложением?

С термином *веб-приложение* существует некоторая путаница, как правило, именно среди профессионалов. Пользователи, которые просто смотрят страницы в интернете до недавнего времени и вовсе не слышали термин (в особенности это касается России). Для большинства из них существует единственный понятный термин — *сайт*. И действительно, технически, любое веб-приложение является сайтом. Однако ключевая разница состоит в следующем: веб-приложение умеет взаимодействовать с пользователем и, чаще всего, обрабатывать данные, которые пользователь предоставляет этому приложению.

Рассмотрим пример сайта организации X. На сайте есть несколько страниц, например "О компании", "Контакты" и "Наши клиенты". Страницы связаны между собой ссылками, кликнув на которые пользователь может перейти к интерейсующей его странице. На сайте есть несколько фотографий и email для связи с организацией. Такой сайт не является веб-приложением: технически пользователь действительно посылает данные на сервер — он отправляет серверу url интересующей его страницы и сервер посылает назад в браузер желаемую страницу. Однако помимо этого взаимодействия ничего, на самом деле, не происходит. Такие сайты обычно называют *статичными сайтами*, потому что все, что они позволяют делать — смотреть страницы, которые никак не изменяются с прошествием времени.

Перейдем к сайту организации Y. Этот сайт позволяет пользователю зарегистрироваться через специальную форму с двумя полями: "email" и "пароль". После регистраци пользователь будет видеть свой email, например в верхнем правом углу сайта, и у него появится возможность заказывать и оплачивать товары, производимые организацией Y. Чтобы уметь это делать, сайт должен знать как поступать с приходящими от пользователя данными: например, куда сохранять логин и пароль и сведения о заказах; как проверить, что введенный пользователем email действительно похож на email; как дать возможность пользователю заплатить за товар с помощью его кредитной карты. Такой сайт можно назвать веб-приложением. Но, конечно же, обычные пользователи вряд ли станут называть

Вконтакте или, скажем, Википедию — веб-приложением. Тем не менее, каждый из этих сайтов требует логики для обработки вводимых пользователем данных, которая реализуется с помощью того или иного языка программирования. И, значит, данные сайты являются программами, т.е. приложениями.

Однако даже профессионалы не всегда соглашаются с таким значением термина *веб-приложение*. В последние несколько лет на рынке стало появляться большое количество веб-приложений, которые заменяют привычные нам десктопные приложения и работают таким образом, что пользователю не приходится перезагружать страницу в браузере или кликать по каким-то ссылкам, чтобы получить больше информации (об этом чуть позднее в разделах Чем веб-приложение отличается от других видов приложений? и Одностраничные веб-приложения и почему Rails не подходит для их создания). Именно потому, что эти приложения ведут себя как десктопные приложения, а Вконтакте и Википедия — нет, некоторые разработчики продолжают считать последние недо-веб-приложениями. Мое мнение заключается в следующем: если потребовалось написать код (на любом языке программирования), чтобы сайт заработал, то перед нами веб-приложение.

Фреймворк Ruby On Rails был изначально разработан для создания веб-приложений, которые для большинства действий требуют перехода на новый url внутри сайта и загрузки новой страницы. Это тот тип приложений на сегодняшний день в большей степени подходит для социальных сайтов, интернет-магазинов или SaaS решений для бизнеса.

Однако чтобы понять, почему именно веб-приложениям нужно загружать новые страницы, я предлагаю перейти к следующему разделу и изучить различия между традиционными десктопными приложениями, мобильными приложениями и веб-приложениями.

Чем веб-приложение отличается от других видов приложений?

В 90-х, когда десктопные компьютеры стали появляться в домах обычных людей, с интернетом, прямо скажем, было еще туго. Практически все, что должно было уметь делать приложение, оно должно было уметь делать на локальном компьютере без доступа в интернет.

Ключевой вопрос для практически любого приложения — где хранить данные, которые передал ему пользователь? В отсутствие инернета этот вопрос решался просто: хранить локально. В виде файла, формат которого определяет автор приложения; либо записывать в локальную базу данных, формат которой определялся разработчиком этой СУБД. Далее, даже без интернета, перед пользователями часто вставал вопрос обмена данными с другими

компьютерами и решался он с помощью внешних накопителей: записываем файл на диск, несем другу, он копирует его себе.

Естественно, такой подход в первую очередь был неудобен для тех приложений, где данные должны были обновляться очень часто, а сам объем пакетов с обновлениями был сравнительно небольшой — это конечно, прежде всего, всевозможный софт для бизнеса. Так что, разумеется, десктопные приложения начали учиться работать с сетью. То есть программа была установлена локально, и, возможно, она даже хранила данные локально, но при этом она использовала сеть для синхронизации локальных данных с другими компьютерами. Или вообще полностью хранила данные удаленно, на другом компьютере (сервере) каждые несколько минут или секунд обновляя экран пользователя новой информацией, которую это приложение получало из сети от сервера. Таким образом, приложения работающие с сетью по сути оказались разделены на две части: клиентскую часть (программа, запускаемая на локальной машине) и серверная часть (программа, запускаемая на сервере и принимающая данные от программы на локальной машине). То есть две эти части как бы являются единым целом и составляют одно ПО.

Однако сразу же становится очевидно, что очень многие задачи, которые выполняет такой софт не требуют доступа к ресурсам локальной машины. Например, Amazon.com конечно мог бы создать приложение для Windows и заставить всех пользователей скачать его, но в этом не было бы никакого смысла — на каждой машине, в том числе под управлением других ОС, есть какой-нибудь браузер, в который они могут загрузить веб-приложение и продавать товары через него. В этом случае, пользователю не нужно скачивать и устанавливать софт на собственный компьютер (многие не знают, как это делается или не имеют доступа) — достаточно набрать адрес в браузере. Кроме того, у вас сразу отпадает вопрос обновления старых версий приложения. Если старая версия вашего десктопного приложения (в данном контексте принято называть его *клиентом*) не поддерживает новую версию вашего ПО на сервере, то вам придется либо огорчить пользователя, либо ждать пока он решит обновиться (что может никогда не произойти) и поддерживать две версии серверного ПО. В случае с веб-приложением, "обновление" произойдет как только пользователь обновит страницу или снова зайдет на сайт.

С другой стороны, ряд приложений используют ресурсы компьютера и требуют производительной видеокарты, аудиокарты или быстрого процессора (например, приложения для 3D-моделирования, обработки видео и аудио, графические редакторы). Теоретически, такие приложения возможно создавать для современных браузеров в виде веб-приложений (и это делают), но для меня кажется совершенно очевидным, что на данный момент это относительно бесполезно и экономически неоправданно. Такие приложения не ставят целью взаимодействие с другими пользователями и получение данных от них — как правило над проектом в таком приложении работает один человек или команда, находящаяся в одном помещении. Плюс к этому, поскольку браузер предоставляет

ограниченный доступ к системе, страдают производительность и функционал — т.е. приложение тормозит и не умеет делать всего того, что умеет делать его десктопный собрат.

Вообще эта особенность браузера — не давать доступ к системе — очень важная вещь. Эта концепция называется *sandbox* (песочница) и это является большим преимуществом веб-приложений. Пользователь, скачавший приложение и запустивший его на своей машине, подвергает себя куда большей опасности, потому что, по сути, скачанный софт получает доступ ко всем файлам пользователя (эту проблему частично решают на мобильных устройствах, предварительно уведомляя пользователя о том, к каким частям системы будет иметь доступ приложение). Веб-приложение работает только внутри браузера и без ведома пользователя не может получить доступ ни к какой информации на жестком диске компьютера. Таким образом, веб-приложения, с точки зрения обычного пользователя, по умолчанию гораздо безопаснее. Безусловно, есть вероятность что в будущем, браузеры будут позволять веб-приложениям запрашивать больше прав для доступа к локальным ресурсам пользователя (по аналогии, опять-таки, с мобильными приложениями), но на данный момент это не так.

Все сказанное выше о десктопных приложениях по большей части относится и к мобильным. В целом, мобильное приложение по идеологии мало чем отличается от десктопного — оно также скачивается и устанавливается на локальный компьютер (смартфон или планшет) и хранит свои данные локально, имея при этом возможность обмениваться данным с сетью. Десктопные и мобильные приложения называют *нативными*, потому что они умеют работать напрямую с ОС, куда они установлены, без посредничества третьего ПО (коим, в случае веб-приложения, является браузер). Иногда может так получиться, что веб-приложение не отличить от нативного: например официальное приложение Gmail для iOS — это веб приложение: иконка аппа просто запускает браузер и автоматически направляет вас по адресу, на котором "живет" веб-приложение Gmail. И сделано оно так хорошо, что понять его "ненативную" сущность удается не с первого раза.

Однако отличить все-таки возможно. И в основном это получается сделать потому, что веб-приложению обычно недоступны все те *виджеты* (элементы управления, такие как кнопки, прогресс-бары и пр.), которые доступны нативному приложению. Максимум, что может сделать веб-приложение в плане интерфейса — попытаться использовать ограниченный набор нативных виджетов, доступных через браузер, или имитировать дизайн нативных виджетов. Лично я считаю подход с итимитацией виджетов в корне неправильным, потому что в современном мире люди пользуются более чем одной ОС на конкретном типе устройств — будет странно, если ваше веб-приложение будет выглядеть как iOS app на устройстве Android.

Наконец, вернемся к вопросу хранения данных, который я назвал ключевым для любого приложения. На сегодняшний день веб-приложения по большей части хранят данные на

удаленном сервере. Если веб-приложение не успело сохранить данные на сервер до того, как вы перешли на другой сайт, есть шанс, что данные будут потеряны. Естественно, этот негативный момент не мог долго оставаться без внимания и уже сегодня в современных браузерах возможно хранить локальную копию данных в т.н. local storage, которое сохраняет всю информацию даже после закрытия браузера. Но эта тема, к сожалению, выходит за рамки этой книги.

Предлагаю подвести итог. Вот что отличает веб-приложения от нативных:

- Веб-приложения предпочитают выбирать для хранения данных в первую очередь удаленный сервер, только потом локальное хранилище. Нативные приложения — как правило наоборот.
- Веб-приложения не требуют значительных ресурсов компьютера и имеют ограниченный доступ к информации на жестком диске пользователя.
- Веб-приложения не могут использовать многие нативные виджеты операционной системы.

Хотелось бы отметить, что сейчас существует тренд делать нативные приложения там, где можно было прекрасно обойтись веб-приложением. Например, Твиттер в свое время выпустил приложение для MacOSX. Безусловно, пользователи этого приложения получили некий незначительный новый функционал, который отсутствовал в веб-версии. Но нужно понимать если вы не твиттер, то у вас, скорее всего не будет ни денег ни времени следовать этой моде и разрабатывать и поддерживать такое нативное приложение, просто потому что это круто. Веб-приложения тем и хороши, что, если они сделаны грамотно, они работают практически на любой платформе и в любом браузере.

Пожалуй, главная мысль, которую мне хотелось бы донести, состоит в следующем: решение о том, какое приложение разрабатывать — нативное или веб — должно зависеть не от моды, а от природы самого приложения и задач, которые оно решает — это определяется минимум теми тремя факторами, которые были перечислены выше.

Из каких частей состоит современное веб-приложение?

Архитектура современного веб-приложения не сильно отличается от архитектуры любого другого приложения, которое имеет клиентскую и серверную часть. В роли клиентской части выступает то, что загружается в браузер — это может быть простая страница с формой (например формой регистрации) или достаточно сложное клиентское приложение

написанное на языке Javascript. В роли серверной части выступает программа, которая работает на сервере — в нашем случае это будет приложение Ruby On Rails.

Отличие от, скажем, мобильного приложения здесь состоит в том, что серверная часть приложения сохраняет некий контроль над клиентской частью — если на сервер приходит тот или иной запрос, сервер как правило решает какую страницу загрузить в браузер пользователя. В случае с мобильным приложением, пользователь один раз самостоятельно загружает апп и затем само загруженное приложение на основе полученных от сервера данных решает что показывать пользователю на экран. Безусловно, примерно такого же результата можно достичь и с веб-приложением (в частности, так в большей степени ведут себя одностраничные веб-приложения), но отличие все равно сохраняется — мобильное или десктопное приложение, как правило, загружается не с того же сервера, где находится серверная часть приложения, а, например, из АппСтора, и серверная часть вообще не знает, что кто-то загрузил приложение себе на локальный компьютер. Rails-приложения, на каждый не-ajax-запрос посылают в браузер пользователю новую страницу, то есть клиентская часть приложения очень сильно связана с серверной частью. Хорошо это или плохо — вопрос не вкуса, а конкретной задачи, стоящей перед авторами приложения.

И так, мы выяснили, что у веб-приложения есть серверная и клиентская части. Теперь подробнее рассмотрим каждую из них.

Серверная часть веб-приложения

Серверная часть приложения в случае *стэка*[1] Ruby On Rails довольно сложная. Она состоит минимум из *веб-сервера*[2], который принимает запросы, приходящие из браузера (чаще всего используется Nginx или Apache) и затем распределяет их между несколькими загруженными в память экземпелярами другого специализированного веб-сервера (чаще всего Passenger, Unicorn или Thin), который, в свою очередь уже умеет запускать приложение на Ruby On Rails. По сути, экземпляр такого специализированного веб-сервера можно рассматривать как экземпляр RubyOnRails приложения и, если посмотреть на количество памяти доступной на вашем сервере и количество памяти занимаемое экземпляром такого приложения, можно понять, сколько экземпляров приложения можно запустить на вашем сервере без того, чтобы ваш сервер начал использовать *swap*[3].

Возникает вопрос — зачем держать несколько экземпляров приложения в памяти? Очень просто: чтобы обрабатывать больше одновременных запросов. Если ваш сайт не очень популярный, вам вполне может быть достаточно пары экземпляров приложения и дешевого хостинга. Если нагрузка растет и на сайт начинает ходить больше посетителей, вероятность одновременного запроса от двух и более посетителей растет, а это значит, что кому-то из них придется ждать загрузки страницы дольше — пока не освободится занятый другим

посетителем экземпляр приложения (это произойдет как только первый пользователь получит в браузер страницу).

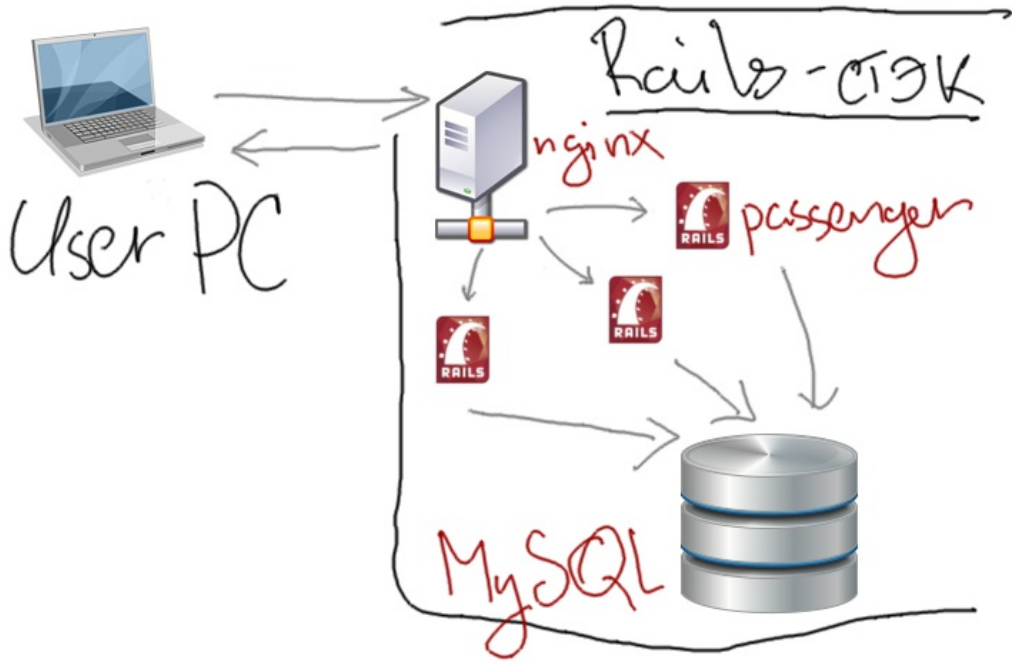
В общем, если говорить коротко, то алгоритм следующий: Nginx или Apache получают от пользователя http-запрос, далее этому запросу находится свободный экземпляр приложения и все данные передаются ему.

И это только начало истории. Самое интересное происходит дальше: ваше приложение обрабатывает запрос (как именно — мы обсудим в следующей главе) и решает, что делать дальше. Например, если пользователь прислал запрос, в котором говорится, что он хочет посмотреть страницу 2 с телевизорами из вашего интернет-магазина, приложение решит, что ему нужно сделать запрос к Базе Данных и получить оттуда следующие 10 телевизоров для второй страницы. База Данных (далее БД) — еще один ключевой элемент стэка, в ней хранятся почти все данные, которые использует ваше приложение — например пароли и личные данные пользователей, информация о товарах, список категорий товаров, пути к файлам с изображениями товаров и т.п. Под БД обычно подразумевают специализированную программу, которая умеет эффективно добавлять, хранить и извлекать данные из специального хранилища. Какие данные нужно извлечь или записать в БД определяется *запросом*, написанном на языке базы данных. Если вы только начинаете разбираться в Ruby On Rails, волноваться не стоит — приложение напишет и отправит такой запрос за вас, еще один язык (SQL) учить не придется.

Базы Данных (или как их еще называют Системы Управления Базами Данных — СУБД) бывают разные. Два основных типа — реляционные (SQLite, MySQL, PostgreSQL) и NoSQL (MongoDB CouchDB). Реляционные базы данных являются наиболее распространенным выбором для RubyOnRails-приложений и их основное преимущество — возможность выполнения сложных запросов (чтобы их выполнять, правда, вам потребуется хотя бы немного разобраться в языке запросов — перечисленные здесь реляционные БД используют язык SQL с небольшими модификациями для каждой БД).

После того, как запрос ушел в базу данных, приложение ждет ответа. Как правило, запрос к БД является т.н. узким местом стэка и "съедает" больше всего времени — чем сложнее запрос к БД, тем больше времени может уйти на его обработку. Наконец, когда ответ приходит — а приходит он в виде выбранных данных — ваше приложение решает что делать с полученными данными. Как правило, оно решит вставить эти данные в заранее заготовленный html-шаблон и отправить получившуюся страницу назад серверу Nginx или Apache, который в свою очередь уже отправит ее пользователю в браузер. Таким образом, пользователь увидит у себя на экране страницу с десятью телевизорами начиная по порядку с 11-ого по 20-ый (это 2-ая страница, помните?).

Вот небольшая схема, которая иллюстрирует описанную выше работу всего стэка на сервере:



Как установить все необходимое ПО Rails-стэка рассказано в главе [Настройка сервера и деплой приложения]

Клиентская (браузерная) часть веб-приложения.

Как я уже сказал ранее, клиентская часть приложения в случае RubyOnRails сравнительно небольшая и в основном состоит из html-разметки, возвращаемой сервером в браузер пользователя. Но даже тут все несколько сложнее, чем может показаться на первый взгляд. Если вы знакомы с языком разметки html, то вы знаете, что внутри каждой html-страницы есть специальные элементы, которые ссылаются на файлы с изображениями, css-стили или javascript-файлы. Когда браузер добирается до этих элементов, он начинает отправлять на сервер новые запросы, которые говорят серверу вернуть браузеру эти файлы. Именно поэтому изображения на страницах сайтов появляются не сразу (особенно это заметно при плохом соединении) — браузер сначала получает от сервера html-страницу, а затем подгружает все необходимые этой странице элементы. Таким образом получается, что клиентская часть приложения может состоять из следующих частей:

- html-страница
- css-стили
- изображения

- javascript

С css и изображениями все более менее понятно (css по сути определяет внешний вид страницы, в то время как html-тэги определяют ее структуру). Javascript может использоваться в веб-приложениях для разных целей, две самые распространенные это: 1) манипулирование DOM-элементами (например показать или спрятать какую-нибудь кнопку или изображение) и 2) *ajax-запросы*[4].

Одностраничные веб-приложения обычно имеют т.н. *толстый* клиент. Чаще всего это означает, что браузер загружает сложную и большую javascript-программу. В этом случае, веб-приложение становится больше похоже на мобильное или десктопное приложение, где именно клиентская часть решает, что именно и когда показывать пользователю; и пользователь может и не подозревать, что клиентская часть приложения в его браузере в данный момент посылает запрос к серверу.

В случае Rails-приложения, чаще всего, запросы на сервер посылаются явно — когда пользователь щелкает по ссылке и нажимает на кнопку возле заполненной им формы. Тогда браузер явно показывает пользователю, что он загружает новую страницу. О веб-приложениях, где большинство действий, которые совершает пользователь, требуют перезагрузки страницы говорят, что у них *тонкий* клиент. В целом, можно сказать, что поскольку возвращаемые Rails-приложением страницы отличаются друг от друга (т.е. например первая страница с телевизорами явно отличается от второй страницы, где будут представлены уже другие телевизоры), то у нас есть одно серверное приложение и много маленьких клиентских приложений — это несколько радикальное заявление, с которым могут не согласиться мои коллеги, но мне кажется, что для более глубокого понимания проблемы такое объяснение будет кстати. В реальной жизни при разработке Rails-приложения вы, конечно же, не будете думать об этом таким образом.

Наконец надо упомянуть, что иногда путают асинхронные (ajax) запросы и обычные запросы и происходит это потому, что при выполнении асинхронного запроса на экране у пользователя все-равно появляется индикатор (например крутящееся колесико возле нажатой им кнопки). Нужно понимать, что такие индикаторы добавлены в клиентскую часть программистом и момент их вывода на экран определен в javascript-программе, которая загружается вместе с html-страницей. Более того, появление такого индикатора вовсе не является гарантией, что асинхронный запрос вообще выполняется. Я могу написать в своей javascript-программе, что индикатор должен появляться возле кнопки после нажатия, но не написать ни строки кода, выполняющего собственно ajax-запрос. Или наоборот, я могу написать код для ajax-запроса, который будет выполняться после нажатия на кнопку под формой, но при этом не написать код, который будет показывать пользователю индикатор (на некоторых сайтах программисты именно так и поступают и пользователи из-за этого теряются в догадках, почему после нажатия на кнопку от веб-приложения нет

никакой реакции).

Подведем итог: клиентская часть веб-приложений состоит из html, css, изображений и javascript-а; и для Rails-приложений свойственен т.н. тонкий клиент; вы, скорее всего, сможете обойтись совсем без javascript-а. Это может быть не очень удобно пользователю и интерфейс[5] может получиться не такой продуманный, но написать Rails-приложение совсем без использования языка javascript вполне реально.

[1] Стэк — совокупность различного софта работающего вместе для обеспечения работоспособности ключевого элемента, в нашем случае — приложения Ruby On Rails.

[2] Веб-сервер — специальная программа, работающая на сервере, которая умеет обрабатывать http-запросы, приходящие из браузера.

[3] swap — специальный раздел на жестком диске, который начинает использоваться операционной системой, когда ей не хватает памяти (RAM). Естественно, это значительно тормозит систему, потому что запись и чтение данных с жесткого диску происходит гораздо медленнее.

[4] Ajax-запрос — это запрос к серверу, который браузер выполняет без перезагрузки страницы (еще его называют *асинхронным* запросом). Подробнее об этом можно будет почитать в главе

[5] Под интерфейсом в данном случае собственно и понимается вся клиентская часть приложения и речь идет о том, что без одного из ее компонентов — javascript-а — она может получиться не такой удобной в использовании.

Понятие фреймворка в программировании

До сих пор мы говорили о Rails-приложениях не уточняя, что же именно такое RubyOnRails. Чтобы это понять, нам необходимо познакомиться с двумя понятиями имеющими отношение к разработке ПО: *либы* (от англ. lib, library или библиотека) и *фреймворки* (framework).

Либы

Программисты при разработке программ часто сталкиваются с ситуациями, где код написанный некоторое время назад ими самим или другими программистами прекрасно подходит для решения текущей задачи. Решение "в лоб" в таких ситуациях — скопипастить код. В реальности, в отдельных ситуациях, это даже может быть наиболее экономически оправданное решение, но в большинстве случаев с таким подходом возникают неизбежные проблемы. Например представьте, что после того как вы скопипастили код в свою часть программы, вы внесли в него небольшие изменения и оптимизировали работу этого куска программы с памятью. В этом случае, если вы хотите, чтобы том месте, откуда вы скопипастили этот код, программа тоже смогла пользоваться преимуществами оптимизированного вами кода, вам необходимо будет внести аналогичные изменения и

туда тоже. А теперь представьте ситуацию, когда вы скопипастили этот код не в одно, а, например в 10 мест!

В этом случае, более элегантное и удобное решение — вынести этот код в отдельный *модуль*[1] и затем повторно использовать этот модуль в тех местах программы, где он необходим. Таким образом, если вам понадобиться оптимизировать этот код, вам нужно будет сделать это только один раз. Такой подход в мире Ruby On Rails называется DRY (Don't Repeat Yourself), но, естественно, программисты в любом языке стремятся делать свой код *реюзабельным* (reusable).

Если вы вынесли свой код в отдельный модуль (фактически, это обычно один или несколько файлов с исходным кодом или скомпилированная из исходного кода библиотека) и сделали так, что этот модуль достаточно автономен (то есть может работать в любой другой программе требуя минимальное количество зависимостей), то вы можете считать, что написали либу. Естественно, программист должен знать, как пользоваться вашей либой, поэтому ответственные авторы библиотек пишут понятную документацию. В мире Ruby On Rails такие либы часто могут быть *гемами* (rubygems) и их можно установить прямо из командной строки с помощью команды `gem install`. Авторы гемов в большинстве случаев выкладывают документацию и исходный код на Github, поэтому, как правило, не составляет труда разобраться в том, как пользоваться той или иной библиотекой.

Фреймворки

Однако не все гемы, строго говоря, являются либами. Некоторые гемы состоят из нескольких библиотек, написанных разными или одним программистом. Наконец, бывают очень большие гемы, решающие множество задач и состоящие из множества библиотек. Одним их таких гемов является Ruby On Rails. Этот гем использует другие гемы и либы, которые умеют "разговаривать" с базами данных, обрабатывать http-запросы, сохранять cookies и проверять вводимые пользователем данные.

Большие структуры, которые состоят из большого количества библиотек собранных вместе для решения большой задачи (в нашем случае — разработка веб-приложения) называют Фреймворками. Фреймворки в любом языке дают вам некую стандартную структуру, пользуясь которой вы можете избежать большого количества связанных с разработкой проблем и перейти к решению непосредственно вашей задачи. Думайте о фреймворках как о скелетах, которые создают основу и частично определяют внешний вид. Любой врач, когда имеет дело с пациентом, в праве ожидать одной и той же структуры скелета и это сильно облегчает работу. Если бы строение скелета у всех людей было разное, я не думаю, что медицина могла на сегодняшний день достичь тех же успехов. Так и в программировании: если вы разрабатываете проект на фреймворке, который известен

другим программистам, то будет достаточно легко подключить их к проекту, когда вам понадобится больше разработчиков.

Или, если провести еще одну аналогию, фреймворки — это примерно как одна модель автомобиля: выглядит почти одинаково, но внутренности и внешние детали могут немного отличаться по желанию покупателя. При этом задачи, которые решает данная модель автомобиля обычно у всех покупателей достаточно похожи — Камаз приобретают не для того, чтобы ездить в супермаркет за покупками. Также и с фреймворками — они позволяют вам экономить время на разработку (представьте, как неэффективно и дорого было бы производить каждый новый грузовой автомобиль, если бы не было серийного производства) и решают ряд задач, которые являются общими для того типа проблемы, над которым вы работаете.

По началу, вы можете и не подозревать о ряде проблем, которые решает за вас фреймворк, но это не значит, что их не существует. Например, вы можете не знать, что такое CSRF-token и зачем он нужен — но использование RubyOnRails автоматически защищает вас от проблем вызванных его отсутствием. В этом смысле, фреймворки как бы являются способом аккумуляции опыта программистов, которые собаку съели на конкретной проблеме, для которой создан фреймворк.

Естественно, Ruby On Rails далеко не единственный фреймворк для разработки веб-приложений. Как минимум, я могу назвать еще Django (на языке Python), .NET (фреймворк от Microsoft) и Sinatra (фреймворк для простых веб-приложений на Ruby). Ruby On Rails, однако, является на данный момент самым активным с точки зрения сообщества — пожалуй ни в одной другой среде разработки коммьюнити программистов не является таким открытым, постоянно развивающимся и относительно профессиональным. Показательно, что ряд фреймворков на PHP и даже .NET позаимствовали несколько удачных идей из Ruby On Rails.

Хотелось бы отметить, что фреймворки — очень широкое понятие. Помимо того, что большое количество фреймворков для разных задач написано на разных языках программирования (фреймворки для веб-приложений — это лишь небольшая их часть), в различных областях человеческой деятельности активно применяют тот же самый инструмент. Например, конституция страны является по сути юридическим фреймворком в рамках которого работают все граждане и организации внутри государства (успешен этот фреймворк или нет — другой вопрос).

[1] Под модулем я здесь понимаю не module в языке Ruby, а некую абстрактную концепцию. В разных языках программирования повторное использование кода реализуется различными механизмами.

Одностраничные веб-приложения и почему Rails не подходит для их создания

Ранее я уже упоминал одностраничные веб-приложения с т.н. толстым клиентом. В таких приложениях большое количество "логики" программы содержится именно в клиентской части и написана она на языке Javascript[1] (либо на языке, который в него компилируется, например Coffeescript или Dart). У одностраничных веб-приложений все равно есть серверная часть, но есть ряд причин, по которым Rails в меньшей степени подходит на эту роль.

Во-первых, фреймворк Ruby On Rails приспособлен для того, чтобы *рендерить*[2] много маленьких html-страниц в ответ на http-запрос-ы из браузера. Одностраничному веб-приложению, как правило, это бывает очень неудобно. Оно само знает каким образом следует демонстрировать данные пользователю, ему интересны непосредственно данные. Передать просто данные (например в формате json) из Rails безусловно можно, но это скорее исключение, чем правило, для Rails-приложений — и делать это несколько менее удобно.

Во-вторых, в RubyOnRails используется концепция REST (подробнее об этом в главе Контроллеры и ресурсы). Она подразумевает, что передаваемые в запросе данные имеют отношение к одному и тому же ресурсу (например данные о "товаре"). Одностраничные приложения как правило должны уметь синхронизировать за один запрос большое количество данных, которые не обязательно связаны между собой. Скажем, пользователь находясь в самолете без доступа в интернет использует одностраничное веб-приложение "текстовый редактор" и вносит изменения в свой документ, а также в настройки своего аккаунта. Это два ресурса ("документ" и "аккаунт"), которые было бы желательно синхронизировать в один запрос, как только появится соединение с сетью. В Rails это сделать возможно, но опять-таки несколько менее удобно.

Наконец, с одностраничными веб-приложениями гораздо более остро стоит вопрос дублирования функционала. Представьте, что вам нужно *свалидировать*[3] данные, вводимые пользователем в форму. В одностраничном веб-приложении, которое обычно должно работать без соединения с сетью, вам придется реализовывать функционал валидирования данных как в серверной части приложения, так и в клиентской. Причем функционал этот — не копипаст, потому что в случае серверной части вы используете Rails и язык программирования Ruby, а в случае клиентской части — Javascript.

Rails, тем не менее, успешно используется сегодня вместе с толстыми клиентами, в частности существует довольно много материала о том, как использовать Rails вместе с фреймворком для клиентской части Backbone.js. Я отношусь к этому достаточно

скептически и считаю, что идеальным вариантом для одностраничных приложений было бы использование одного и того же фреймворка написанного на Javascript в клиентской и серверной части. Таким образом, чтобы, например, клиент и сервер использовали одинаковые *модели*[4] — это позволило бы резко сократить количество дублируемого функционала и сильно упростило бы разработку. К сожалению, на данный момент такого фреймворка не существует и разработка одностраничных веб-приложений продолжает оставаться довольно нетривиальным делом.

[1] Javascript — единственный язык программирования, который на сегодняшний день понимают практически все браузеры.

[2] Рендерить (to render) — в терминологии RubyOnRails означает генерировать html-страницы с данными, которые вставляются в эту страницу динамически из переменных Ruby. Как правило эти данные в вашем Rails-приложении вы получаете из БД, записываете их в переменную и затем используете эту переменную в html-шаблоне.

[3] Свалидировать (to validate) — убедиться в корректности, проверить на ошибки.

[4] модель из паттерна MVC, о котором речь пойдет в главе Преждем чем начать: знания, которые вам понадобятся

Преимущества и недостатки Rails

Любой фреймворк — в силу того, что он приспоблен решать определенный тип задач — имеет свои недостатки. Подобно тому, как на Камазе не очень удобно и выгодно ездить за покупками в супермаркет, Rails неудобно использовать в определенных случаях и удобно использовать в других.

В предыдущем разделе мы уже рассмотрели один вариант, когда Rails не вполне подходит под задачу — одностраничные веб-приложения. Сейчас вкратце рассмотрим другие варианты, где Rails проигрывает:

- **Многопользовательские веб-приложения, с постоянным соединением для каждого пользователя,** например онлайн игры. Как вы помните, на сервере со стэком Rails запущено несколько экземпляров приложения. Каждый экземпляр умеет обрабатывать одновременно только один http-запрос, и, кроме того, чтобы ответить пользователю, сам этот экземпляр нашей программы вынужден ждать ответа от БД. Это не самая удачная архитектура для описанной выше задачи и, возможно, тут больше подойдет Node.js и один из фреймворков для этой технологии, т.к. основное преимущество Node.js как раз и состоит в возможности обрабатывать большое количество параллельных запросов.
- **Очень простые веб-приложения состоящие из 2-3 страниц.** Создавая такие приложения нужно понимать, что, возможно, использование Rails в этом случае — это попытка стрелять в мух из пушки, предназначенной для жестокого подавления восстания слонов. В этом случае, я бы посмотрел в сторону Sinatra

или чего-то еще проще, например StaticMatic, который позволяет генерировать статичные сайты, но при генерировании статичного сайта у вас есть возможность использовать всю мощь Ruby и Ruby-гемов (например scss и coffee).

К преимуществам Rails можно отнести следующее:

- Решение большого количества проблем, с которыми сталкивается разработчик веб-приложения: cookies и сессии, sql-injections, кодировки, часовые пояса, аутентификация (при помощи гемов Authlogic или Devise), race-conditions, проверка данных вводимых пользователем и многое другое.
- Огромное количество качественной документации
- Очень активное коммьюнити
- Постоянно развивающаяся база кода, активная разработка фреймворка
- Большое количество новых удобных технологий и надстроек, которые позволяют сделать веб-разработку проще и удобнее и которые работают с Rails по умолчанию (например Bootstrap)

В этой главе мы обсудили как устроены веб-приложения и рассмотрели различные их типы. Это необходимая информация, но вам необязательно помнить все, что вы прочитали. Понимание того, что я описал в этой главе придет со временем и опытом разработки приложений на Rails; я надеюсь, что общая картина и абстрактные понятия — часть из которых, я думаю, вы все же усвоили и поняли — помогут скорее разобраться в практической части.

Глава 2. Инструменты, которые вам нужны, чтобы начать работу

Если вы хотите эффективно писать программы на определенном языке используя определенный фреймворк, первым делом вам следует поинтересоваться из чего состоит *среда разработки*. Под средой разработки обычно понимают совокупность всего ПО, которое необходимо программисту, чтобы написать программу. Можно думать об этом как о *стэке*, но только на компьютере разработчика. В этой главе мы как раз и посмотрим на среду разработки Ruby On Rails. Стоит учитывать, конечно, что у программистов бывают разные предпочтения и они любят конфигурировать свои среды по разному — используют разные ОС и софт для этих ОС, и даже настройки для этого софта могут отличаться. Я постараюсь дать вам общую картину, но, конечно же, у меня не получится избежать ситуации, когда я буду рассказывать о предпочитаемой мной среде разработки немного больше.

Почему вам следует отказаться от Windows

Строго говоря, в современном мире существуют две основные операционные системы — UNIX и Windows. Все другие названия операционных систем, которые вы могли слышать — это производные операционные системы построенные вокруг одной из этих идеологий. Например Ubuntu, MacOSX, Android и даже iOS — все это UNIX. Несмотря на то, что интерпретатор языка Ruby вполне успешно запускается под Windows, UNIX-системы предоставляют ряд преимуществ при разработке программ на Ruby (и, соответственно на Rails). Можно сказать, так получилось исторически, что Ruby-программисты по большей части работают в UNIX-системах. И, естественно, поскольку подавляющее большинство Ruby-программистов используют UNIX, эти же самые программисты придумали много разных утилит и инструментов, чтобы облегчить себе и другим жизнь. Естественно, о Windows они думали в последнюю очередь и эта ОС продолжает быть немного обделенной.

Тем не менее, я готов признать, что разрабатывать Ruby-программы под Windows — затея вполне осуществимая. Некоторое время назад даже появился инсталлятор, который автоматически ставит вам в Windows весь необходимый Rails-стэк — InstantRails. Тогда почему же я продолжаю настаивать на том, что для Rails-разработки вам следует поменять

ОС?

- Во-первых, даже если вы успешно сможете разрабатывать Rails-приложение в Windows, вам все равно придется иметь дело с сервером, на котором, конечно же, установлена та или иная версия Unix (скорее всего какой-нибудь дистрибутив Linux). Опыт использования командной строки и хотя бы небольшое знание *bash*[1] окажутся незаменимы. Это знание можно получить и уметь эффективно применять, только если вы постоянно используете терминал.
- Во-вторых, ряд руби-гемов, которые вам могут понадобиться, со скрипом устанавливаются в Windows (это может измениться в будущем, если авторы гемов найдут время и желание сделать их совместимыми с Windows).
- Кроме того, как я уже упоминал выше, Ruby-коммьюнити в основном работает на UNIX-системах, а это означает, что при возникновении проблемы (а они будут возникать), шанс найти решение, которое бы вам подошло, у вас резко снижается.
- Наконец, Ruby-интерпретатор работает под Windows гораздо медленнее. На данный момент, активно развиваются альтернативы, например IronRuby.

Поэтому мой совет такой: оцените все ЗА и ПРОТИВ. Отказываясь от Windows (хотя бы на время работы), вы, безусловно, отказываетесь от привычного вам ПО, которое установлено у вас в системе. Это, на мой взгляд, один из самых серьезных барьеров для людей при переходе на другую систему. С другой стороны, если вы действительно настроены серьезно и хотите научиться программировать на Ruby — это закономерный шаг. Мой личный опыт таков: я не сразу "переехал" на Linux, это было относительно постепенно и сопровождалось приступами ярости. Со временем, тем не менее, я осознал, что все эти различия между системами не так уж велики и я готов пожертвовать удобствами в развлечениях (игры? любимый IM-клиент? удобный музыкальный плеер?) ради удобства в работе.

Большинство Linux-дистрибутивов позволяют вам установить на ваш компьютер две системы одновременно и выбирать что загружать — Windows или Linux — когда вы включаете компьютер. Но, если по началу даже это кажется слишком страшным, попробуйте установить Linux на виртуальную машину VirtualBox (софт бесплатный) и вы сможете запускать Linux прямо из Windows!

Наконец, если я все еще не убедил вас — попробуйте почитать это обсуждение ограничений при Windows для запуска Ruby и Rails, на странице есть ряд полезных советов, в том числе касающихся среды разработки Rails в Windows.

Ах да, совсем забыл. В этой книге далее будет подразумеваться, что вы используете UNIX-систему.

[1] О том, что такое bash мы поговорим в разделе Зачем и как пользоваться терминалом

Выбор операционной системы

И так, вы решили отказаться от Windows в пользу UNIX-системы. Возникает вопрос, какую именно выбрать? По сути, выбор сводится к двум вариантам: Ubuntu Linux или MacOSX. Здесь я не буду рассматривать другие дистрибутивы Linux (потому что, по большому счету, всмысле установки Rails-стэка они похожи на Ubuntu) и такие ОС как FreeBSD (потому что энтузиасты смогут разобраться самостоятельно, а мой опыт с "фряхой" практически нулевой). Как бы там ни было, эти два варианта являются на сегодняшний день самыми популярными.

Ubuntu Linux

Эта операционная система является, на мой взгляд, идеальным выбором для начинающего Rails-программиста. Во-первых она не требует никаких материальных вложений — дистрибутивы Linux это свободно-распространяемое ПО, систему можно скачать с официального сайта. Во-вторых, ее можно поставить в качестве второй ОС на тот же самый компьютер — установщик сделает все за вас. Более того, вам даже необязательно переразбивать диск во время установки[1], вы можете поставить Ubuntu на виртуальный диск, коим будет являться специальная папка, местоположение которой вам дадут выбрать. Кроме всего прочего, Ubuntu прекрасно устанавливается на виртуальную машину VirtualBox (также бесплатное ПО) и ее можно запускать прямо из Windows.

У десктопной версии Ubuntu современный интерфейс и большое количество бесплатного ПО, которое доступно через "магазин", поэтому вы всегда сможете найти замену своим любимым программам из Windows. Ну, и конечно, вы автоматически получаете преимущество — устанавливая десктопную версию системы вам будет гораздо проще настраивать серверную версию позже, когда вы будете выкладывать свое приложение в сеть.

MacOSX

Эта система долгое время была и продолжает оставаться одной из самых популярных платформ для разработки приложений на Rails. Из потенциальных отрицательных моментов можно назвать необходимые дополнительные денежные вложения в покупку компьютера Apple, меньше возможностей для кастомизации системы, невозможность установки на

VirtualBox и некоторые препятствия на пути к установке среды разработки (впрочем, препятствия будут вам встречаться всегда и в любой ОС). В целом, с моей точки зрения, MacOSX ничем не лучше и не хуже Ubuntu в плане разработки Rails-приложений. Я работаю в обоих системах одинаково продуктивно.

[1] Во время установки вам будет предложено создать новые разделы на диске. Если вы сомневаетесь как правильно это сделать, то чтобы случайно не потерять данные — установите Ubuntu на "виртуальный диск" прямо из Windows.

Что должен уметь текстовый редактор

Текстовый редактор — программа в которой вы будете проводить большую часть своего времени как программист, поэтому к его выбору следует отнестись со всей серьезностью. Возможно вам стоит попробовать несколько редакторов, пока вы не остановитесь на том, который посчитаете наиболее подходящим вам. Но, какой бы редактор вы ни выбрали, вам обязательно нужно потратить некоторое время и основательно разобраться в его функциях. Я до сих пор продолжаю разбираться в своем текстовом редакторе (Vim) — хотя пользуюсь им уже лет 5 — и примерно раз в месяц посвящаю немного времени тому, чтобы узнать пару новых трюков или попробовать новый интересный плагин. Причина проста — грамотно настроенный и хорошо знакомый вам текстовый редактор повышает вашу продуктивность и удовольствие от работы. Чего не стоит делать, так это тратить, например, всю свою рабочую неделю на настройку редактора перед тем, как приступить непосредственно к написанию кода. Обновление настроек редактора и его изучение должно происходить постепенно и быть следствием ваших задач.

Ниже я перечислю несколько важных функций, которые должны обязательно присутствовать в вашем текстовом редакторе, если вы разрабатываете Rails-приложение:

- **Подсветка кода** — самая базовая функция текстового редактора для программистов. Естественно, редактор должен поддерживать не только Ruby, но также Javascript, html, erb, css и желательно haml, scss и Coffescript. Обычно, в текстовых редакторах предусматривают возможность скачивать специальные расширения, чтобы редактор научился понимать тот или иной язык и правильно подсвечивать код, написанный на этом языке.
- **Замена табуляции на пробелы** — функция совершенно необходимая, если вы пишите код на Ruby. Если вам уже приходилось писать какой-нибудь код на любом языке, вы, я думаю, быстро оценили необходимость форматировать код символвами табуляции (многие текстовые редакторы делают это за вас). Посмотрите на отформатированный табами кусок кода:

```
if current_user.admin?
  redirect_to("/admin/posts")
else
  redirect_to @post
end
```

В этом примере в настройках редактора указано, что табы визуально заменяются на 2 пробела. Если вы пошлете этот код своему коллеге-программисту у которого в настройках редактора табы представляются не как 2, а как 4 пробела, то картина будет следующей:

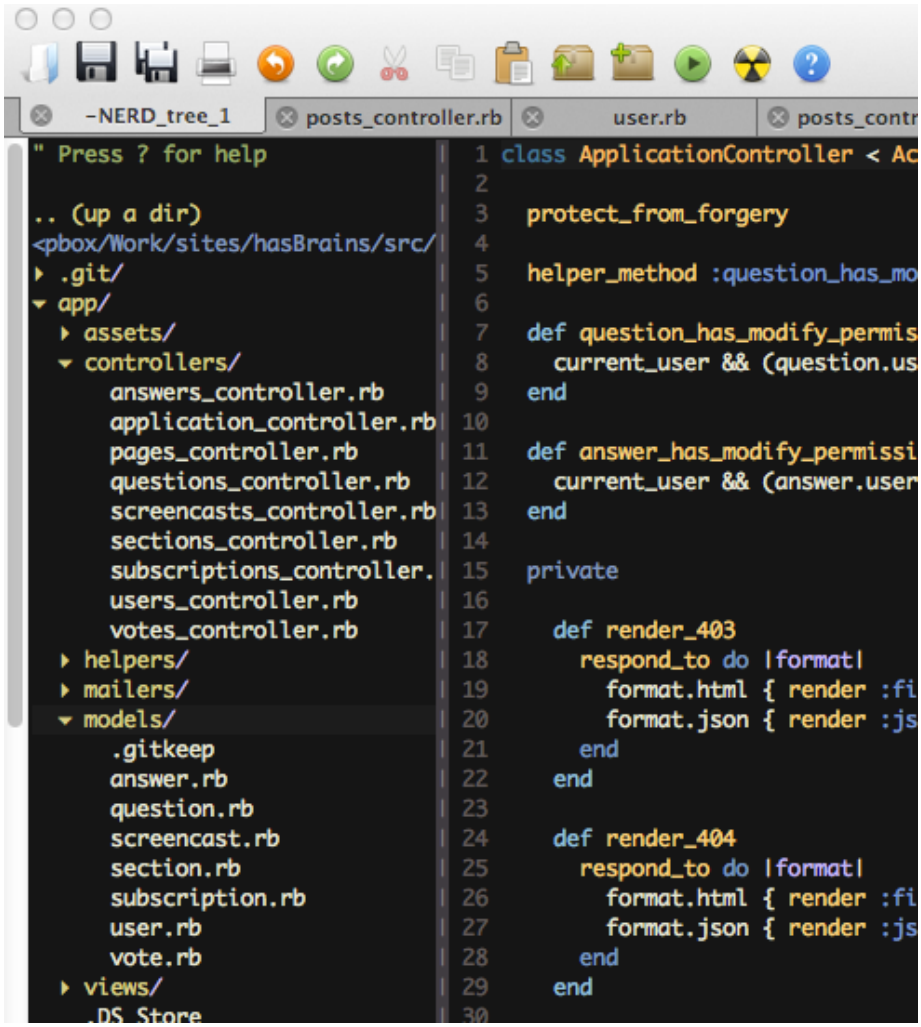
```
if current_user.admin?
    redirect_to("/admin/newspaper_post_orders")
else
    redirect_to @post
end
```

На первый взгляд может показаться, что в этом нет ничего страшного и что каждый программист должен иметь право смотреть на код с теми настройками, которые больше нравятся ему. Однако, форматирование кода[1] является неотъемлемой частью самого кода и влияет на его восприятие. Написать красивый и читабельный код также важно, как написать код, который работает. В случае с нашим небольшим примером, возможно, не очень легко оценить необходимость одинаково выглядящего кода, но, поверьте: когда перед вами на экране большой файл, начинаешь совсем иначе думать о том, как бы сделать так, чтобы он выглядел аккуратно. И если вы, в какой-то момент, выложите свой код на Github, никто и читать его не станет, если он отформатирован не так, как следует.

Поэтому в Ruby-сообществе договорились, что все символы табуляции в коде должны *жестко* заменяться на 2 пробела[2]. То есть, замена должна быть не только визуальной — текстовый редактор должен автоматически — при нажатии вами клавиши `Tab` — подставлять 2 пробела. Найдите соответствующую настройку в вашем редакторе. Если возможность настроить это поведение отсутствует — выбирайте другой редактор.

- **Номер строк слева от кода.** Это крайне важная настройка, отлаживать программы без которой практически невозможно. Любой современный компилятор или интерпретатор языка при возникновении ошибки показывает имя файла и номер строки, в которой эта ошибка произошла. Кроме того, кога вы будете запускать тесты с помощью rspec, тесты, которые "не прошли" тоже будут сообщать вам номер строки, на которой условие не выполнилось. Да и при работе в команде с другими программистами, быстро сообщить коллеге номер строки, на которую нужно обратить внимание, тоже бывает абсолютно необходимо.

- **Навигация по директориям** — чтобы, не выходя из текстового редактора и не открывая дополнительных диалоговых окон, можно было просматривать список файлов и папок и открывать их. Обычно, самый удобный способ, когда это реализовано в виде панели сбоку. В приведенной иллюстрации показан пример того, как это выглядит в моем редакторе:



- **Удобная навигация по тексту без мыши** — важная составляющая вашей продуктивности. Чем меньше вы будете отрывать руки от клавиатуры, тем быстрее будете писать код. Любой хороший текстовый редактор имеет развитый функционал для этого. Как правило это реализуется с помощью сочетания клавиш или специальных команд, которые надо вводить в редактор.

Ну и, разумеется, будет уместным перечислить тут популярные на сегодняшний день текстовые редакторы для программистов:

- Vim (платформы: Windows, Linux, MacOSX и др., бесплатный)
- Emacs (платформы: Windows, Linux, MacOSX и др., бесплатный)
- Sublime (платформы: Windows, Linux, MacOSX, платный)
- Textmate (платформы: MacOSX, платный)
- gEdit (платформы: Linux, ,бесплатный)

[1] Лично я предпочитаю следовать правилу *1 Таб заменяется на 2 пробела* не только с файлами содержащими Ruby-код, но и с любым другим языком программирования.

[2] Под форматированием кода понимается в основном выравнивание с помощью пробелов.

Зачем и как пользоваться терминалом

Я из тех, кому нравится говорить "люди делятся на два типа...". Пусть это не всегда точно, но мне кажется это первый шаг к тому, чтобы лучше понять человека. Так вот, люди делятся на два типа: те кому нравится черный экран и *командная строка*[1] и те, кто боятся ее и пытаются избегать необходимости иметь с этим дело. Или я ошибаюсь? Чтобы это понять, давайте разберемся что такое командная строка и зачем ее придумали?

Что такое командная строка?

Начнем с примера. Скажем, я написал на Ruby небольшой метод, который считает среднее арифметическое:

```
def mean(*numbers)
  sum = numbers.inject { |sum, i| i+sum }
  sum.to_f / numbers.size
end
```

Теперь я хочу сделать из этого настоящую программу и начать продавать ее пользователям. Однако пока что у меня нет для нее интерфейса — то есть пользователи никак не могут взаимодействовать с этой программой. И тут, по большому, счету у меня есть два пути: сделать GUI (графический интерфейс) или сделать мою программой утилитой для командной строки.

В первом случае, я нарисую форму, состоящую из нескольких полей и кнопки "рассчитать" и затем каким-либо образом привяжу событие "нажатие кнопки" к вызову моего метода, передав в этот метод все, что пользователь ввел в поля формы. Затем я получу то, что вернет мне метод и выведу на экран пользователю.

Во втором случае, я немного модифицирую свой код следующим образом:

```
#!/usr/bin/ruby
def mean(*numbers)
  sum = numbers.inject { |sum, i| i+sum }
  sum.to_f / numbers.size
end
puts mean(*ARGV.map())
```

Затем я сохраню этот код в файле с названием `calc_mean` и сделаю файл исполняемым. Добавив две строки в свой код я, фактически, сделал из своей программы утилиту для командной строки. Теперь я могу открыть терминал, перейти в директорию где лежит этот файл и набрать команду `./calc_mean 1 2 3` после чего я получу в этот же терминал сообщение `2.0`

На этом примере прекрасно видно, что по сути и в одном и в другом случае метод `#mean` никак не меняется, а GUI и команданая строка являются просто средствами доступа к этому методу. Иначе говоря, и GUI и командная строка являются интерфейсами к одной и той же программе.

Теперь несложно догадаться, что когда компьютеры были еще не очень мощными и умными, проектировать и запускать графический интерфейс было дорогостоящим и неэффективным занятием, особенно для небольших задач. Поэтому в UNIX-системах начало появляться большое количество т.н. утилит для командной строки. Таким образом, командная строка вовсе не является просто предпочтением — это насущная необходимость.

Но почему командная строка до сих пор в моде? На то есть несколько причин. Во-первых, часто бывает быстрее и удобнее сделать ряд действий именно в командной строке, а не через графический интерфейс. Например, вот команда, которая прочтет последние 50 строк лог-файла, а затем выведет только те, которые начинаются на "error":

```
tail development.log -n50 | grep /^error/
```

Но удобство — не единственная причина того, что командная строка до сих пор активно используется программистами. В конце концов всегда можно придумать достаточно удобный GUI выполняющий ту же задачу (другое дело, что это может быть экономически не оправдано). Командная строка активно используется при настройке и администрации серверов, работающих на UNIX-системах — на сервере ведь нет десктопа и окошек. Хотя, безусловно, в последнее время стали появляться удобные GUI-веб-интерфейсы для настроек сервера, командная строка остается и будет оставаться незаменимым инструментом.

Большое количество команд используется для взаимодействия программиста с Rails-приложением, например `rails s` запускает сервер с приложением на порту 3000, после чего вы можете увидеть ваше веб-приложение в браузере набрав в адресной строке `http://localhost:3000`. А `rails c` запускает в терминале специальную Rails-консоль, которая позволяет управлять вашим приложением прямо из терминала (мы не раз вернемся

к этому в следующих главах).

Так что, в реальности, я бы сказал, что люди делятся на следующие два типа: те, кто понимает, что командная строка — это просто еще один интерфейс и те, кому еще предстоит это осознать. В конце концов, у Photoshop-а тоже есть графический интерфейс, но ведь это вовсе не означает, что научиться работать с ним — легко. Разбираться придется в любом интерфейсе в любом случае и командная строка едва ли сложнее, чем GUI.

Как использовать командную строку?

В вашей операционной системе (и, я надеюсь, что это не Windows!) есть приложение, которое так и называется "Terminal". Открыв его, вы увидите перед собой окно с черным фоном (или белым фоном — это настраивается и не имеет значения) и *приглашением* (т.н. command promt). Приглашение может отличаться в зависимости от настроек вашего терминала, но примерно это может выглядеть вот так:

```
~ $
```

Символ ~ означает что вы находитесь в "домашней папке" пользователя (`/home/username` в Ubuntu или `/Users/username` в MacOSX). Вместо него там может быть путь к папке, в которой вы находитесь в данный момент, например `~/Desktop`. Символ $ является как бы разделителем после которого вы можете печать свои команды. После того, как команда напечатана, необходимо нажать `Enter` и тогда она выполнится. Вот так и работает терминал: вы вводите команды, нажимаете `Enter`, получаете какой-нибудь ответ от запущенной вами команды. Попробуйте набрать команду `ls` и вы увидите содержимое текущей папки.

Кроме того, каждой команде можно передавать *флаги* и *аргументы*. Флаги — это специальные опции влияющие на поведение команды, они указываются после символов `-` или `--`. Например следующая команда

```
ls -al
```

выведет содержимое папки включая скрытые файлы (флаг `-a`) и покажет содержимое в виде подробной таблицы (флаг `-l`). Обратите внимание, что флаги "слиплись", то есть указаны подряд после одного дефиса.

Аргументы — это все, что идет после названия команды и не является флагом. Например, мы можем передать команде `ls` имя директории в качестве аргумента и она выведет нам содержимое указанной директории, а не текущей, как было бы в случае, если бы мы не передавали никаких аргументов:

```
ls -al ~/Desktop
```

Естественно, каждая команда ведет себя по своему и надо сверяться с документацией и смотреть какие флаги и аргументы она принимает и что они означают.

В UNIX существует огромное количество утилит для командной строки. Никто не умеет эффективно пользоваться всеми, да и помнить наизусть все утилиты вряд ли возможно. Скорее, у каждого есть набор команд, которыми он пользуется чаще всего и которые составляют львиную долю того, что он вбивает в терминал. Ниже я приведу список самых распространных команд, которых вам должно хватить на первое время использования терминала.

`cd`	переход в другую папку
`ls`	показать содержимое папки
`touch`	создать пустой файл
`mkdir`	создать папку
`rm`	Удалить файл или папку. Внимание: будьте крайне осторожны с формой этой команды `rm -rf`. Не набирайте это в терминале, если вы абсолютно точно не уверены, что это означает — вы можете случайно удалить важные файлы.
`cat`	вывести содержимое файла
`tail`	вывести последние n строк из файла
`man`	получить справку о команде

Поэкспериментируйте с каждой из команд, но перед этим обязательно почитайте справку. Сделать это можно набрав последнюю упомянутую команду `man` передав ей в качестве аргумента имя интересующей вас команды, например:

```
man ls
```

Выдаст справку по команде `ls`.

В дальнейшем мы будем активно использовать терминал для работы над Rails-приложением, поэтому я советую вам начать постепенно осваивать его. Мне в свое время очень помогла книга Learning the bash Shell: Unix Shell Programming. Я думаю, сейчас в моей памяти осталось примерно треть материала из этой книги, но ценность потраченного на ее прочтение времени состоит в том, что теперь я знаю *что возможно сделать* используя терминал и знаю, *где можно посмотреть как это сделать.*

[1] Далее я буду употреблять понятия *терминал* и *командная строка* как синонимы. На самом деле, небольшая разница между двумя терминами есть и помимо них часто встречаются такие термины как shell, консоль и tty. Подробнее о различиях можно почитать в многочисленных топиках на эту тему.

[1] *Learning the bash Shell: Unix Shell Programming*

[2] *CommandLine Fu - сайт-сборник команд для терминала с рейтинговой системой, пополняется пользователями.*

[3] *Superuser.com - сайт, являющийся частью сети StackExchange посвященный вопросам управления системой для продвинутых пользователей, в том числе содержащий много вопросов по использованию командной строки.*

[4] *unix.stackexchange.com - сайт также, являющийся частью сети StackExchange и посвященный Unix-системам. Поскольку использование командной строки распространено именно среди пользователей Unix-систем, то там встречаются вопросы и на эту тему тоже.*

[5] *8 Linux Commands Every Developer Should Know - любопытная статья с объяснением нескольких нетривиальных команд и примерами их использования.*

Системы контроля версий

Зачем они нужны?

Простой пример: вы внесли изменения в код программы, а на следующий день решили, что изменения неверные и нужно сделать все как прежде. Текстовый редактор закрыт, отменить действия не получится. Как быть? Для этой и других ситуаций была придумана отдельная категория ПО, которая называется *VCS — Version Control Systems*.

Идея простая: вы говорите VCS за какой папкой необходимо следить (в какой паке создавать репозиторий) и затем, в определенные моменты времени, сохраняете текущее состояние папки — это действие называется "сделать коммит" или "закоммитить". Таким образом, если вам когда-либо понадобится вернуться к предыдущему состоянию, вы сможете посмотреть историю предыдущих коммитов, выбрать тот, который вас интересует и восстановить состояние репозитория на момент времени, когда был сделан этот коммит.

В целом, системы контроля версий выполняют множество задач, самые важные из которых сводятся к следующим:

- Контроль за изменениями в проекте и возможность "откатиться" к состоянию в прошлом
- Параллельная работа нескольких человек в команде, разрешение конфликтов при внесении изменений в один и тот же файл
- Параллельная работа над несколькими вариантами одной и той же программы
- Синхронизация изменений и рапспространение исходного кода программы

Очень простой пример использования

Чтобы лучше понять, как это на самом деле работает, я предлагаю воспользоваться самой популярной в Ruby On Rails сообществе системой контроля версий, которая называется `Git` и рассмотреть пару примеров. `Git` — это по сути набор утилит, которые работают из командной строки (то есть нам понадобится терминал). Для начала убедитесь, что Git установлен в вашей системе: выполните в терминале команду `git -v`. Если терминал ответил, что такой команды не знает, то вам нужно поставить Git в систему. В MacOSX это делается командой `brew install git`[1] а в Ubuntu Linux командой `sudo apt-get install git-core`.

И так, мы готовы на примере посмотреть, как работает Git. Создайте новую папку и перейдите в нее в терминале (команда `cd`). Откройте текстовый редактор, создайте файл в этой папке файл `hello.rb`и добавьте в него следующий ruby-код:

```
puts "hello world"
```

Теперь мы готовы создать репозиторий в этой папке[2] — то есть, очень грубо говоря, объяснить Git-у, что мы должны следить за этой папкой. Это делается командой `git init` (после ее выполнения в вашей папке появится скрытая папка `.git` — никогда не изменяйте файлы внутри этой папки, только Git знает как это делать). Создав новый репозиторий в текущей папке мы можем "делать коммиты". Поскольку у нас уже есть первый файл (то есть мы внесли изменения в содержимое папки), мы можем сделать и первый коммит. Это делается несколькими командами. Сначала необходимо объяснить Git-у, что мы хотим включить добавленный файл в репозиторий (по умолчанию, git игнорирует все новые файлы в текущей папке):

```
git add .
```

Это действие называется "добавить в индекс", а точка в качестве аргумента означает "все файлы" — мы могли бы вместо точки написать имя нашего файла, но в данном случае это не имеет большого значения. Второй шаг, это сделать, собственно, коммит:

```
git commit -a
```

Флаг `-a` означает "автоматически коммитить все файлы в которые были внесены изменения" (опять-таки, по умолчанию git коммитит только новые файлы либо те, которые были указаны в качестве аргументов к этой команде).

После выполнения последней команды, у вас автоматически должен открыться дефолтный для системы текстовый редактор — у меня это Vim и он открывается прямо в терминале. В первой строке, то есть на самом верху, вам необходимо ввести описание производимого коммита. Описание — очень важная вещь, т.к. именно по нему вы (и программисты из вашей команды, если вы работаете не один) в дальнейшем сможете найти

нужный вам коммит. Описание должно четко и коротко отражать, что именно было сделано в этом коммите. Я предпочитаю, чтобы оно было на английском, но можно писать его и на русском — в том случае, если вы уверены, что его будут читать ТОЛЬКО люди, говорящие на русском. Вот как могло бы выглядеть описание для нашего первого коммита:

```
Added simple 'puts' statement that shows standard greeting in
the terminal
```

Только после того, как вы сохраните изменения и выйдете из текстового редактора — коммит будет добавлен.

Теперь внесем небольшие изменения в нашу программу, изменим код, чтобы он выглядел следующим образом:

```
puts "hello #{ARGV[0]}"
```

и снова закоммитим изменения с помощью команды `git commit -a`. В этот раз, описание коммита могло бы выглядеть следующим образом:

```
Greeting is now customized, user can type in the second word as
an argument.
```

После того, как вы сохраните коммит, вы можете посмотреть на историю коммитов, которые мы уже сделали (их будет всего 2) с помощью команды `git log`. В самом верху описания каждого коммита будет примерно такая строка:

```
commit 2334022ec6e417d58201f7802885f6b879d62d9a
```

То, что идет после слова "commit" является уникальным идентификатором коммита. Сейчас мы воспользуемся им, чтобы "откатиться" к нашему первому коммиту. Найдите этот коммит в списке (он должен идти последним), выделите первые 8 или 10 символов этой длинной строки (без слова commit, разумеется) и скопируйте их. Затем, введите в терминале следующую команду:

```
git checkout 2334022ec6
```

Естественно, id вашего коммита будет отличаться от приведенного тут. И, причина по которой мы скопировали только 10 первых символов заключается в том, что в большинстве случаев Git-у их вполне достаточно, чтобы идентифицировать коммит.

После выполнения этой команды, Git откатит вас к состоянию вашего первого коммита и если вы откроете файл `hello.rb`, вы увидите там следующий код:

```
puts "hello world"
```

Чтобы вернуться к самому последнему (свежему) коммиту, наберите команду `git checkout master`

Что потребуется знать, чтобы работать с Git-ом?

Тот пример, который я привел выше демонстрирует только часть функций, которые выполняет Git. Вам потребуется знать немного больше, чтобы эффективно использовать Git и работать с Github-ом. В конце раздела я привел ссылки на бесплатные англоязычные и русскоязычные материалы по Git-у (в основном, написанные понятным для новичка языком), а ниже я перечислю список основных команд, которые вам следует знать для ежедневной работы с Git-ом:

`git init`	Создание нового репозитория
`git add`	Добавление файлов в индекс
`git commit`	Создание нового коммита
`git push`	Залить все новые коммиты в удаленную копию репозитория
`git pull`	Скачать новые коммиты из удаленной копии
`git checkout`	Перейти к конкретному коммиту или ветке
`git branch`	Показать все ветки в репозитории
`git log`	Показать историю коммитов в текущей ветке
`git diff`	Показать различия между двумя коммитами

Чтобы получить подробную информацию по каждой команде, вы можете набрать в терминале

```
git help [имя команды]
```

или найти соответствующую команду в онлайн-документации [3] (содержимое онлайн-документации идентично).

Несколько слов про коммиты

Новички, которые только знакомятся с Git-ом не всегда осознают необходимость его использования, а также когда и как нужно делать коммиты. Здесь я хотел бы обратить внимание на несколько важных моментов, которые всегда нужно помнить при работе с Git-

ом.

Что коммитить?

В примере выше мы вносили очень незначительное изменение в наш файл — была изменена всего лишь одна строка. Тянет ли это на коммит? Вполне. Но естественно, в большинстве случаев ваши коммиты будут охватывать несколько строк и скорее всего несколько файлов. Коммит должен представлять из себя некую неделимую и завершенную единицу функционала программы. Например коммит, который добавляет регистрацию пользователей на сайт; или коммит, который дает возможность администратору сайта редактировать все товары в интернет-магазине; или коммит, который исправляет критическую ошибку в программе, которая не позволяла пользователям загрузить их изображение — если оно, скажем, было в формате PNG, а не в формате JPG. В этом смысле, коммиты часто условно разделяют по назначению на несколько разных типов:

- Исправление ошибок (Fix)
- Рефакторинг старого кода (Refactoring)
- Добавление нового функционала
- Oops-коммиты — исправление опечаток и других мелких вещей

Я взял в привычку добавлять соответствующие префиксы (Fix, Refactoring, Oops) перед описанием коммита, так что например описание коммита, который исправляет возможность загрузки изображения выглядит так:

```
Fix: users couldn't upload images in any format other than JPG
```

Часто так бывает, что пока вы вносите изменения касающиеся одной задачи (и, соответственно, предназначенные для одного коммита) вам приходится вносить изменения в ряд других файлов — изменения, которые имеют только косвенное отношение к текущей задаче и которые можно было бы выделить в отдельный коммит. Что делать в таких случаях? Вы всегда можете напрямую указать команде `git commit` список файлов, которые вы хотели бы закоммитить. И, таким образом, у вас есть возможность выделить в отдельный коммит косвенные изменения.

Чего не стоит делать, так это объединять изменения, которые имеют отношение к двум или более задачам в один коммит. Обычно это становится очевидным, когда вы начинаете составлять описание для коммита и понимаете, что в одном предложении соеденены две абсолютно разные задачи, например:

```
Added user authentication and fixed image uploading.
```

Понятно, что такой коммит — плохая идея.

Описание коммитов

Составление описания для коммитов — своего рода искусство. Описание каждого коммита очень важно — его будут читать живые люди (другие программисты и вы в том числе) и поэтому нужно предельно точно и коротко объяснять в описании суть коммита.

Если вам требуется больше места и описание коммита не умещается, скажем, в 80 символов, вы можете сделать следующее: придумать описание, которое уместится в 80 символов, а затем дать более развернутое объяснение для коммита. Для этого оставьте пустую строку после первой и затем начните писать развернутое описание с 3-ей строки.

Пример описаний коммитов можно посмотреть на Github-е, например в репозитории RubyOnRails. Наконец, я мог быть порекомендовать вам пробежаться по большому количеству статей, посвященных правильному написанию commit messages.

[1] Подробнее о менеджере пакетов `Homebrew` читайте в разделе Установка необходимого для запуска приложения софта

[2] Скорее всего, если вы в первый раз используете Git в вашей системе, вам потребуется сообщить ему ваш email и имя — это нужно для идентификации вас как автора коммитов. Сделать это можно двумя командами:

```
git config --global user.email "my@email.com"
git config --global user.name  "Ivan Ivanov"
```

Крайне желательно указывать ваше настоящее имя и email, т.к. их будут использовать другие программисты для связи с вами если, например, в будущем вы будете выкладывать свои руби-гемы или какой-то другой код на GitHub.

[1] Книга "Волшебство Git" на русском (бесплатно)

[2] Книга "Pro Git" на (бесплатно)

[3] Документация на официальном сайте

[4] Мануал по использованию Git для пользователей Linux

[5] Введение в Git (мануал для начинающих)

[6] Приложение GitBox для MacOSX, графический интерфейс для Git

Почему IDE - это плохо

Некоторые программисты вместо работы в терминале и текстовом редакторе предпочитают ипользовать *IDE — Integrated Development Environment*[1]. Для тех, кто не знаком с термином, вкратце объясню, что это такое: это специальная программа, которая похожа на текстовый редактор, но при этом имеет большое количество дополнительных функций, помогающих в разработке приложения и, по сути, способствует тому, чтобы

программист никогда не покидал пределы этой программы — что бы ему не нужно было сделать.

С первого взгляда, может показаться, что это отличная идея. Что может быть лучше, чем собрать в одном месте все необходимые инструменты? Однако на мой взгляд, в использовании IDE кроются два момента, которые люди ошибочно считают преимуществами:

- **Графический интерфейс (GUI) при выполнении различных задач облегчает и ускоряет работу.** Но так ли это? Сомневаюсь, что отрывая руки от клавиатуры и занимаясь поиском определенной кнопки или пункта меню в IDE вы экономите больше времени, чем вводя команду в терминал. Терминал — это просто еще один интерфейс, который, точно так же, как и IDE, нужно освоить. Подумайте об этом, когда вы в первый раз откроете незнакомую вам IDE и начнете в ней разбираться.
- **Не нужно выходить из программы, все собрано в одном месте.** Или, если говорить более точно, в одном окне. Но так ли сложно переключаться между окнами? И не засоряет ли ваш экран обильное количество кнопочек и менюшек, доступных в IDE? Можно ли это на самом деле считать преимуществом?

В целом, я вижу с IDE ряд проблем. Все они, по сути, сводятся к тому, что провоцируют вас оставаться пассивными в плане обучения. Например, функция автозаполнения (которая, к слову, есть и в некоторых текстовых редакторах) вовсе не способствует быстрому написанию кода. Подавляющее большинство примеров, которые я видел выглядят так: человек начинает писать ключевое слово и затем 2-5 секунд выбирает подходящее из выпадающего списка. Если я знаю это ключевое слово или название метода, я напишу его за секунду или меньше. Если я не знаю, какой метод мне нужен или я забыл его название, я иду в документацию и ищу там нужную мне информацию[2]. Это не только помогает подробно разобраться в том, как работает тот или иной метод, но и помогает мне запомнить его название. А это в свою очередь важно потому что вы сможете: быстрее писать свой код, правильно называть свои собственные методы и переменные, эффективнее читать чужой код.

Тот же аргумент можно применить к различным функциям IDE, которые, по сути, выполняют за вас те или иные команды в терминале. Возможно вам действительно будет проще нажать на кнопку в GUI, чем написать команду (особенно длинную), но это неизбежно приведет к тому, что вы будете меньше знать и не сможете придумывать полезные команды для терминала "на лету".

Наконец, еще одна глобальная проблема с IDE состоит в том, что они "привязывают" вас к себе. Каждый производитель IDE хочет, чтобы вы использовали только его софт, поэтому

интерфейс у различных IDE обычно отличается в достаточной степени, чтобы вызывать неудобства. Плюс ко всему, как только вы по какой-то причине окажетесь без этой IDE (например производитель перестанет поддерживать ее, а старая версия будет некорректно работать с новой версией Rails; или вам нужно будет настроить сервер!), вы окажетесь в ситуации, когда вы фактически не сможете работать. На мой взгляд очень неразумно ограничивать себя таким образом.

Безусловно, иногда получается так, что IDE — единственный способ создавать софт для определенной платформы. Например, это как раз случай с компанией Microsoft и ПО для Windows. Хорошо это или плохо — судить вам. Хорошая новость тут в том, что Microsoft будет поддерживать свои IDE до тех пор, пока существует Windows. Плохую новость прекрасно сформулировал пользователь Hacker News в одном из обсуждений на сайте: "иногда у меня создается впечатление, что Microsoft придумала все эти огромные IDE с удобным графическим интерфейсом именно потому, что сам API Windows невыносимо уродлив и запутан."

В случае UNIX-систем, API (коим являются утилиты командной строки и файлы конфигураций) относительно прост, а документация и возможность задать вопрос и получить (или нагуглить) быстрый ответ всегда доступна в открытом доступе и из множества источников. В общем, для UNIX-систем IDE — это относительно чуждое понятие.

Подводя итог и учитывая все вышесказанное, на мой взгляд IDE — вредны для новичков, потому что не способствуют реальному обучению и привязывают их к определенному интерфейсу. Если я бы и мог порекомендовать использование IDE, то только после того, как вы хорошо освоите разработку с помощью текстового редактора и командной строки. Только тогда все те негативные стороны работы IDE в большей степени могут быть сведены на нет.

[1] Список популярных IDE для Ruby и Ruby On Rails приведен в конце раздела.

[2] О том, как пользоваться документацией для Ruby, Ruby On Rails и других библиотек, рассказано в главе Что дальше: как разбираться в проблемах и кому задавать вопросы?, ее можно прочитать в любой момент, материал в ней не сильно привязан к предыдущим главам.

[1] RubyMine

[2] Aptana

[3] NetBeans

[4] Обсуждение различных IDE для разработки приложений на Ruby и RubyOnRails на StackOverflow.

Установка необходимого для запуска приложения софта

Менеджеры пакетов

Установить необходимый для разработки Ruby On Rails приложений софт в UNIX-системах можно несколькими способами. Самый древний и самый сложный — "скомпилировать из исходников", что означает скачивание исходного кода программы и использование установленного в системе компилятора для того, чтобы создать из этого кода работающую программу. К этому способу иногда до сих прибегают, когда вам требуется тонкая настройка системы и приложения. Проблемы, с ним, как правило, заключаются в необходимости самостоятельно устанавливать все *зависимости*[1], а также в обновлении или удалении этого софта в последствии (в UNIX-системах программы устанавливаются не в одну папку, как в Windows, а часто распределяют свои файлы по множеству системных папок; без информации о том, куда все эти файлы были установлены, обновление или удаление становятся довольно сложными задачами).

Наиболее удобный в большинстве случаев способ — это установка с помощью менеджера пакетов. *Менеджер пакетов* — это специальная программа, которая скачает исходный код, скомпилирует его за вас (или скачает уже скомпилированную для вашей системы версию), установит скомпилированную программу в систему и затем позволит без лишних проблем обновить или удалить ее, если в этом будет необходимость. Почти как AppStore, с тем лишь исключением, что скачиваемый софт — бесплатный и работать с менеджерами пакетов обычно нужно из командной строки. Далее мы рассмотрим два менеджера пакетов: `Homebrew` для MacOSX и `Aptitude` для Ubuntu-Linux.

Менеджер пакетов Homebrew в MacOSX

`Homebrew` не установлен по умолчанию в каждую копию MacOSX, поэтому вам придется приложить немного больше усилий чем пользователям Ubuntu. Чтобы установить `Homebrew`, вам нужно просто открыть терминал и ввести туда следующую команду:

```
ruby (curl -fsSkL raw.github.com/mxcl/homebrew/go)
```

Обратите внимание на две вещи: 1) для установки используется программа написанная на Ruby, которая предварительно скачивается с удаленного сайта и 2) мы не используем префикс `sudo`, таким образом и при установке софта через Homebrew нам не нужны будут права администратора.

После того, как вы введете указанную команду в терминал, внимательно следите за сообщениями, которые будут появляться в терминале и следуйте указаниям. Подробнее об установке можно почитать в Wiki Homebrew на Github-е и на главной странице Homebrew.

И еще один важный шаг: многие пакеты, которые будут устанавливаться через `Homebrew`, требуют компиляции, поэтому вам наобходимо иметь компиляторы в вашей системе. По умолчанию они не установлены в MacOSX и вам необходимо сделать 2 вещи:

- Скачать из AppStore программу, которая называется XCode (это IDE для разработки программ для iOS и MacOSX, как таковой он нам не понадобится). Приготовьтесь скачать 3 с лишним гигабайта.
- Открыть XCode и затем через меню `Preferences -> Downloads` установить `Command Line Tools for Xcode`.

Если вы этого не сделаете, вы будете получать ошибки при установке различных пакетов в `Homebrew`.

После того, как вы выполните все шаги по установке `Homebrew` — проверьте, все ли в порядке, выполнив команду `brew doctor`, и следуйте инструкциям в случае, если на экране появятся сообщения об ошибках.

Менеджер пакетов Aptitude в Ubuntu Linux

В случае, если вы используете Ubuntu, вам не нужно устанавливать менеджер пакетов, т.к. он уже по умолчанию есть в системе. Более того, вам не понадобился бы даже и компилятор, т.к. все скачиваемые менеджером пакеты уже собраны для вашей платформы — если бы только для установки некоторых ruby-гемов компилятор не был необходим. Поэтому его (и необходимые зависимости) нужно предварительно установить следующей командой:

```
sudo apt-get install build-essential
```

Обратите внимание, что в данном случае используется префикс `sudo`, поэтому перед установкой вас попросят ввести пароль текущего пользователя (и этот пользователь должен быть добавлен в т.н. sudoers, т.е. список администраторов системы).

Git

Система контроля версий `Git` понадобится нам не только для разработки приложения, но так же и для установки RVM. Установить `Git` очень просто, достаточно ввести команду:

```
brew install git
```

или

```
sudo apt-get install git-core
```

О том, что такое `Git` и как и зачем мы будем его использовать, читайте подробнее в разделе Системы контроля версий

RVM

RVM расшифровывается как Ruby Version Manager и его придумали затем, чтобы можно было безболезненно и быстро устанавливать на один и тот же компьютер несколько версий Ruby и быстро между ними переключаться. RVM появился в тот момент, когда Ruby-коммьюнити разрывалось между Ruby 1.8.7 и Ruby 1.9 — две версии интерпретатора с существенными отличиями. Тогда было принято тестировать свои программы на обоих версиях, чтобы быть уверенным в их совместимости. Сейчас все в основном переехали на Ruby 1.9, но RVM продолжают активно использовать по нескольким причинам. Во-первых, новые версии Ruby выходят довольно часто (имеются в виду т.н. minor-versions и патчи). Во-вторых, RVM предоставляет несколько больше удобств, чем просто возможность держать различные версии Ruby на одном компьютере — например очень удобна возможность `создавать` *гемсеты*[2], особенно если вы работаете над несколькими проектами одновременно.

Как бы там ни было, даже если вам не нужно пользоваться гемсетами и держать несколько версий Ruby на одном компьютере, RVM является наиболее удобным способом установить самую свежую стабильную версию Ruby.

Чтобы установить RVM, вам нужно выполнить в терминале следующую команду:

```
curl -L https://get.rvm.io | bash -s stable --ruby
```

Эта команда скачает RVM, установит его и затем сразу же скачает, скомпилирует и установит стабильную версию Ruby (тут вам как раз потребуется, чтобы в системе был компилятор — `Command Line Tools for Xcode` в MacOSX или `build-essential` в Ubuntu).

Убедитесь, что после перезапуска терминала, версия ruby, доступная в системе по умолчанию, соответствует той версии, которую установил RVM — сделать это можно командой `ruby -v`. Вот теперь вы готовы начать установку необходимых гемов.

Необходимые ruby-гемы

Основная идея, которую нужно понимать при установке гемов, состоит в том, что они делятся на два типа: 1) те, что работают с тем или иным софтом установленным в системе и являются по сути ruby-интерфейсом к нему и 2) те, что не требуют системных зависимостей. Проблемы с установкой возникают как раз в случае первого типа гемов. И обычно они сводятся к тому, что в системе не установлен нужный софт или установлена неправильная его версия.

Давайте вкратце посмотрим, какие гемы из первой категории будет использовать рельсовое приложение:

- **Гемы для работы с БД: sqlite3 или mysql или pg.** Для установки каждого из этих гемов вам необходимо вначале установить саму соответствующую СУБД — sqlite3, MySQL или PostgreSQL в систему, естественно с помощью менеджера пакетов.
- **Гемы для работы с изображениями: rmagick или paperclip** требуют установленной в системе утилиты ImageMagick.

Часто, помимо самого софта, бывает необходимо установить т.н. -dev библиотеки. Они понадобятся, чтобы скомпилировать *native-extensions*[3] некоторых гемов. Проблема в основном возникает в Ubuntu, где -dev библиотеки нужно устанавливать отдельно через тот же самый менеджер пакетов, в то время как Homebrew в MacOSX устанавливает их автоматически.

После того, как RVM и необходимый софт установлены в систему, вы можете установить несколько гемов, которые нам необходимы для начала работы:

```
gem install bundler rails rspec
```

Все остальные гемы (в том числе гемы для работы с БД и изображениями), которые требуются нам для работы Rails-приложения, будут установлены позднее с помощью Bundler-a.

[1] Под зависимостями понимают другой софт и библиотеки, без которых устанавливаемая программа не сможет запуститься.

[2] Гемсеты (gemsets) в RMV — это наборы гемов установленных для текущей версии Ruby. Например у вас может быть два гемсета для версии ruby 1.9.3, где в одном из них будет установлен gem Rails версии 3.1, а в другом — gem Rails версии 3.2. Переключаясь между гемсетами и вызывая команду `rails -v` вы увидите, что у вас установлены разные версии рельс. О том, в каких случаях вам могут понадобиться гемсеты, вы сможете понять, почитав подробную документацию на сайте RVM.

[3] Native extensions — части гемов, написанные не на Ruby, а, например, на Си и требующие компиляции. Для этого компилятору как раз и необходимы -dev библиотеки.

Настройка dotfiles для удобной работы

Dotfiles — понятие, которое применяется к файлам настроек, как правило находящимся в вашей домашней директории ~. Отличительная особенность этих файлов заключается в том, что их имена начинаются с точки (dot), отсюда и название.

Файлы настроек в домашней папке существуют для разного используемого вами ПО, но наиболее известный из всех файлов — это файл настроек вашей *оболочки*[1] (а вернее, программы `bash`, которая и выступает в роли этой самой оболочки и запускается в окне терминала). Такой файл носит название `.bashrc` в Ubuntu и `.bash_profile` в MacOSX. Если вы не можете найти этот файл и вообще вы не видите dot-файлов в вашей домашней папке, то, скорее всего, программа которую вы используете для просмотра содержимого папок (Finder, Nautilus), по умолчанию скрывает от ваших глаз такие файлы — поэтому вам нужно покопаться в настройках и заставить ее показывать скрытые файлы.

.bashrc

Этот файл состоит из команд bash и является bash-скриптом, который автоматически запускается каждый раз, когда вы заново открываете окно терминала. Но какие настройки там можно задать? Ниже я приведу список настроек и соответствующих им команд, которые я считаю наиболее важными для своей комфортной работы в терминале. Вы можете добавить их в самый конец своего файла:

- Показывать полный путь к текущей папке в приветствии

  ```
  export PS1="\w"
  ```

  В MacOSX по умолчанию показывается не полный путь, а только имя текущей директори, что не очень удобно. Данная строка позволит показывать полный путь.

- Добавлять текущий *git-branch*[2] в конец пути в приветствии

  ```
  git_branch() {
    git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/\1/'
  }
  export PS1="$PS1 \$(git_branch)\$ "
  ```

  Чтобы увидеть как это работает на примере, вам необходимо перейти в папку, где существует git-репозиторий.

- Несколько сокращений для удобной работы с git-ом

  ```
  alias gca='git commit -a'
  alias gcam='git commit -a -m '
  ```

```
alias gl='git log --pretty=format:"%an, %ar, %h: %s"'
alias gd='git diff'
alias gb='git branch'
alias gco='git checkout'
alias gp='git pull --rebase'
alias gpp='git pull --rebase  git push'
alias gph='git push'
alias gm='git merge'
```

Ничего страшного, если вам не сразу понятно, как это работает. Главное, чтобы вы понимали, зачем мы добавляем это в файл настроек и как ведет себя терминал (bash) после перезагрузки. Поэкспериментируйте. Чтобы не закрывать и открывать окно терминала заново, вы можете ввести команду `source ~/.bashrc` в Ubuntu или `source ~/.bashrc` в MacOSX и настройки будут автоматически перезагружены.

.vimrc

Еще одним интересным dot-файлом является файл настроек редактора `Vim`. Так как выбор редактора гораздо менее предопределен, чем выбор оболочки, я не буду уделять ему столько же внимания. Упомяну только, что .vimrc написан на языке Vim script, который понимает редактор `Vim`. Вот пример файла `.vimrc` на моем макбуке:

```
:so ~/.vim/bundles.vim

:set number "shows line numbers on the left
:syn on
:set anti
:set gfn=Monaco:h12
:colorscheme jellybeans
:set cursorline
:set hlsearch

:set enc=utf-8
:set fencs=ucs-bom,utf-8,default,latin1

:set tabstop=2
:set shiftwidth=2
:set expandtab

" No trash files
:set nobackup
:set nowritebackup
:set noswapfile

" Handle buffers and tabs
:tab sball
:set switchbuf=usetab

" Folding
:set foldmethod=indent
:set foldlevel=100
```

```
:let mapleader = ","

" Don't break words in the middle while using wordwrap
:set formatoptions=l
:set lbr

:so ~/.vim/RussianMappings.vim
:so ~/.vim/SaveCopyPasteShortcuts.vim

:filetype plugin on
```

Обратите внимание на опции `tabstop`, `shiftwidth` и `expandtab` — это те опции, о необходимости которых мы говорили в разделе Что должен уметь текстовый редактор

В целом, существует большое количество интересных примеров dot-файлов и многие программисты выкладывают свои собственные файлы с пояснениями на Github-е. Погуглите "dotfiles" чтобы найти что-нибудь интересное — возможно вы позаимствуете у кого-нибудь полезную для себя настройку.

[1] Shell (оболочка) — программа, которая работает в окне терминала и обрабатывает вводимые пользователем команды (самые распространенные оболочки: sh, bash, zsh).

[2] О том, что такое *git branch* можно почитать в документации по Git.

Глава 3. Преждем чем начать: знания, которые вам понадобятся

В этой главе мы рассмотрим все, что вам скорее всего понадобится узнать в процессе изучения фреймворка RubyOnRails. Если в предыдущей главе мы рассматривали инструменты — ПО необходимое в разработке RubyOnRails приложений — то в этой главе мы посмотрим на технологии и языки, которыми вам неизбежно придется пользоваться.

В каждом из случаев я объясню, почему и в какой степени необходимо знать данную технологию или язык. Все перечисленное в этой главе необходимо для того, чтобы вы стали хорошим разработчиком Ruby On Rails приложений.

Хорошее знание языка Ruby и объектной модели

Очень часто люди берутся изучать фреймворк Rails без знания даже основ Ruby. Почему это происходит? Я полагаю, потому что очень распространен миф о "волшебстве" Rails и многим кажется, что просто установка фреймворка уже решит за вас задачу создания веб-приложения. Это не так. Основная проблема с которой, на мой взгляд, сталкиваются новички разбирающиеся в Rails без знания Ruby — это, условно выражаясь, невозможность решать проблемы, которые не были освещены в скринкасте "как сделать блог за 15 минут".

Если говорить более абстрактно, то без знания языка Ruby у вас отсутствует возможность свободно мыслить. Как если бы вы учили иностранный язык только одним просмотром сериалов, то вы бы знали заранее заготовленные и знакомые фразы, но не умели бы четко и свободно выражать все свои мысли в той форме, которая характерна именно для вас — для этого требуется знание грамматики. Или как если бы вы пытались написать песню выучив основные аккорды на гитаре — песню, конечно, написать можно, но ничего нового и оригинального вы скорее всего не привнесете, потому что вы совсем не знаете нотную грамоту, гармонию и композицию, и, следовательно, у вас отсутствует возможность учиться на огромном количестве музыкального материала. Многие люди, чему бы они не начинали учиться, покупаются на этот миф, что главное в этом деле — "душа". Избегайте этого мифа. Он опасен, потому что в итоге вы ничему не научитесь и вас никто не будет воспринимать всерьез.

Другая распространенная точка зрения заключается в том, что может быть было бы

полезно изучать Ruby параллельно с Rails. Не могу сказать, что это полностью необоснованно. Но такой подход требует гораздо больше дисциплины от ученика. Как правило, когда люди начинают знакомство с Rails, им хочется быстрее создать приложение любой ценой. В итоге они копипастят код из интернета, *генерируют его с помощью самих рельс*[1], но крайне редко пишут его самостоятельно. Естественно, скопированный или сгенерированный код очень часто остается до конца не понятым и изучение языка Ruby происходит гораздо медленнее.

Я хотел бы особо подчеркнуть этот момент: если вы копируете, генерируете или читаете чужой код или даже если вы пишите его сами — убедитесь, что вы понимаете его на 100%. Убедитесь что вы понимаете каждое слово этого кода и почему оно стоит именно в этом месте. Если у вас в чем-то возникают сомнения, не говоря уже о случаях, когда вы понятия не имеете, что означает эта строка кода — прекратите любую дальнейшую работу и разберитесь в коде.[2]

Вам не нужно знать весь синтаксис и возможность языка Ruby, чтобы работать с рельсами. Возьмусь предположить, что многие программисты, на каком бы языке они не писали, не знают всех особенностей языка. Ниже я приведу чеклист того, что вам понадобится хорошо понимать относительно языка Ruby, чтобы начать работу с Rails:

- Объектно Ориентированное Программирование: что такое классы и объекты и зачем они нужны
- Методы: публичные и приватные методы, вызов методов, аргументы, возвращаемые значения
- Переменные: локальные, инстансные (переменные экземпляра), область видимости переменных, переменные класса, глобальные переменные, константы
- Операторы: присваивания, сравнения, арифметические, логические
- Блоки: что такое замыкание (closure), самые распространные методы с блоками, как написать свой метод принимающий блок.
- Основные типы данных[3]: строки, числа (Integer, Float), дата и время, символы и чем они отличаются от строк
- Управляющие конструкции: if-else, циклы while и until

В конце раздела приведены ссылки на материалы, по которым можно освоить язык Ruby. Не начинайте изучать Rails, пока вы не проставите галочки возле каждого пункта чеклиста, приведенного выше.

[1] Генерируют чаще всего с помощью т.н. скаффолдинга (scaffolding) — об этом я поговорю отдельно в разделе Как не нужно программировать на Ruby On Rails.

[2] Начать разбираться в строке кода можно попытавшись ее немного видоизменить в соответствии с той теорией, которая у вас есть в голове, относительно того, что значит и как работает этот код. После того, как вы измените ее, следите за поведением программы и попытайтесь понять в чем тут дело. Или откройте irb и исследуйте изолированную проблему в ruby-консоли. В крайнем случае, вы всегда можете обратиться за помощью к другим людям — как это сделать рассказано в главе Что дальше: как разбираться в проблемах и кому задавать вопросы?.

[3] Вернее будет сказать — классы, представляющие типы данных.

[1] Книга "Programming Ruby", онлайн-версия, относительно старая версия (бесплатно)

[2] Книга "Programming Ruby", PDF или бумажная версия, новое издание покрывающее Ruby 1.9

[3] Вики-книга "Ruby Programming", онлайн-версия (бесплатно)

[4] Скринкасты по Ruby на русском (бесплатно)

Архитектура MVC

MVC — это т.н. называемый *шаблон программирования*[1], который использовался с 80-х годов для создания программ с графическим интерфейсом. Этот шаблон придумал программист на языке Smalltalk которого звали Trygve Reenskaug, в далеком 1979-ом.

Основная идея заключается в том, что ваша программа разделена на три части, каждая из которых представлена буквой в этой аббревиатуре: M — Model (модель), V — View (представление), C — Controller (контроллер). Ниже я расскажу зачем нужна каждая из частей.

Модель

Модели — это некие объекты содержащие в себе всю информацию о предметной области приложения. Например, если вы создаете интернет-магазин, у вас могут быть такие модели как `Item` (товар), `User` (пользователь), `Cart` (корзина) и `Order` (заказ). В Rails-приложении каждая модель представляет из себя отдельный класс, с экземплярами (объектами) которого мы будем в дальнейшем работать. Например, мы могли бы создать объект класса `Item` и задать для него разные свойства, например #price, #weight и #name.

В теории, такую модель (при условии грамотного подхода) можно вынести за пределы Rails-приложения и использовать в каком-нибудь другом фреймворке для создания, например, десктопной версии приложения. В этом случае, модель и в веб-версии и в десктопной версии будет одна и та же (то есть нам не придется переписывать код), в то время как остальные части приложения будут отличаться. Например, оформленный через десктопное приложение заказ будет успешно сохранен благодаря модели `Order` и затем его

можно будет просмотреть через веб-версию приложения.

На практике, Rails-приложения нечасто имеют десктопную версию и, скорее всего, код моделей не используется там повторно (потому что десктопные версии, обычно, пишутся не на языке Ruby). Мы тем не менее, сможем отдельно использовать наши модели для манипуляции данными в нашем приложении, когда мы будем изучать Rails-модели и "общаться" с ними через rails-консоль (подробнее об этом читайте в главе Модели, ActiveRecord и взаимодействие с БД)

Представление

О Views можно думать как о скине для вашей модели, который позволяет показывать данные пользователю в приятном и читабельном виде. Представление в Rails — это обычно erb или haml шаблоны, файлы содержащие html или haml разметку с вставкой ruby-кода в нужных местах. Например, часть такого файла может выглядеть следующим образом:

```
%h1= @item.name
%ul
  %li= @item.price
  %li= @item.weight
%p= @item.description
```

Это фрагмент haml-кода; если вы незнакомы с haml — самое время заглянуть на официальный сайт и быстренько посмотреть что это такое. Все, что идет после знака `=` является ruby-кодом. Таким образом, как можно догадаться, мы можем вставлять результат выполнения Ruby-кода в html-страницу и, в итоге, пользователь в браузере увидит информацию о товаре.

Подробнее про views читайте ниже в разделе Фронтэнд: как сделать хороший интерфейс и в главе Views, хэлперы и ассеты.

Контроллеры

Контроллер представляет из себя связующее звено между моделью и представлением. Он реагирует на действия пользователя, принимает решение о том, что делать с поступившей от пользователя информацией, а также что именно показать пользователю в ответ.

Если мы говорим о Rails, то тут контроллер по сути очень тесно связан с тем, что пользователь видит в адресной строке браузера. Например запрос на адрес `/orders/new` в нашем приложении скорее всего (если мы написали соответствующий код) отправится в контроллер `OrdersController` и попадет в его метод `#new`, в котором, в свою очередь, будет написан код призванный вывести в браузер пользователю шаблон с формой для создания

нового заказа.

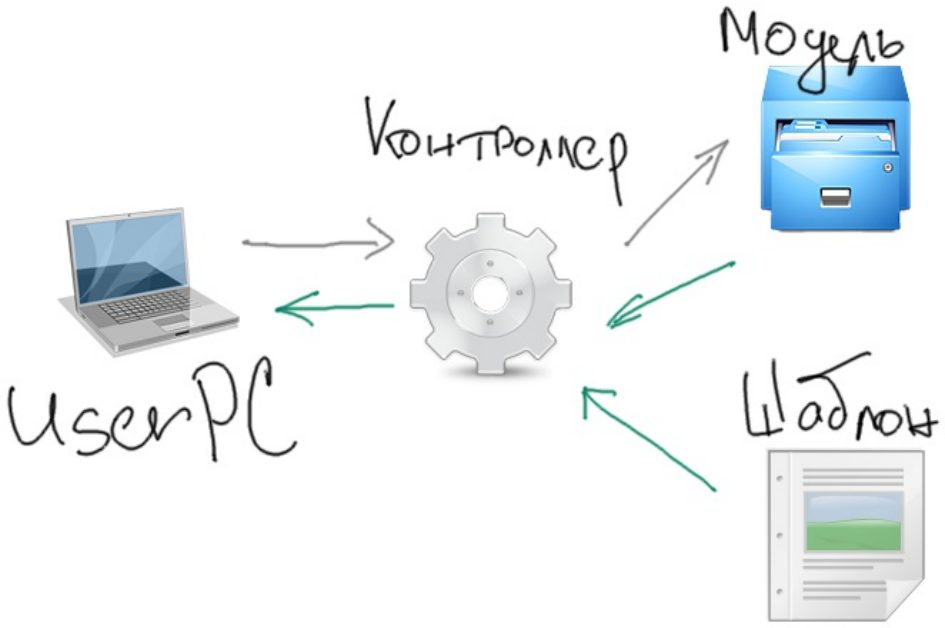
В Rails-приложении, контроллеры являются классами, имена которых заканчиваются на слово `Controller`. Эти классы, в отличие от классов для моделей, не нужно нигде создавать вручную при помощи метода `#new` — вам нужно только "заполнить" их правильным кодом. В Rails-приложении контроллеры как правило, должны содержать куда меньше кода, чем модели. Поэтому говорят, что контроллер должен быть "тонкий".

Как это работает вместе

Связывая все три компонента вместе, жизненный цикл нашего Rails-приложения выглядит следующим образом:

1. Пользователь нажимает на кнопку возле формы или переходит по ссылке или вводит url вручную — браузер отправляет соответствующий запрос
2. В зависимости от запроса, он попадает в тот или иной контроллер, который решает что с ним делать
3. Скорее всего, метод контроллера, который обрабатывает запрос, найдет ту или иную модель и, возможно, обновит ее свойства
4. Затем контроллер сделает редирект на другой url (в этом случае цикл повторится сначала) или примет решение отрендерить тот или иной шаблон из views

Эта же последовательность представлена на схеме ниже.



Если раньше вы никогда не сталкивались с архитектурой MVC, вам, возможно, многое осталось непонятным. Это нормально. Вы расширите свое понимание MVC когда мы перейдем в главам, непосредственно посвященным каждому из элементов этого паттерна и когда вы сможете на примере понять как все это работает вместе в Rails-приложении — это произойдет только когда вы напишите свой собственный код. Ну и, конечно, будет нелишним изучить материалы, приведенные в конце этого раздела.

[1] Шаблон программирования или Design Pattern — общее решение для некой проблемы, которое можно использовать повторно для решения такого же класса проблем.

[1] Understanding Model-View-Controller

[2] Статья в википедии

[3] Статья в википедии на русском

[4] История MVC от автора шаблона

[5] MVC-архитектура в Rails

Фронтэнд: как сделать хороший интерфейс

Фронтэндом обычно называют ту часть приложения, которая непосредственно

взаимодействует с внешним миром. В случае веб-приложения — это страница, которую пользователь видит в браузере. В отличие от других платформ для разработки приложений, веб еще не успел обзавестись популярным и простым редактором интерфейсов и стандартизированным набором виджетов (элементов интерфейса). То есть, если, например, вы хотите сделать интерфейс для iPhone приложения вам не нужно знать никаких дополнительных языков — вы просто рисуете интерфейс в редакторе. Попытки сделать то же самое для веба продолжают предприниматься, но на сегодняшний день, все что существует — это довольно разрозненная инфраструктура, состоящая из разных библиотек (например Twitter Bootstrap или jQuery UI), которые с трудом совместимы друг с другом. И для их интеграции, вам все равно понадобится минимальное знание html, css и javascript.

В будущем можно ожидать появление фреймворков, которые позволят разработчикам создавать веб-интерфейсы без знания html и css вообще. Я думаю, что это будет в первую очередь касаться одностраничных приложений, в которых в качестве основного и единственного языка разработки будет использоваться Javascript. Мы же, т.к. имеем дело с RubyOnRails, вынуждены знать упомянутые выше языки и ниже я приведу более подробное их описание и объясню предназначение каждого.

HTML, Erb и CSS

Эти три вещи — минимум, необходимый для разработки фронтэнда приложения на Ruby On Rails. Если вы уже с ними знакомы, у вас есть большое преимущество и изучение, например, haml и scss не составит труда.

В целом, *HTML (Hypertext Markup Language)*[1] — это специальный язык разметки (не язык программирования). Ключевое понятие в этом языке разметки — *теги*. Тэги представляют из себя специальные ключевые слова по обоим сторонам которых стоят символы < >. Смысл в том, что используя тэги, можно объяснить браузеру, что означает конкретная часть страницы. Например, такой код:

```
<h1>Телевизор с большой диагональю</h1>
Описание телевизора с большой диагональю
```

говорит браузеру, что текст "Телевизор с большой диагональю" нужно воспринимать как заголовок.

Erb по сути представляет из себя расширение html, которое понимает фреймворк Ruby On Rails. Erb-шаблоны — это html перемешанный с Ruby-кодом. Например:

```
<h1><%= @item.name %></h1>
Описание телевизора с большой диагональю
```

В данном случае мы подставили вместо статичного текста — значение свойства модели

`@item` и в дальнейшем, когда пользователь будет заходить на страницу с описанием конкретного товара, Ruby On Rails будет читать такой erb-шаблон, подставлять название товара и затем возвращать в браузер окончательный html (потому что браузер не понимает Ruby-код и erb!)

Идея здесь заключается в том, что наше Ruby On Rails приложение может держать 1 единственный erb-шаблон и использовать его для вывода информации о любом товаре. Гораздо разумнее, чем держать заранее созданную статичную html-страницу для каждого товара.

CSS (Cascading Stylesheets) — специальный язык, который также как и html понимают все браузеры и который позволяет задать внешний вид вашей страницы основываясь на указанных тэгах. Например, мы могли бы создать такое правило для нашего тэга h1:

```
h1 { font-size: 25pt; color: green; font-weight: bold; }
```

Этот css-код укажет браузеру, что текст находящийся внутри тэга h1 необходимо сделать размером 25 пунктов и отображать жирным и зеленого цвета.

В дальнейшем, я буду подразумевать, что вы знакомы с этими языками и не буду останавливаться и подробно объяснять что означает написанный на этих языках код. Если вы не знакомы с этими языками, проследуйте в конец этого раздела, где я привел ссылки на материалы, по которым их можно изучить.

haml и scss

`Haml` и `scss` являются языками, которые на данный момент фактически заменили html и css при разработке Rails-приложений. По своей сути это ruby-гемы, которые интерпретируют haml- и scss-код и затем компилируют его в привычный html и css. Зачем? Сравните два отрывка на html и haml-е:

```
<div class="content">
  <h1>@item.title</h1>
  <p><%= @item.desciption %></p>
</div>
```

```
.content
  %h1= @item.title
  %p=  @item.desciption
```

Не правда ли, второй вариант выглядит несколько более лаконично? Haml был придуман как раз для этой цели: сделать html-код лаконичнее, читабельнее и, кроме всего прочего, вам больше не придется помнить о незакрытых тэгах — haml сделает это за вас. С тех пор как я сам перешел на использование haml-а вместо erb, я думаю что сэкономил приличное

количество времени на том, что мне не приходится печатать лишнего (закрывающие тэги? я забыл об этом) и мой код в шаблонах практически всегда остается аккуратным и приятным для глаз[2].

Несколько другие преимущества дает scss. Это расширение языка css, возможно, не сделает ваши стили читабельнее и красивее, но зато оно добавляет ряд очень важных и так недостающих разработчикам фич, самая важная из которых, на мой взгляд — вложенные селекторы. Сравните два отрывка, написанных на css и scss соответственно:

```
.content h1   { font-size: 25pt; }
.content p    { font-size: 10pt; }
.content p b  { font-weight: bold; color: green; }
```

```
.content {
  h1 { font-size: 25pt; }
  p  { font-size: 10pt; b { font-weight: bold; color: green; }}
}
```

Как можно догадаться, если ваш файл со стилями будет расти и количество стилей увеличиваться, то возможность вкладывать селекторы один в другой становится очень удобным инструментом. Но, помимо этого, scss также добавляет к привычному css такие возможности как определение переменных, импорт файлов и функции.

Кроме scss, существует также и альтернативный язык с очень похожими возможностями — LESS (его использует Twitter Bootstrap). В целом, его внешний вид практически не отличается, поэтому его изучение, если вы уже знакомы с scss, займет максимум 10 минут. Я оставляю выбор расширения для css на усмотрение читателя.

Javascript и jQuery - асинхронные запросы

Ruby On Rails приложение может вполне успешно обойтись без единой написанной вами строки на Javascript-е. Другое дело, что, скорее всего, вы захотите сделать свое приложение несколько более привлекательным для пользователя. В частности, Javascript и jQuery понадобятся вам, если вы захотите делать ajax-запросы и манипулировать DOM-элементами.

Приведу пример. Предположим, что за каждый товар в вашем интернет-магазине пользователи могут голосовать. Возле изображения товара на странице мы расположим две кнопки с надписями "+" и "-". После того, как пользователь нажимает на одну из кнопок, нам необходимо сделать три вещи: послать запрос на сервер с сообщением и новом голосе за этот товар, обновить счетчик голосов и спрятать кнопку. Все эти три проблемы можно решить одновременно, если при нажатии на кнопку на сервер будет отправляться обычный запрос и страница в браузере будет перезагружаться. Но, конечно же, это очень неудобно для пользователя — проголосовав за товар, он вынужден ждать, пока страница

перезагрузится. Поэтому, тут нам на помощь придут асинхронные ajax запросы и манипуляция элементами на странице (обновить счетчик, спрятать кнопку) — все это реализуется на фреймворке jQuery. Вот как может выглядеть код, выполняющий описанную задачу:

```
$("#upvote_item").click(function() {
  var item_id = $(this).attr("data-item_id")
  $.ajax({
    url: "/items/" + item_id + "/upvote",
    dataType: "json",
    type: "post",
    success: function(response) {
      $("#upvote_item").hide();
      $("#item_votes_count").text(response.votes_count)
    }
  });
});
```

Сам jQuery — это фреймворк (или, некоторые предпочитают называть его просто библиотекой, т.к. до звания фреймворка он не совсем дотягивает) написанный на языке Javascript. Естественно, чтобы им пользоваться, вам необходимы базовые знания языка Javascript. Если у вас их нет, тогда вам следует сначала разобраться в самом языке при помощи этой очень популярной книги [5], а затем перейти к изучению jQuery [6]. До того, как вы начнете изучать jQuery, убедитесь, что вы понимаете как работают замыкания и анонимные функции в Javascript-е.

Почему вам не нужно быть дизайнером

Знание перечисленных выше языков и технологий не гарантирует, что у вас получится красивый интерфейс. Строго говоря, я бы разделил понятие красивый на 2 части: 1) визуально опрятный и 2) красивый. Первый возможно сделать без знания фотошопа, но только в том случае, если вы грамотно владете всем описанным выше и ваш вкус на интерфейсы развит в достаточной степени. *Красивый* интерфейс рисуют профессиональные дизайнеры и, скорее всего, вы не один из них, раз вы решили заниматься программированием.

Даже если вы немного знаете фотошоп и можете нарисовать там достаточно интересный GUI — дважды подумайте, стоит ли тратить на это время и усилия. С моей точки зрения, самое главное в любом ПО — выполнение поставленной задачи. В 9 случаях из 10 интерфейс не влияет на результат работы программы. Безусловно, и вам и пользователям приятно, если приложение выглядит красиво. Но, с экономической точки зрения, совершенствовать дизайн вашего приложения следует только тогда, когда он становится конкурентным преимуществом. Например, если вы делаете еще одну социальную сеть, то

скорее всего, вам нужно инвестировать время и деньги в дизайн, т.к. социальных сайтов в последнее время стало появляться все больше и больше, а задачи которые они решают, остаются все теми же. С другой стороны, если вы делаете приложение, которое решает какую-то задачу прежде никем не решенную и ваши клиенты уже готовы нести вам деньги — скорее всего, дизайн тут не при чем.

Кроме того, нежелание тратить время на отрисовку интерфейса вовсе не означает, что он должен быть некрасивым. Вы всегда можете использовать готовые решения, такие как `Twitter Bootstrap` для того, чтобы элементы на вашем сайте выглядели аккуратно. Это не является идеальным решением и, возможно, вам не полностью нравится то, как отрисованы виджеты в конкретной библиотеке, но поиск готовых GUI-решений часто является разумным компромиссом.

[1] К сожалению я не могу посоветовать книгу по HTML и вы не найдете ссылку на такую книгу в конце раздела. Дело в том, что я начинал свое знакомство с HTML более 10-ти лет назад и сейчас абсолютно не в курсе, какие достойные книги есть в продаже. Однако хорошая новость состоит в том, что они точно есть, причем я уверен, что существует и огромное количество качественного материала на русском языке.

[2] О том, почему это важно, читайте в разделе Несколько слов о читабельном коде

 *[1] Лучшая, судя по отзывам, книга по CSS на русском*

 *[2] Сайт-документация по HTML и CSS на русском*

 *[3] Сайт языка haml с документацией и инструкциями для новичков*

 *[4] Сайт языка scss (sass) с документацией и инструкциями для новичков*

 *[5] Книга "JavaScript. Подробное руководство."*

 *[6] Книга "jQuery. Подробное руководство по продвинутому JavaScript."*

 *[7] Сайт библиотеки jQuery*

Английский язык

Мы живем в мире, где английский язык стал де-факто стандартом во многих областях, в том числе в программировании. Все языки программирования используют английские слова и латинские символы в качестве основы. И, конечно, все самые актуальные дискусии в сети, связанные с программированием, ведутся на английском.

Тут бывают исключения, но даже эти исключения ненадолго остаются таковыми. Например дискуссии касающиеся языка Ruby, который придумал японец Юкихиро Мацумото, до сих пор активно ведутся и на японском. Тем не менее, поскольку английский язык знают гораздо больше людей и поскольку Ruby On Rails — фреймворк, который сделал язык Ruby популярным — придумали в США, то все-таки английский в мире Ruby и Rails

остается самым главным языком.

Вы можете задать себе вопрос: зачем мне учить английский, если уже есть большое количество материалов про Ruby и Ruby On Rails на русском? Ответ можно разделить на несколько пунктов:

- Перевод литературы часто оставляет желать лучшего. Этим занимаются не профессиональные программисты, у которых просто нет времени и желания выполнять такую работу, а люди, имеющие к программированию весьма смутное отношение. Естественно это сказывается на вашем понимании материала[1].
- Литература переводится с задержкой, соответственно вы получаете не самый актуальный материал.
- Выбор литературы на русском всегда будет гораздо меньше чем на английском. Не факт, что переводят самые лучшие книги. Скорее те, с чьими издателями была возможность заключить наиболее выгодные договоры.
- Помимо самой литературы, существует огромное количество мест в сети (форумы, листы рассылки, irc-чаты, github, сайты вопросов/ответов, твиттер) где люди общаются на заданную тему на английском. Без общения с другими программистами — не только теми, кто работает в вашей команде — процесс вашего обучения и степень профессионализма будет оставлять желать лучшего.
- Документация, как правило, переводится редко и часто обновляется. Чтение документации требует знания языка, т.к. все чаще авторы либ и фреймворков уделяют ей особое внимание и примеров кода в документации бывает гораздо меньше, чем обычных слов и предложений.

Вам не нужно становиться экспертом и говорить с идеальным акцентом. Что требуется, так это умение четко и правильно выражать свои мысли и не делать ошибок в письменной речи и четко (а не примерно) понимать, что пишут другие люди. Я не являюсь экспертом в области обучения английскому языку, поэтому воспринимайте указания приведенные ниже с долей скептицизма. Тем не менее, возмусь утверждать следующее:

- **Выучите грамматику.** Можно сколько угодно смотреть сериалы на английском и заучивать полюбившиеся фразы, но без знания грамматики вы не сможете свободно конструировать свои предложения и четко понимать, что имели в виду другие. В английском языке 24 времени[2]. Убедитесь, что вы знаете как образуется каждое время и в каких случаях оно употребляется.
- **Постоянно читайте.** Книги, статьи, даже Твиттер. Я убежден, что для того чтобы научиться грамотно говорить нужно не только слушать, но и читать. Причем

желательно не художественную литературу. Причина по которой я не рекомендую читать художественную литературу заключается в ограниченной возможности применять ее для обсуждения идей. А это именно то, что интересует программиста — абстрактные идеи. Если обобщать (тут я уверен, вызову гнев любителей хороших романов), то художественная литература часто описывает места, предметы и людей — это нас, как программистов, интересует в меньшей степени.

- **Смотрите новости и аналитику в зомбоящике.** Не смотрите слишком много фильмов и сериалов. По той же причине, по которой я не рекомендую читать художественную литературу.
- **Не зубрите слова.** Во всяком случае, когда вы уже накопите какой-то неплохой начальный словарный запас и вам не нужно будет лезть в словарь, чтобы переводить каждое второе слово в предложении. Вот в этот момент нужно переставать зубрить слова. Мой любимый способ учить слова такой:
    - когда я встречаю незнакомое слово я пытаюсь догадаться из контекста, что оно значит
    - затем я смотрю его в словаре, подтверждаю или опровергаю свою теорию и продолжаю читать дальше
    - пытаюсь употребить его в речи в ближайшее время, чтобы лучше запомнить
    - если я встречаю его снова и не могу вспомнить, что оно значит — я просто заново смотрю в словарь

  Я не подписываю перевод слов сверху (да и как это сделать с электронной книгой?) и не выписываю их в тетрадку.

Решайте сами, нужен ли вам преподаватель, чтобы выучить английский. Главную ошибку, которую допускают люди, нанимая преподавателя для обучения чему бы то ни было — подсознательное предположение, что преподаватель загрузит все знания в ваш мозг просто потому, что вы присутствуете на занятии. Этого не произойдет и, конечно же, основную работу вам придется как всегда проделывать самому. Максимум, что на мой взгляд может дать преподаватель — это фидбек и исправление ошибок.

[1] Хороший пример трудностей перевода приведен в одном из вопросов в раздел Ruby на сайте hasbrains.ru

[2] Интернет полон таблицами времен английского языка. Не все сходятся в определении "времени", поэтому сложно сказать сколько времен на самом деле.

Чтение документации

У новичков, мне кажется, возникает некий ступор, когда говоришь им "читай документацию". Документация воспринимается как нечто сложное в использовании и довольно скучное. Это не так. Мы подробнее коснемся вопроса работы с документацией в главе Что дальше: как разбираться в проблемах и кому задавать вопросы?, а здесь я просто хотел бы объяснить что такое документация, откуда она берется и как она выглядит.

Документация — это описание, обычно на английском языке, *API*[1] той или иной библиотеки или фреймворка. Возьмем, к примеру, документацию API самого Ruby. Она существует в виде веб-версии [1], но это только один из возможных форматов. Помимо него, существует еще возможность получить те же самые сведения через, например, утилиту `ri`[2]. Какой бы формат мы не выбрали, если мы хотим получить сведения о конкретном методе конкретного класса, мы можем с легкостью сделать это воспользовавшись поиском по документации. Предположим, я хочу узнать, как работает метод `#map` из класса `Array` — я найду его в документации и получу следующий текст[3]:

```
Invokes block once for each element of self. Creates a new array
containing the values returned by the block. See also Enumerable#collect
 If no block is given, an enumerator is returned instead.
```

И кроме этого текста, внизу, скорее всего, будет несколько примеров кода на языке Ruby, показывающего как использовать этот метод.

Примерно таким образом построена любая документация, не только для Ruby, но и для Ruby-гемов и фреймворка Rails в том числе. В случае относительно небольших гемов, документацию обычно составляют программисты, которые непосредственно являются авторами этого гема. В случае большого фреймворка, этим может заниматься отдельно назначенный человек, однако я не уверен, что это на самом деле имеет место быть в случае Rails или языка Ruby и что в целом это хорошая идея. Кто лучше всего понимает и знает как использовать API — тот, кому пришлось в нем разбираться или тот, кто его придумал?

Иногда вся доступная документация — это файл README, который обычно лежит в корневой папке Ruby-гема и именно он по умолчанию выводится Github-ом на странице с репозиторием. Во многих случаях этого бывает достаточно, чтобы понять как пользоваться библиотекой. Но очевидно, что в случае таких больших структур, как Ruby On Rails README-файл не решит проблемы, поэтому создатели Rails позаботились не только о написании API-документации [2], но также создали сайт Rails-guides [3], который в более структурированной манере объясняет как пользоваться фреймворком.

Таким образом, всю существующую документацию можно, в принципе, разделить на две части:

- API-документация — описание классов и методов
- Tutorials — инструкции по использованию, которые больше напоминают книги и имеют тенденцию быть структурированными по какому-либо принципу, например от простого к сложному.

README-файлы и Rails-guides как раз относятся ко второму типу.

Что касается первого типа документации, то его необходимость не всегда оправдана. Часто, программисту использующему библиотеку достаточно посмотреть в исходный код, чтобы понять как работает тот или иной метод. Кроме того, API-документация для Ruby-кода может быть сгенерирована автоматически при помощи различных утилит, самая популярная из которых — RDoc [4]. Идея состоит в том, что *парсер*[4] читает Ruby-код, генерирует стандартный внешний вид и минимальный текст документации за вас, а затем дополняет эту документацию комментариями из вашего кода (подробнее работа RDoc описана в книге "Programming Ruby"). Именно так создается документация для большинства (если не всех) Ruby-библиотек — то есть вручную верстать сайт на котором будет храниться документация для вашего Ruby-гема вам не придется.

Подробнее об эффективном использовании документации рассказано в разделе из главы Что дальше: как разбираться в проблемах и кому задавать вопросы?. А пока что, я рекомендую добавить сайты Документация к Ruby API и Документация к RubyOnRails API в избранное вашего браузера.

[1] API (Application Programming Interface) — это набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом). В Ruby мы будем понимать под этим совокупность всех классов и их методов предоставляемых данной либой, гемом, или фреймворком.

[2] Чтобы подробнее узнать, что такое `ri` и как пользоваться этой утилитой, наберите в терминале команду `man ri`.

[3] Если вы чувствуете, что прочесть и понять этот текст вам сложно, то это может говорить о том, что вам необходимо подтянуть английский или знание языка Ruby.

[4] Парсер — программа, которая читает код или текст и разбирает его на логические части в соответствии с инструкциями, которые были определены заранее.

[1] *Документация к Ruby API*

[2] *Документация к RubyOnRails API*

[3] *Сайт Rails-guides - инструкция по использованию Rails*

[4] *Документация к утилите по созданию документации RDoc*

Rails-сообщество: как оно устроено и как (и зачем) становиться его частью

Под сообществом мы будем в дальнейшем подразумевать всех людей так или иначе использующих Ruby On Rails и принимающих участие в общении, а также онлайн-инфраструктуру, которая необходима для организации этого общения.

Для начала давайте разберемся, почему программистам нужно общаться. Как бы очевидно это ни было, я хотел бы остановиться на этом немного подробнее, чтобы нарисовать правильную картину и объяснить, как вы сможете стать частью сообщества. Программисты общаются, чтобы обмениваться знаниями, которые необходимы нам для выполнения работы. Во время обмена знаниями происходит еще один важный и не столь заметный процесс — генарация новых знаний. Например два программиста обсуждающих в списке рассылки какой-либо баг Rails могут придти к выводу, что Rails API в этом месте был сконструирован не самым удачным образом. Это новое знание, в свою очередь, может привести к тому, что один из программистов внесет изменения в исходный код Rails и предложит эти изменения тому, кто отвечает за Rails-репозиторий[1].

Чем новее технология (а Rails — относительно молодой фреймворк), тем больше вопросов возникает у тех, кто ее использует. Поскольку Rails не только молодая технология, но еще и достаточно популярная, то интенсивность общения в сообществе очень высока по сравнению с другими сообществами построенными вокруг других технологий и языков. А это, в свою очередь, влияет на количество новой информации и новых решений, которые появляются каждый день: кто-то все время придумывает новые Ruby-гемы, которые решают задачу лучше, чем гемы созданные ранее (пример: `Paperclip` vs `attachment_fu` и `Devise` vs `Authlogic`). Постоянно появляются новые сайты призванные помочь программистам с навигацией в мире Rails (например Сайт "RubyToolbox" структурирует различные гемы и либы по категориям и популярности.). Да и API самих Rails значительно менялся за последние несколько лет, хотя сейчас этот процесс замедляется.

С одной стороны, живое коммьюнити и высокая интенсивность появления новых решений — замечательная вещь, потому что это позволяет более совершенным решениям занять место старых и неэффективных. С другой же стороны, это создает сложности для программистов: можно сказать — разбегаются глаза и очень сложно уследить за тенденциями. Знание API конкретной библиотеки, которую вы используете сегодня могут оказаться относительно бесполезными завтра, когда появится новая библиотека и все станут использовать ее. В том числе поэтому, важно твердо знать основы Ruby и понимать принципы, по которым программисты строят API библиотек (это приходит с опытом их использования), а не просто разбираться в конкретном API тех же Rails.

Чтобы стать частью сообщества, для начала можно просто читать. В конце раздела я привел ссылки на популярные места в сети, где на данный момент наблюдается большая активность Ruby и Rails программистов (это может измениться в будущем). Добавьте их себе в rss-ленту и по началу следите за обновлениями. Посмотрите, какой из сайтов вам читать интересно и наиболее полезно, обратите внимание как именно (в каком стиле)

другие программисты общаются на этом сайте и какие вопросы они задают — после этого попробуйте подключиться к общению сами. Подробнее о том как следует общаться и как задавать вопросы я расскажу в главе Что дальше: как разбираться в проблемах и кому задавать вопросы?

[1] В терминологии Git-а и Github-а — программист скорее всего сделает *pull-request* в репозиторий Rails.

[1] Сайт "RubyToolbox" структурирует различные гемы и либы по категориям и популярности.

[2] RubyGems.org является официальным сайтом Ruby-гемов с удобным поиском и статистикой.

[3] Русскоязычная гуглогруппа посвященная Rails

[4] Вопросы/Ответы по языку Ruby на сайте hasBrains.ru

[5] Вопросы/Ответы по RubyOnRails на сайте hasBrains.ru

Готовые решения

Когда вы разрабатываете Rails-приложение, перед вами стоят вполне конкретные задачи: реализовать регистрацию пользователей, позволить пользователям загружать изображения в свой профайл, выводить список товаров в вашем интернет-магазине постранично и так далее. Большинство из задач, которые стоят перед вами, стояли также и перед другими программистами. И поэтому, скорее всего, некоторые программисты, которые осознали, что та или иная задача будет появляться вновь и вновь в новых проектах, решили вынести решение задачи в отдельную библиотеку, оформить эту библиотеку в виде гема и выложить, например, на Github-е.

Безусловно такой гем может не вполне подойти вам. Но, как всегда, все сводится к экономической целесообразности: готов ли я пожертвовать значительным количеством времени и написать свой собственный код или я готов (хотя бы временно) пожертвовать небольшим количеством функционала, которого нет в чьем-то готовом решении, и пустить сэкономленное время на решение задач в моем проекте, которые могу решить только я? Этот вопрос обычно задают себе опытные программисты.

Новичкам же следует задавать себе совсем другие вопросы. Понимаю ли я на самом деле, как должно выглядеть решение моей проблемы? Хватит ли мне опыта и знаний написать свое собственное? Безусловно, написание собственного кода способствует обучению и, в этом смысле, выбор в пользу создания своего собственного решения кажется разумным. Но точно также (а в некоторых случаях еще больше) накоплению знаний и опыта способствует изучение грамотно написанного чужого кода или хотя бы использование чужого решения. Новичок может не осозновать всех подводных камней которые стоят перед разработчиком,

когда он решает конкретную задачу.

Скажем, вы хотите реализовать аутентификацию пользователей. Вроде бы все относительно просто: нужно хранить логин и пароль в базе данных и проверять пару логин и пароль при попытке входа. Но, оказывается, вам еще нужно решить как зашифровать пароль перед тем, как сохранять его в базу данных, потому что хранение plaintext паролей делает вашу систему уязвимой к утечкам данных (получивший доступ к БД — хотя бы на чтение — хакер фактически сможет залогиниться от имени любого пользователя). Эта и множество других проблем связанных с аутентификацией уже решены в гемах Devise и Authlogic.

Первый шаг, в случае выбора готовых решений — выстроить правильное предположение: могла ли моя задача стоять перед другими программистами и если да, то насколько вероятно, что другие программисты уже придумали для этого достаточно гибкое и устраивающее меня решение. Правильно отвечать на этот вопрос становится легче с опытом.

После того, как вы определили вероятность существования готового решения, **вторым шагом** будет начало его поисков. Делать это можно разными способами: погуглить, поискать на специализированных сайтах вроде Ruby Toolbox или спросить других Ruby-программистов в одном из мест в сети, где они ведут дискуссии.

Третьим шагом, после того как вы нашли конкретных кандидатов, станет оценка их пригодности. Это следует делать по нескольким параметрам:

- Посмотрите на дату первого релиза библиотеки. Если это произошло довольно давно — возможно она устарела.
- Также посмотрите когда был последний релиз или последний коммит. Если оба события произошли больше года назад, возможно библиотека заброшена и уже потеряла свою актуальность.
- Внимательно почитайте документацию. Если документации нет или она написана плохо, есть вероятность, что автор не особо думал о пользователях своей библиотеки и вам придется туго, когда вы будете разбираться в том, как ее использовать. Кроме того, есть вероятность, что он также небрежно подошел и к написанию кода.
- Посмотрите на количество followers (на Github-е сейчас есть три варианта, которые можно квалифицировать как followers: watchers, starred и forked). Если их мало — возможно библиотека скоро будет заброшена автором.
- Наконец посмотрите, потрудился ли автор протестировать свой код юнит-тестами — это важный показатель надежности кода и ответственности автора.

В целом, выбор готового решения — довольно ответственная задача. Но не поддавайтесь на соблазн оставить это "сложное" дело — разбираться в чужом коде или API — и написать "свое". Особенно если вы новичок, скорее всего это будет неверный выбор. Лучшее что вы можете сделать — это учиться на чужом опыте и на чужом коде. Отказ от готового решения должен быть осознанным и иметь вполне конкретные причины. Если ваша аргументация такого решения сводится к "я хочу свое", вы поступаете неверно.

Несколько слов о читабельном коде

Читабельный код имеет отношение не столько к Rails и Ruby, сколько к программированию вообще, но наблюдая за тем, как новички пишут код и какие стилистические ошибки они делают, я не могу не включить небольшой раздел посвященный этому вопросу в книгу.

Прежде всего, я должен сказать, что подхожу к вопросу форматирования кода крайней серьезно и, возможно отчасти, это оправдано только субъективными ощущениями не имеющими серьезных корней в реальности. Но по моим наблюдениям, большинство хороших программистов также фанатично подходят к форматированию кода. Настоящая причина кроется в том, что хорошо отформатированный код проще читать.

Самое главное, что обязательно стоит помнить — необходимость отделять вложенные блоки кода двумя пробелами. Рассмотрим код определяющий вот такой метод:

```
def validate_author_rights
  unless section.try(:user_id) == user_id || user.is_admin?
  errors.add(:base, "user is not owner")
  end
end
```

В данном случае автор допустил грубую ошибку форматирования: внутри блока `unless` строка `errors.add(:base, "user is not owner")` не была сдвинута на два пробела вправо. Из-за этого читабельность кода резко ухудшилась и кажется, что все три строки в теле метода никак не зависят друг от друга и выполняются в любом случае одна за другой, в то время как в действительности это не так.

Как правило, основное форматирование сводится к выравниванию частей кода относительно каких-то ключевых слов или символов, например сравните два идентичных по своему предназначению, но отличных в формартировании отрывка кода:

```
has_many :votes, as: :votable, dependent: :delete_all
has_many :comments, as: :commentable, dependent: :delete_all
```

```
has_many :votes,    as: :votable,     dependent: :delete_all
has_many :comments, as: :commentable, dependent: :delete_all
```

Во втором случае мы выровняли код по названию ключей хэша, передавамоего в метод `#has_many` и картина сразу преобразилась. Естественно, мы смогли это сделать потому, что интерпретатору Ruby все равно сколько пробелов стоит после запятой, разделяющей элементы хэша. Вот еще один пример, но теперь мы выравниваем код по оператору присваивания и арифметическим операторам:

```
a = 10 + 1423 - 43
summ = 1321 + 5 - 78
```

```
a    = 10   + 1423 - 43
summ = 1321 + 5    - 78
```

У программистов есть различные предпочтения при форматировании кода, но очень важно следовать каким-то постоянным правилам, которые вы сами для себя выбираете. Например, вот код из модели в Rails-приложении:

```
class Answer

  belongs_to :question
  belongs_to :user

  has_many :votes,    as: :votable,     dependent: :delete_all
  has_many :comments, as: :commentable, dependent: :delete_all

  validates(:question, :user, presence: true)
  validates(:body, length: { minimum: 20 })
  validate :validates_uniqueness_of_question_and_user, on: :create

end
```

Обратите внимание как автор неожиданно решил использовать круглые скобки вокруг аргументов при вызове метода `#validates`. Его код — правильный и ошибок не вызовет, но вот форматирование — непоследовательно. Автору следовало бы либо использовать круглые скобки везде, либо не использовать их нигде при вызове методов в контексте класса. В данном случае, Rails-программисты имеют негласное соглашение не использвать скобки, поэтому именно так здесь и стоило поступить.

Лучше всего научиться форматировать свой код можно изучая чужой код. Именно это я настоятельно рекомендую вам делать как можно чаще.

Как не нужно программировать на Ruby On Rails

Некоторое внимание хотелось бы уделить ошибкам, которые совершают начинающие

программисты, когда приступают к изучению Ruby и Rails. Все эти ошибки можно было бы охарактеризовать следующим образом: они предлагают программисту решение текущей проблемы в ущерб знаниям и опыту в целом.

Генерирование кода

Рассмотрим самый распространенный пример — скаффолдинг (scaffolding). Без примеров его использования, пожалуй, не обходилась до сих пор ни одна книга по Rails (в этой книге примеров работы со скаффолдингом вы не найдете!). Но что такое скаффолдинг? По своей сути это утилита для командной строки, которая генерирует за вас ряд файлов и их содержимое. Например, вы начинаете работать с вашим интернет-магазином и добавляете с помощью скаффолдинга сущность "Товар". Утилита за вас создаст файлы с миграцией, моделью, REST-контроллером и шаблоны в папке `views`. Если открыть эти файлы и взглянуть на них, вы увидите, что утилита также сгенерировала за вас достаточно большое количество кода. Теперь, возможно, в вашем браузере уже можно просматривать товары и кажется, что ваш магазин приобрел значительную часть своего функционала. Но задайте себе один простой вопрос: понимаете ли вы весь код, который был сгенерирован?

Понимание кода — ключевой момент. Как я уже упоминал ранее, если вы не понимаете код, который находится перед вами, то скорее всего у вас появятся проблемы, как только понадобится внести малейшие изменения. Генерирование кода скаффолдингом не означает умение использовать Рельсы. Более того, я бы сказал, что это крайне вредный подход, потому что он создает иллюзию прогресса. Поэтому, мой совет в данном случае простой: забудьте про скаффолдинг и сосредоточьтесь на изучении моделей, контроллеров, хэлперов и views по отдельности.

То же самое относится к любой библиотеке или гему, которые генерируют код за вас. Скажем, если вы просто создаете модель с помощью генератора `rails g model item`, то генератор создаст два файла — `db/migrate/[текущее время в секундах с начала эпохи]_create_items.rb` и `app/models/item.rb`. Если взглянуть на эти файлы, то количество кода сгенерированного за вас — минимально (в случае `item.rb` это просто определение класса Item). Такие генераторы не являются опасностью для новичков и их можно и нужно использовать. Другое дело генераторы, результатом работы которых является большое количество строк кода не каждую из которых вы понимаете. Именно таких генераторов нужно опасаться. На каком бы языке вы не программировали, помните, что генераторы кода — это потенциальная опасность для вас, как разработчика.

Нежелание посвящать время написанию кода

Еще одна ошибка, которую делают новички — они тратят больше времени на чтение книг / просмотр скринкастов / обдумывание проекта / настройку среды разработки, чем на написание кода. До тех пор, пока вы не решитесь испачкать руки написанием кода, прогресса не будет. Начните писать код сейчас. Даже если вы не понимаете что писать — пишите что-нибудь. Напишите небольшой отрывок кода и попробуйте запустить программу. Не произойдет ничего страшного, если программа не заработает и выдаст ошибку. Напротив, ошибка в программе — отличная вещь, которая позволит вам понять, что вы делаете не так, исправить свой код и научиться чему-нибудь новому.

Многие не посвящают достаточно времени написанию кода, по объективным причинам, например — работа, которая требует их внимания (часто работа связана совсем с другими технологиями и люди во вне рабочее время пытаются освоить Rails). К сожалению, очень сложно ожидать, что человек научится программировать на Ruby и Rails на достойном уровне и будет способен сделать что-нибудь самостоятельно, если он не посвящает написанию кода минимум 2-3 часа (лучше больше) в день. Найдите это время.

Нежелание читать ошибки

Если вы будете писать код, у вас неизбежно будут возникать ошибки. Это примерно как коррупция неизбежно возникает там, где есть государство (где-то ее меньше, где-то больше, но есть везде). Очень важно не игнорировать содержание этих ошибок, а внимательно читать, что вам пишет интерпретатор Ruby. Интерпретатор всегда покажет в какой строке произошла ошибка и объяснит ее природу. Самая простая ошибка — синтаксическая (Syntax Error): означает, что вы просто написали невалидный Ruby-код, который достаточно легко исправить (скорее всего, вы где-то забыли закрыть скобку или поставить ключевое слово `end` — в общем что-то в этом роде). Чем больше вы будете писать код и чем больше ошибок вы будете встречать, тем быстрее вы будете учиться, и тем меньше этих ошибок вы будете получать в будущем. Единственный способ научиться писать код, который генерирует минимальное количество ошибок — это прилежно и медленно работать над каждой из них.

Оптимизация кода

По-английски то, о чем я хочу рассказать называется *Premature Optimization* и этот термин означает, что программист пытается оптимизировать код, когда это еще не является основной задачей и не сильно влияет на результат. Возьмем наш интернет магазин и представим, что вы решили добавить кэширование каталога. Возможно, это не самая плохая идея, но если посмотреть на проблему повнимательнее, то, скорее всего, вам станет ясно, что после запуска ваш магазин не получит аудиторию Amazon.com — разумно

предположить, что, по началу, на сайт будет приходить очень мало посетителей. То есть кэширование не даст никакого выйгрыша вам, т.к. сервер прекрасно справится с нагрузкой и без кэширования, в то время как посетителям кэширование может создать определенные неудобства — например добавленный в каталог товар будет появляться в нем не сразу. А учитывая то время (пусть и не очень большое), которое вам понадобится для настройки Memcached и добавления кода в ваше приложение, мы имеем типичный случай premature optimization.

Естественно, вопрос может касаться не только кэширования. Оптимизировать можно что угодно и ваша основная задача, чтобы не попасть в эту ловушку, будет сводиться к тому, чтобы мыслить следующим образом: "мне нужно, чтобы это просто заработало." Исправить и усовершенствовать код вы сможете позже. Естественно это не означает, что вы должны небрежно подходить к написанию кода — оптимизация и качество кода все-таки разные вещи. Аккуратно и понятно написанный код не означает, что он оптимизирован, и, точно так же, оптимизированный код далеко не всегда выглядит аккуратно.

Длинные методы

Если вы заметили, что ваш метод (в каком бы классе он не находился) стал очень длинным — это свидетельствует о том, что вы делаете что-то не так. Идея разделять код на процедуры (методы) — это один из самых старых способов структурировать программу и было бы очень глупо его избегать. Когда ваш метод слишком длинный, его становится сложно читать, а значит в него сложнее вносить изменения. Он может выполнять задачу, но спросите себя: сможете ли вы в нем разобраться, если взглянете на этот код через полгода? Если ответ отрицательный — попытайтесь разбить метод не несколько независимых небольших методов.

В этом разделе я постарался собрать все наиболее важные ошибки, которые совершают начинающие программисты. Если вам интересно, какие еще ошибки могут вас подстерегать, попробуйте погуглить learning programming mistakes.

Глава 4. Создание Rails-приложения

В этой короткой главе мы посмотрим на то, как создавать Rails-приложение и изучим его структуру. Я объясню для чего предназначены все директории в Rails-приложении и какого рода файлы находятся в каждой из них.

Генерирование необходимых файлов

Первый шаг в создании Rails-приложения — запустить генератор, который создаст базовую структуру файлов и папок. В главе Преждем чем начать: знания, которые вам понадобятся я упоминал, что использование генераторов препятствует процессу понимания того, над чем программист работает. В случае с генератором Rails-приложения это не совсем верно, потому что через некоторое время вы будете прекрасно ориентироваться в структуре и содержимом сгенерированных файлов. То есть, как грамотный разработчик, вы на самом деле будете знать и понимать зачем нужен каждый файл и каждая директория и продолжать пользоваться генератором в дальнейшем, для создания новых приложений. Так что в этом случае мы сделаем исключение.

И так, чтобы создать новое Rails-приложение, вам для начала нужно убедиться, что в вашей системе установлен гем `rails`. Это можно проверить выполнив команду `rails -v`. Если терминал ответит, что такая команда не найдена, установите Rails с помощью команды `gem install rails`[1]

Теперь вы готовы сгенерировать новое Rails-приложение следующей командой:

```
rails new my_app_name
```

Последний аргумент — `my_app_name` — является, как нетрудно догадаться, названием вашего приложения (его нужно написать без пробелов). После запуска этой команды, в текущей папке (убедитесь, что вы находитесь там, где вы хотите создать приложение: на моей машине я предпочитаю держать все Rails-приложения в папке `~/Work`) будет создана папка `my_app_name` в которой будет создана структура нового приложения. Выполненная команда выведет вам в терминал список всех файлов и папок, которые создал генератор. Кроме того, после того как файлы были созданы, будет автоматически запущена команда `bundle install`, которая установит все гемы, перечисленные в файле `Gemfile` — в дальнейшем `bundle install` вам нужно будет запускать самостоятельно каждый раз, когда

вы добавляете, обновляете или удаляете гем в файле `Gemfile`, но сейчас эту работу за вас уже проделал генератор Rails-приложения.

[1] Естественно подразумевается, что вы уже установили все необходимое ПО обсуждавшееся в главе Инструменты, которые вам нужны, чтобы начать работу включая RVM и последнюю стабильную версию Ruby.

Структура файлов в приложении

Ниже я объясню, зачем нужна каждая директория (и в некоторых случаях остановлюсь на поддиректориях) в Rails-приложении, а также объясню назначение нескольких важных для приложения файлов.

- **app/** В этой директории находится все, что составляет модель MVC. С файлами, которые находятся в поддиректориях вы будете работать чаще, чем с чем-либо еще в Rails-приложении.
    - **app/models/** — здесь находятся файлы с моделями; как правило в этих файлах содержатся классы, которые наследуются от ActiveRecord-а.
    - **app/controllers/** — папка где находятся файлы с контроллерами; имена файлов, находящихся в этой папке должны заканчиваться на `_controller.rb`.
    - **app/views/** — внутри этой папки находятся другие поддиректории, обычно они соответствуют именам контроллеров. Например, если у вас в приложении есть контроллер `ItemsController`, то скорее всего у вас будет существовать папка `app/views/items/` в которой уже будут находиться .erb или .haml шаблоны. Исключение здесь составляет папка `layouts/`, в которой находятся шаблоны т.н. лэйаутов — они используются всеми контроллерами для рендеринга одинаковой основы разных страниц.
    - **app/assets/** — папка, в которую вы будете помещать css-стили (`app/assets/stylesheets`), javascript-файлы (`app/assets/javascripts`) и изображения (`app/assets/images`).
- **config/** Как несложно догадаться по названию, в этой директории находятся файлы с настройками приложения, самый важный из которых называется `application.rb`. Если заглянуть внутрь этого файла, вы найдете там класс, который соответствует названию вашего приложения, в нашем случае это может быть `MyAppName`. Кроме того, внутри есть еще несколько папок, которые также

важны для разработчиков.

    - **app/config/initializers/** служит домом для небольших файлов, код из которых выполняется сразу после запуска Rails-приложения (отсюда и название директории).
    - **app/config/environments/** по умолчанию содержит 3 файла: `development.rb`, `test.rb` и `production.rb` отвечающих за настройки конкретной среды, где запущено приложение (подробнее об этом в разделе Среды (Environments))
    - **app/config/locales/** содержит файлы с переводом приложения на различные языки (по умолчанию там содержится пустой файл для английской версии). Подробнее о т.н. интернационализации приложения можно почитать на сайте Rails-guides [1], т.к. тема выходит за рамки этой книги.
    - **app/config/routes.rb** — специальный файл, который содержит инструкции для Rails-приложения о том, в какой контроллер нужно направлять тот или иной запрос. Например инструкция `root :to => "items#index"` говорит о том, что запрос к главной странице сайта должен направляться в контроллер `ItemsController` экшену `index`.

- **db/** — директория содержащая миграции для базы данных (`db/migrations/`), файлы базы данных на локальной машине (только если вы используете sqlite) а также файл с текущей схемой БД — `db/schema.rb`. В этот файл нельзя вносить изменения вручную, но его полезность для программиста заключается в том, что открыв его, можно увидеть какие поля для каких моделей существуют (этого нельзя понять открыв файл с моделью ActiveRecord) или, если говорить более точно, какие таблицы и какие столбцы в этих таблицах существуют на данный момент в базе данных.
- **doc/** — предполагается, что там должна храниться документация для Rails-приложения, но очень редко можно встретить программиста, который бы пользовался этой директорией. Полезность документации невозможно отрицать, но т.к. структура большинства Rails-приложений похожа, разобраться в них обычно не составляет труда и все что требуется — хорошо написанный README в корневой директории.
- **lib/** — здесь, скорее всего, будут находиться ваши модули, которые вы будете инклудить в свои модели или контроллеры. Или классы, которые не принадлежат ни к моделям, ни к контроллерам. Кроме того, в этой папке есть подпапка

`lib/tasks/` в которой можно создавать свои собственные *rake-маски*[1].

- **log/** — здесь содержатся файлы с логами для различных сред приложения. Запущенное Rails-приложение записывает всю (или только важную, в зависимости от настроек) активность в лог-файл, соответствующий среде, в которой приложение запущено. Эти файлы бывают полезны, чтобы изучить какой-то конкретный запрос, пришедший на сервер, и разобраться в ошибке.
- **public/** — в более ранних версиях Rails в этой папке хранились css-стили, javascript-файлы и изображения, но сейчас все это по определенным причинам, которые мы обсудим позже, переехало в папку `app/assets`. По сути, папка `public/` — корневая папка вашего приложения и все файлы в ней будут доступны посетителям сайта напрямую. Например, в ней есть файл `index.html` — если его не удалить, то обращение к главной странице вашего приложения будет выводить именно этот файл, а не то, что вы задали в `config/routes.rb`.
- **script/** — директория, где лежат запускаемые из терминала программы, которые так или иначе необходимы для работы Rails-приложения. По умолчанию там уже есть файл, который называется `rails` — это скрипт, запускающий rails-сервер (то есть, по сути, ваше приложение).[2] Вы также можете самостоятельно создавать свои скрипты в этой папке. Например, в вашем приложении может быть скрипт, который очищает кэш и на своем сервере вы можете настроить утилиту `cron`, чтобы она запускала этот скрипт каждые 15 минут.
- **test/** — директория с unit-тестами. В дальнейшем мы ее удалим, т.к. будем использовать `rspec` и вместо нее у нас появится директория `spec/`. Подробнее о тестировании можно прочитать в главе Тестирование (моделей и контроллеров) и зачем это нужно
- **/tmp** — папка с временными файлами. Сейчас она не должна нас волновать, в ней нет ничего важного.
- **/vendor** — раньше в этой папке находились плагины для Rails, но т.к. на данный момент большинство плагинов устанавливаются в качесте гемов, эта папка на development-машине будет пустовать. На вашем сервере, именно в нее (если у вас правильно настроен `rvm` и `capistrano`) будут устанавливаться гемы).
- **Gemfile** — файл для утилиты Bundler [2], который содержит список гемов, необходимых вашему приложению. Вы можете добавить те гемы, которые вам необходимы и затем запустить команду `bundle install` (на забудьте после этого перезапустить rails-сервер). Хотел бы обратить внимание, что в корневой директории также существует файл `Gemfile.lock` — его ни в коем случае нельзя

редактировать вручную. Если вы хотите внести изменения в список гемов, которые нужны вам для вашего Rails-приложения, вы должны редактировать именно `Gemfile`.

[1] rake-таск — это небольшой скрипт, который можно запустить из терминала командой `rake` за которой обычно следует название этого таска, например `rake db:migrate`. При разработке Rails-приложений применяется ограниченное количество rake-тасков, посмотреть на все доступные в вашем приложении можно командой `rake -T`. Создание своих собственных rake-тасков выходит за рамки этой книги, но на эту тему доступно большое количество материалов в сети.

[2] Запустить rails-сервер можно двумя способами — командой `rails s`, а также командой `script/rails`. Первая — это просто удобный способ вызвать вторую.

[3] Подробнее о том, что такое Bundler и зачем он нужен можно прочитать на сайте этого гема [1].

 *[1] Интернационализация (i18n) в Rails.*

 *[2] Сайт гема Bundler с подробным описанием и инструкциями по использованию.*

Среды (Environments)

Любое Ruby On Rails приложение умеет запускаться в нескольких т.н. средах. Под *средой* в данном случае понимают набор настроек, с которыми приложение будет запущено. Например, на локальной машине вы можете использовать базу данных sqlite, а на сервере — PostgreSQL. Чтобы наше приложение знало о том, с какой базой ему предстоит работать, оно будет читать соответствующий своей среде файл с настройками.

Файлы настроек для каждой среды находятся в папке `config/environments/`. По умолчанию там существует три файла: `development.rb`, `test.rb` и `production.rb` — их названия соответствуют названием сред. `Development` — это среда, в которой ваше приложение обычно работает на локальной машине. `Test` — среда, в которой запускаются все unit-тесты приложения (это НЕ среда для тестовой копии приложения, которую обычно тоже запускают на сервере). Наконец `Prodcution` — среда, в которой должно будет работать ваше приложение на сервере.

Чем же отличаются настройки в этих файлах? Давайте посмотрим на некоторые их них. Откройте все три файла, чтобы они были у вас перед глазами, а ниже я приведу примеры нескольких важных настроек, которые будут отличаться для каждой среды (за основу я беру настройки из `development.rb`):

- `config.cache_classes = false` — настройка, которая гарантирует, что каждый раз, когда происходит обращение к приложению (загружается новая страница в браузере), Rails будет читать любые файлы с классами в папке *app/* заново. Это очень удобно при разработке приложения на локальной машине, т.к. не

приходится каждый раз перезапускать это приложение (изменения принимаются почти "на лету"), но зато значительно повышает нагрузку на компьютер, что неприемлемо и бессмысленно на удаленном сервере, где запущено ваше приложение.

- `config.action_mailer.perform_deliveries = false` — инструктирует приложение, что не следует отправлять почту с локальной машины (например если вы написали приложение, которое рассылает уведомления пользователем о новом комментарии к их посту, с локальной машины такое уведомление не уйдет). С одной стороны это полезная настройка, но я предпочитаю быть уверенным, что почта отправляется корректно, поэтому я обычно настраиваю на локальной машине почтовый сервер Postfix и устанавливаю эту опцию в значение `true`.
- `config.assets.compress = false` — настройка, которая позволяет задать необходимость сжимать css и js-файлы. Это крайне полезно в production-среде и позволяет сократить размер файлов, которые должен загрузить пользователь, чтобы посмотреть вашу страницу. В то же время, это бесполезно и иногда даже неудобно на локальной машине, т.к. разработчикам часто бывает нужно залезть в исходный код css и js из браузера.

С остальными настройками на данный момент я предлагаю вам ознакомиться самостоятельно — файлы снабжены подробными комментариями, а если этого недостаточно — Гугл полон полезной информации на эту тему. По мере изучения Ruby On Rails мы будем возвращаться к этим файлам и рассматривать другие опции.

Еще один важный момент заключается в том, что файлы настроек для каждой среды — не единственное место, в котором имеет значение в какой среде запущено ваше приложение. В любом месте вашего Rails-приложения вы можете получить информацию о текущей среде следующим образом: `Rails.env`. Например, в том месте Rails, где определяется с какой базой данных запущенное приложение должно работать сначала читается файл `config/database.yml` и затем, в зависимости от текущей среды, выбирается тот или иной адаптер для базы данных (если вы сейчас посмотрите на файл `database.yml` в вашем приложении, вам станет понятно о чем я говорю).

Наконец, следует упомянуть, что на практике, у вас будет больше чем три среды. В моих приложениях у меня их обычно целых пять: помимо основных трех я добавляю еще среду `migration` и `staging`

- Среда `migration` необходима мне для запуска миграций при деплое приложения. Дело в том, что чтобы добавить в базу данных новые таблицы пользователю этой

базы[1] необходимы соответствующие права, в то время как пользователь БД, которого использует запущенное Rails-приложение таких прав не требует (обычно ему нужно читать и добавлять записи в уже существующие таблицы). Таким образом, я просто создаю двух пользователей в БД — `migration` и `production` — и две среды с таким же именем. И затем в файле `database.yml` я прописываю данные каждого из пользователей для соответствующей ему среды. Выглядит это примерно так:

```
migration
  adapter: postgresql

  # обратите внимание, название БД - одинаковое
  database: my_application_production

  encoding: utf8

  # а имя пользователя - отличается
  username: migration

  password: 1234567
  host: localhost

production:
  adapter: postgresql
  database: my_application_production
  encoding: utf8
  username: production
  password: 7654321
  host: localho
```

- Среда `staging` необходима для запуска тестовой версии приложения, но не локально, а на сервере. Это бывает особенно полезно в больших проектах с несколькими разработчиками, или когда вам нужен постоянный фидбек от клиента, для которого вы разрабатываете приложение.

Фактически, две описанные среды будут чаще всего идентичны в своих настройках среде `production` — достаточно сделать симлинк на файл production.rb[2]. Отличия, как правило, будут как раз в файле `database.yml` и, возможно, в других местах.

У вас мог возникнуть разумный вопрос: а как же запустить приложение и назначить ему при запуске ту или иную среду? Нет ничего проще:

```
rails s RAILS_ENV=production
```

Этот код запустит приложение в среде `production`. Да, это можно сделать на вашей локальной машине, правда это не будет иметь большого смысла. По умолчанию приложение запускается в среде `development`.

[1] Системы управления базами данных тоже имеют такое понятие как пользователи (если только речь не идет об sqlite). Фактически, когда ваше Rails-приложение запущено, оно подключается к базе данных используя имя пользователя и пароль указанные в файле `database.yml`. Естественно, такой пользователь должен быть предварительно создан в этой СУБД администратором сервера. Как это сделать, мы рассмотрим в главе Настройка сервера и деплой приложения

[2] Чтобы это сделать, находясь в папке проекта, выполните в терминале команду `ln -s config/production.rb config/migration.rb`. О том, что такое симлинк (symlink) можно почитать в википедии.

Что следует сделать сразу после создания?

Обновление файла .gitignore

Файл `.gitignore` [1], который находится в корневой папке вашего приложения говорит git-у какие файлы и папки нужно исключить из списка добавляемых в репозиторий. То есть, эти файлы останутся исключительно на вашей локальной машине и если git-репозиторий (который мы, в скором времени, создадим) будет склонирован другим разработчиком, то этих файлов он там не обнаружит.

Это бывает необходимо для файлов, которые хранят какие-либо локальные настройки или для файлов, которые пользователи вашего приложения загружают или генерируют. Ниже я приведу содержимое моего файла `.gitignore` (он немного отличается от того, что генерирует Rails) с пояснениями.

```
/.bundle      # Папка использующаяся гемом bundler
db/*.sqlite3  # Файлы базы данных sqlite (нужно, если
              # в development-среде вы используете sqlite - обычно
              так и есть)
.sass-cache/  # Папка, использующаяся гемом sass

tmp/*         # Временные файлы, на каждой машине должны быть свои
!tmp          # Восклицательный знак означает оставить саму папку
              # (но предыдущая команда проигнорирует файлы в ней)
log/*.log     # log-файлы тоже на каждой машине - свои
!log

# Обычно *.yml файлы в папке config/ содержат настройки
# приложения, специфичные для каждой машины.
# Например, пароли и другие данные для доступа
# к БД или различным API. По соображениям удобства, а также по
# соображениям безопасности, эти файлы не должны быть
# добавлены в репозиторий.
config/*.yml

# В эту папку будут компилироваться все asset-ы (обычно в production-среде)

public/assets/

# Файл, который обычно создается только в production-среде
```

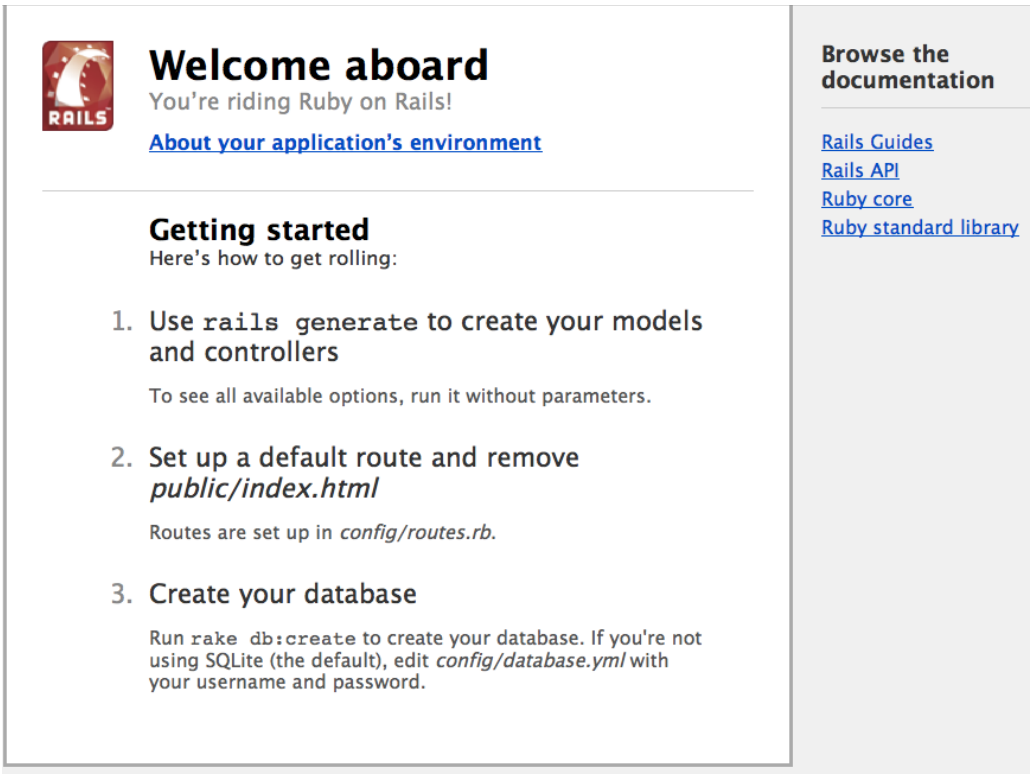
```
# на сервере. Говорит nginx-у, что ваше приложение на данный
# момент не работает и показывает пользователям временную
# страницу с сообщением о неисправности
# (сама страница содержится в этом файле).
public/maintenance.html

# В эту папку пользователи загружают изображения
# (если так настроить приложение, конечно).
# На локальной машине в роли пользователя выступаете
# только вы, на сервере - все посетители вашего сайта.
# Следовательно, содержимое этих папок должно отличаться. Поэтому
git должен их игнорировать.
!public/images/
public/images/*
public/images/**/*  # выражение /**/* означает все файлы в поддиректориях


.DS_Store           # Системный файл MacOSX, мешается под ногами.

# Пустая папка, в которой нет ничего, кроме этого
# файла будет все равно добавлена в git-репозиторий.
# Полезный файл, не будем его игнорировать
# (следовательно, восклицательный знак в начале).
!.gitkeep
```

Проверка database.yml

Перед тем как вы запустите приложение, вам необходимо проверить файл `config/database.yml`. В обычном случае, вы пока что будете использовать БД sqlite и никаких изменений в этот файл вносить не нужно. В дальнейшем, в вашей карьере вам придется использовать на локальной машине ту же СУБД, что и на сервере — например PostgreSQL. В этом случае, вам будет необходимо сначала настроить эту БД и создать соответствующих пользователей и базы данных, а затем ввести данные для доступа в файл `database.yml`

Тестовый запуск приложения

Настало время в первый раз запустить наше приложение. Сделать это можно набрав в терминале команду `rails s` (я предпрочитаю держать в терминале два таба, в одном из которых запущено rails-приложение, а в другом я могу набирать команды). После этого вы можете перейти в браузер и набрать следующий адрес: `http://localhost:3000`. В браузере должно появиться примерно следующее:



Если вы увидели то же самое, то все в порядке и ваше приложение успешно запустилось. Конечно же, в дальнейшем, с помощью routes, у вас будет возможность объяснить Rails-приложению, что нужно показывать на главной странице. Но не забудьте удалить файл `public/index.html` — это как раз тот html-файл, который сейчас загружен в ваш браузер. Если вы оставите его там, то эта страница будет грузиться в качестве главной всегда. Удалив этот файл и обновив страницу в браузере, вы скорее всего увидите вашу первую ошибку — не беда, так и должно быть. После того, как мы познакомимся с Rails routes в главе Контроллеры и ресурсы, мы сможем ее исправить.

Создание git-репозитория и коммит

После того как мы сгенировали приложение, проверили его настройки и убедились, что оно запускается — можно создавать git-репозиторий и делать первый коммит. Вот последовательность команд[1], которые необходимо выполнить в терминале, находясь в папке проекта:

```
git init   # создаем новый репозиторий в текущей папке
git add .  # добавляем в индекс репозитория все файлы
           # (кроме тех, что указаны в .gitignore)

# создаем первый коммит с сообщением "Initial commit"
git commit -m "Initial commit"
```

Отлично. Можем считать, что первый шаг навстречу миру Ruby On Rails сделан.

[1] Эти команды уже обсуждались в разделе Системы контроля версий главы Инструменты, которые вам нужны, чтобы начать работу.

[1] Подробное руководство по использованию файла .gitignore

Глава 5. Модели, ActiveRecord и взаимодействие с БД

Основы моделей в Rails

Мы уже касались понятия MVC в разделе Архитектура MVC главы Преждем чем начать: знания, которые вам понадобятся. Здесь я постараюсь подробнее раскрыть тему моделей и на примерах показать для чего создается модель в Rails. В целом, можно сказать, что разработка Rails-приложения начинается с моделей, а вернее с решения о том, что должно стать моделью в вашем приложении. Модели — это мозг вашей программы без которого программа не сможет выполнять никаких полезных функций.

Как решить, что должно стать моделью?

Рассмотрим типичный интернет-магазин. Какой функционал в нем должен присуствовать? Пользователи должны иметь возможность просматривать категории и товары в них, добавлять товары в корзину, оформлять заказы и оставлять отзывы о конкретных товарах. Чтобы решить, какие модели нужны в нашем Rails-приложении, необходимо внимательно посмотреть на предыдущее предложение и выделить в нем существительные, имеющие отношение к предметной области: **пользователи**, **категории**, **товары**, **корзина**, **заказы** и **отзывы**. Именно для этих понятий в нашем приложении и будут существовать модели и, в дальнейшем, они будут уметь взаимодействовать друг с другом. Можно думать о приложении как о real-time стратегии, где модели представляют из себя различные юниты, появляющиеся и исчезающие на карте сражений.

При разработке приложения, вам не следует создавать все возможные модели сразу и мучаться, не забыли ли вы какую-то другую модель. Напротив, сначала нужно ограничиться минимальным количеством моделей и постепенно, по мере продвижения разработки, добавлять новые. Идеально сначала добавить только одну единственную модель — ту, которая, по вашему мнению, является ключевой для вашего приложения — и начать заполнять ее кодом. Очевидно, что в нашем случае такой моделью является Товар. После добавления этой модели в наше приложение мы начнем постепенно писать ее функционал — валидации, коллбэки, методы и другие части кода.

Хотелось бы отметить, что когда я говорю о постепенном добавлении моделей в приложение, я вовсе не имею в виду, что вы будете выкладывать это приложение на продакшен в неготовом виде (хотя и такой подход тоже возможен). Предположим, ваш

клиент дает вам срок разработки интернет-магазина — 1 месяц. Тогда вы можете запланировать, что в первые две недели вы добавите модели Товар, Категории и Пользователи; в следующие две недели вы запрограммируете функционал для моделей Корзина и Заказы; и только в конце месяца займетесь отзывами.

Как правильно назвать модель?

Название моделей в Rails должны представлять из себя существительные, или словосочетания заканчивающиеся на существительные — естественно на английском языке и обязательно в единственном числе. Таким образом, для наших слов, которые мы "вытащили" из предложения описывающего функционал интернет-магазина, мы могли бы создать следующие модели: User (пользователь), Category (категория), Item (товар), Cart (корзина), Order (заказ), Review (отзыв). В дальнейшем, в нашем магазине могут произойти изменения и нам понадобится разделять товары на физические и виртуальные — в этом случае, по соглашению именования классов в Ruby, мы бы добавим две модели с именами `RealItem` и `VirtualItem`, т.е имя каждой модели будет состоять из слов в *CamelCase*[1].

Не называйте модели прилагательными, глаголами или другими частями речи. И особенно важно помнить не называть модели существительными во множественном числе. Кроме того не стоит использовать в именах предлоги. Ниже я приведу несколько примеров неверного именования моделей и рядом покажу как было бы правильно назвать модель:

Примеры именования моделей в Rails

Неверный вариант	**Возможный преемлемый вариант**
Users	`User`
`ItemThatIsReal`	`RealItem`
`Virtualitem`	`VirtualItem`
`Ordering`	`Order`

Как создать модель?

Выполните в терминале команду:

```
rails g model [model_name]
```

Естественно, вместо [model_name] необходимо подставить название вашей модели. Причем в т.н. *snake case*[2]. Например, вот две команды, которые создадут модели `Item` и `VirtualItem` соответственно:

```
rails g model item
rails g model virtual_item
```

Каждая из команд запустит генератор, который создаст два файла: файл с миграциями (на него мы посмотрим в разделе Миграции, атрибуты в моделях и что такое ORM) и файл с моделью. В нашем случае, когда мы создаем модель Item, это будет файл `app/models/item.rb`. Открыв этот файл, можно будет увидеть следующую картину:

```
class Item < ActiveRecord::Base
end
```

Как видите, в файле ничего нет, кроме этих двух строк (возможно будет еще закомментированная строка, начинающаяся на `attr_accessible` — к этому мы вернемся позже), то есть генератор не сделал за нас слишком много работы. Это хорошо, потому что, как вы помните, нам необходимо понимать *каждую* строку кода в нашем приложении.

Класс `ActiveRecord::Base`, от которого наследуется наш класс `Item` уже содержит в себе массу методов, которые окажутся полезны при разработке нашего приложения. То есть, наша модель — вовсе не пустой класс, объекты которого ничего не умеют делать. Каждый экземпляр Item будет довольно серьезным объектом с массой полезных методов. Однако, прежде чем наша модель заработает, нам придется написать для нее миграцию.

[1] CamelCase — способ именования сущностей (в Ruby так именуются классы), при котором каждое слово в имени записывается с большой буквы и слова при этом не разделяются пробелами. Свое название способ получил из-за отдаленного сходства с силуэтом верблюда.

[2] snake_case — способ именования сущностей (в Ruby так именуются методы и переменные), при котором все символы в каждом слове (в том числе первые) пишутся символами нижнего регистра, а сами слова разделяются символами нижнего подчеркивания _. Свое название способ получил из-за отдаленного сходства со змеей.

Миграции, атрибуты в моделях и что такое ORM

Атрибуты в моделях

После того, как мы создали модель, необходимо решить, какие *поля* должны быть у каждой модели. Под полями далее я буду понимать различные категории данных, которые будет содержать в себе каждая создаваемая модель. Поля также называют *атрибутами* (не вдаваясь в подробности, будем считать, что в данном случае это синонимы) и свойствами (чем отличается этот термин мы посмотрим чуть ниже).

Как правильно понять, какие данные нам нужны в моделе и как назвать атрибуты для

этих данных? Задайте себе вопрос "что есть у моей модели?" — например, у товара есть название, подробное описание, цена и вес. Соответственно, наши атрибуты будут называться `title`, `description`, `price` и `weight`.

Атрибуты, которые есть у модели ActiveRecord определяются в т.н. файле миграции. Именно его мы будем заполнять первым делом после того, как сгенерируем модель. Но остается вопрос: что такое миграции и зачем они нужны?

Object Relational Mapping

Любой товар, который мы добавляем в магазин, по сути, представляет из себя набор данных. А данные нужно где-то хранить. По волшебству добавленный товар не станет доступным каждому посетителю магазина — как только программа выполнится (т.е. завершится запрос пользователя к серверу и сервер вернет в браузер пользователю страницу) все объекты созданные в памяти компьютера, в т.ч. объекты класса `Item`, пропадут. Чтобы этого не произошло, данные содержащиеся в нашем объекте можно записать в базу данных. Теоретически, мы могли бы сделать это вручную — написав SQL запрос на вставку данных в базу. Однако с практической точки зрения нам понадобится хорошо знать диалект SQL, который используется нашей БД и также постоянно думать о потенциальных неприятностях типа SQL-инъекций[1]. Поэтому фреймворк Rails и класс ActiveRecord (от которого, напомню, наследуются все наши модели) с радостью сделают это за нас.

И сделают они это с помощью т.н. Object Relational Mapping. Дело в том, что каждой модели в нашем приложении соответствует таблица в базе данных. В этой таблице каждый столбец будет соответствовать определенному атрибуту в модели. Например, для нашей модели Item в базе данных будет существовать таблица `items` с полями `title`, `description`, `price` и `weight`. Каждый раз, когда мы будем создавать новую модель, в таблице `items` базы данных будет появляться новая строка, где каждая ячейка будет заполнена данными из модели, которые соответствуют названию столбца.

Таким образом, когда мы будем просматривать каталог товаров, Rails-приложение найдет в таблице `items` в БД все записи, далее создаст для каждой записи объект класса `Item` и поместит в его атрибуты данные, полученные из базы. И все это произойдет автоматически, благодаря библиотеке ActiveRecord.

Теперь наша задача — объяснить Rails какие атрибуты должны быть у моделей и таблицу с какими столбцами необходимо создать в БД.

Создание миграции для модели

Миграции — это файлы, которые находятся в папке `db/migrations`. Каждый такой файл представляет из себя инструкцию на языке Ruby, благодаря которой Rails-приложение знает, какие таблицы с какими полями нужно создать в базе данных. Только что сгенерированная пустая миграция для модели `Item` будет выглядеть следующим образом:

```
class CreateItems < ActiveRecord::Migration

  def change
    create_table :items do |t|
      t.timestamps
    end
  end

end
```

`t.timestamps` автоматически добавляет в таблицу `name два столбца: created_at и updated_at`. Эти два поля будут автоматически обновляться Rails-приложением каждый раз, когда вы будете сохранять свою модель и поэтому там всегда будут содержаться время создания и обновления модели соответственно.

Чтобы добавить поле в таблицу (и, соответственно, атрибут в модель `Item`), вам необходимо воспользоваться специальным объектом `t`, который доступен внутри блока, передающегося методу`#create_table`. Этот объект имеет набор методов, которые соответствуют типам данных в вашей БД. Например, чтобы создать в таблице БД поле типа "строка" для имени товара, нужно добавить следующий код:

```
t.string :name
```

Конечно же, не все поля будут строчного типа. Например цена должна быть числом с плавающей точкой. Следовательно, мы напишем следующий код для создания поля `price`:

```
t.float :price
```

У объекта `t` доступны следующие методы, соответствующие типам данных в БД: `#binary`, `#boolean`, `#date`, `#datetime`, `#decimal`, `#float`, `#integer`, `#primary_key`, `#string`, `#text`, `#time`, `#timestamp`. Подробнее о каждом типе данных можно почитать в разделе migrations в Rails-guides [1]

Стоит отметить, что типы данных в БД и типы данных в Ruby — не одно и то же. Например, как вы могли заметить, в миграциях есть два типа данных различие между которыми не очевидно: `string` и `text`. И действительно, когда вы будете читать атрибут из вашей модели, в Ruby значением и того и другого атрибута будет объект класса `String`. Но с точки зрения БД между этими двумя типами данных есть различия: в большинстве БД поддерживающих SQL, тип данных `string` позволяет хранить строки не длиннее 256 символов, в то время как тип `text` предназначен для очень длинных строк (у него тоже есть

оргранничения, но они будут зависеть от использующегося типа БД).

Теперь мы можем полностью дописать нашу миграцию для модели Item и она будет выглядеть примерно так:

```
class CreateItems < ActiveRecord::Migration

  def change
    create_table :items do |t|
      t.string :name
      t.text   :description
      t.float  :price
      t.float  :weight
      t.timestamps
    end
  end

end
```

После того как миграция написана, нужно ее "прогнать" — запустить программу, которая прочитает инструкции написанные в миграции и применит их к базе данных. Чтобы это сделать, перейдите в терминале в папку с вашим приложением и запустите следующую команду:

```
rake db:migrate
```

и ваша миграция применится к базе данных. Теперь мы можем проверить, все ли работает так, как мы ожидаем и будут ли у нашей модели соответствующие атрибуты.

У методов миграций — помимо имени столбца — есть различные опции, которые им можно передавать. Например можно установить значение по умолчанию или набор неприемлимых значений для поля. Их мы рассмотрим в разделе Миграции, часть 2: добавление столбцов и индексов, а сейчас настало время вживую посмотреть, как же работает модель.

Чем свойства отличаются от атрибутов?

Строго говоря, свойство любого объекта в Ruby — это инстансная переменная, которая хранит некое значение на протяжении всей или части жизни объекта. Например, можно было бы создать такой объект:

```
class Item < ActiveRecord::Base

  after_initialize do
    @made_in = "China"
  end

end
```

Формально у всех объектов класса `Item` теперь есть свойство `made_in`, однако доступ извне к нему получить невозможно, потому что у объекта нет геттера и сеттера. Добавив одну строку в наш класс мы сможем исправить эту проблему:

```
class Item < ActiveRecord::Base

  attr_accessor :made_in # обратите внимание, это не метод attr_accessible!


  after_initialize do
    @made_in = "China"
  end

end
```

(Мы использовали коллбэк `after_initialize` вместо привычного конструктора `def initialize`, потому что у моделей ActiveRecord есть свой собственный конструктор, который нельзя переопределять). Теперь мы можем устанавливать и читать свойства объектов класса `Item` следующим образом:

```
item1 = Item.new
item1.made_in = "USA"
item1.made_in # => "USA"
```

Написанный нами код, однако, никаким образом не сохранит значение свойства `made_in` в базу данных, потому что в миграции не было определено соответствующее поле для таблицы `items` в БД. Таким образом, атрибутом модели называют такое свойство, для которого существует столбец в соответствующей таблице в БД и о котором Rails знает через миграции. Естественно, если мы задаем атрибут модели в миграциях, то код выше нам уже не нужен — ActiveRecord автоматически добавит геттеры и сеттеры для этого атрибута в модель.

[1] На самом деле, разбираться в SQL и уметь писать запросы — полезный скилл, который каждому серьезному разработчику веб-приложения придется прокачивать в любом случае. Но вам не нужно делать это прямо сейчас.

[2] Если вы не знаете что такое таблицы в реляционных базах данных, то можно для начала думать о них, как о таблицах в программе Excel, с тем лишь исключением, что столбцы в них называются не A, B, C..., а конкретными именами.

[1] Поддерживаемые миграциями типы данны, Rails guides.

Сохранение и получение данных из БД - проверяем работоспособность модели

Чтобы проверить работоспособность нашей модели, нам придется запустить rails-консоль и писать наш код там. В дальнейшем, код, который сейчас мы будем выполнять в rails-консоли, мы увидим в наших контроллерах. Работа с rails-консолью — это инструмент, который применяется относительно нечасто. Тем не менее, поскольку пока что у нас отсутствует какой-либо другой способ проверить модель (у нас нет ни контроллеров ни views, ни даже unit-тестов для модели), мы перейдем в терминал и воспользуемся командой `rails c` — это запустит irb с загруженной в него средой Rails-приложения. То есть нам будут доступны все классы из нашего Rails-приложения (в т.ч. класс `Item`).

Проверить работоспособность модели можно создав объект, например таким способом:

```
item1 = Item.new
```

Теперь в переменной `item1` будет содержаться новый объект класса `Item`. Мы можем проверить, существуют ли в этом объекте указанные нами в миграции атрибуты, попытавшись установить их с помощью сеттеров:

```
item1.name  = "Porsche"
item1.description = "A really fast car"
item1.price = 3000000
```

Однако пока что для этого объекта не существует записи в базе данных. Если сейчас набрать команду `exit` и выйти из терминала, то объект будет потерян навсегда. Чтобы этого не произошло, нужно вызвать на объекте метод `#save`, который и создаст такую запись в БД, т.е. "сохранит модель":

```
item1.save
```

Теперь вы можете набрать команду `exit`, затем, снова оказавшись в шелле[1], набрать `rails c` и после этого ввести в Rails-консоль следующий код:

```
item1 = Item.last
```

Метод `.last` вернет последний объект класса `Item`, запись для которого была сохранена в БД — то есть тот самый объект, который мы создали, когда первый раз запускали Rails-консоль.

Чтобы задавать атрибуты моделей вовсе необязательно использовать сеттеры. Можно передать хэш с атрибутами прямо в метод `.new`, например вот так:

```
Item.new(name: "Porsche", description: "A really fast car", price:
3000000)
```

Важное замечание: чтобы такой код не вызвал ошибки, вам необходимо перечислить все эти атрибуты в качестве аргументов метода `attr_accessible`, который уже вызван в вашем

вновь созданном классе Item (помните я упоминал единственную закомментированную строку в пустом классе? Это она). Вам необходимо будет расскомментировать строку, где вызывается метод `attr_accessible` и написать следующее:

```
attr_accessible :name, :description, :price, :weight
```

Этот код говорит нашей модели, что ей позволено принимать значения для атрибутов через хэш в `.new`. Если бы строка была закомментирована, при попытке это сделать у нас возникла бы ошибка (проверьте и посмотрите что произойдет). Это нужно из соображений безопасности [1], чтобы злоумышленники не смогли изменить важные атрибуты ваших моделей.[2]

Однако даже если передать методу `.new` хэш с атрибутами, запись не будет автоматически сохранена в базу данных. Нам все равно придется вызывать метод `#save`. Чтобы убить двух зайцев сразу, можно вместо метода

```
.new
```

воспользоваться методом `.create`:

```
Item.create(name: "Porsche", description: "A really fast car",
price: 3000000)
```

Этот код установит значения для атрибутов модели `Item` и сразу сохранит соответствующую запись в БД.

Очень похож на метод `.create` метод `#update_attributes` (обратите внимание: первый — это метод класса, второй — метод, доступный на объектах этого класса). Он позволяет изменять значения атрибутов уже существующей модели следующим образом:

```
item1 = Item.create(name: "Porsche", description: "A really fast
car", price: 3000000)
item1.update_attributes(price: 4000000)
```

Вторая строка этого кода обновит запись в базе данных и изменит значение цены в соответсвующем столбце для этой записи. Естественно, новая цена сразу же будет доступна через геттер этого атрибута:

```
item1.price # => 4000000
```

Наконец, еще один важный метод — `#destroy`. Он позволяет удалить из БД запись, соответствующую этому объекту:

```
item1.destroy
```

Вот все основные методы, которые нам понадобятся для начала, чтобы эффективно работать с моделями.

[1] Под *шеллом* я имею в виду оболочку операционной системы. Грубо говоря, вашу привычную консоль, где вы можете набирать UNIX-команды.

[2] Мы подробнее вернемся к этому в главе Views, хэлперы и ассеты

[1] Подробнее о том, что такое attr_accessible в моделях ActiveRecord.

Валидации

Перед тем, как данные будут сохранены в базу данных, вы, скорее всего, захотите убедиться, что эти данные правильные. Например, что email похож на email, что в номере телефона содержатся только цифры и символы-разделители, что название товара не превышает, например, 100 символов и что цена товара (который администратор может добавлять через форму на сайте) является числом больше нуля. В случае, если модель содержит ошибки, мы хотели бы вновь показать пользователю форму, в которой эти ошибки отмечены — за это решение будет отвечать уже контроллер. Но во всех моделях ActiveRecord существует способ узнать, прошла ли модель валидации или в ней есть ошибки.

Стандартные валидации ActiveRecord

Рассмотрим уже знакомое нам создание модели Item:

```
item = Item.new(name: "Car", description: "super fast car", price: 0)
```

Как видите, мы сейчас намеренно указываем цену равную нулю. **Обычно валидации "прогоняются" на объекте перед каждым сохранением его в базу данных** (то есть когда мы вызываем метод `#save`, но до фактического запроса в БД). Однако, если попытаться сохранить модель, пока никаких валидаций у нас не создано, то проблем не возникнет. Более того, мы можем убедиться в отсутствии ошибок с помощью метода `#errors.empty?`:

```
item.save           #=> true
item.errors.empty?  #=> true
```

Разумеется, нас это не устраивает. Нельзя позволять сохранять товар с неправильной ценой в базу данных. Поэтому мы добавим в модель валидацию.

Стандартные валидации в ActiveRecord — это методы класса, которые мы должны

вызвать в контексте самого класса.[1] Рассмотрим валидацию, которая будет проверять, что значение в поле цена является числом больше 0:

```
class Item < ActiveRecord::Base
  ...
  validates :price, numericality: { greater_than: 0 }
  ...
end
```

Подробно разберем на части это выражение:

- `validates` является в данном случае названием специального метода класса, доступного всем моделям ActiveRecord. В зависимости от передаваемых ему аргументов, создается та или иная валидация.
- `:price` — это первый аргумент передаваемый в метод, который является символом и который говорит валидации какой атрибут нужно проверить.
- `numericality: { greater_than: 0 }` целиком является вторым аргументом. Это хэш[2], ключи которого являются названиями типов валидаций, которые будут применены к полю — в данном случае мы добавили валидацию, которая называется `numericality`.
- В качестве значения для ключа `numericality` мы передаем еще один хэш, который представляет из себя инструкции для данной валидации. В данном случае мы просим валидацию `numericality` проверить, что значение поля `price` больше нуля.

Мы можем указать не одно, а сразу несколько полей к которым будет применяться та или иная валидация. Например, та же самая валидация подойдет и для поля `weight`, а значит мы можем написать ее следующим образом:

```
class Item < ActiveRecord::Base
  ...
  validates :price, :weight, numericality: { greater_than: 0 }
  ...
end
```

Как видите, в данном случае хэш с описанием типов валидаций, которые нужно применить к перечисленным полям, сместился и стал не вторым, а третьим аргументом, передаваемым в метод `.validates`.

Наконец, мы можем указать несколько типов валидаций для тех же самых полей. Для этого достаточно добавить нужный ключ и значение для него в хэш, идущий последним аргументом. Например, мы могли бы дополнительно убедиться, что поля не только будут содержать числа больше нуля, но также не будут содержать значение `nil`:

```
class Item < ActiveRecord::Base
  ...
  validates :price, :weight, numericality: { greater_than: 0 },
  presence: true
  ...
end
```

Другое дело, что это было вовсе не обязательно, т.к. валидация `numericality` проверяет наличие `nil` автоматически и добавляет ошибку, если поле пустое. Так, однако, бывает далеко не всегда и валидация `presence` бывает очень полезной в некоторых случаях. Обратите внимание, что пара ключ-значение `presence: true` не является еще одним аргументом метода `validates`, а только лишь новым элементом хэша, передаваемого этому методу в качестве последнего аргумента. Поэтому, мы могли бы написать валидацию таким образом:

```
validates :price, :weight, { numericality: { greater_than: 0 },
presence: true }
```

Теперь если мы попробуем сохранить наш объект в базу данных, нам это не удастся:

```
item.save          #=> false
item.errors.empty? #=> false
```

Также, вы можете воспользоваться методом `p`, чтобы вывести содержимое объекта `item.errors` и посмотреть на то, какие ошибки валидации были добавлены в объект.

Помимо того, что мы можем указать методу `.validates`, какие валидации необходимо применить к полям, мы также можем сказать ему, когда нужно это делать. Дело в том, что валидации можно выполнять при создании объекта (т.е. когда записи для этого объекта еще нет в БД и мы в первый раз вызываем на объекте метод `#save` или создаем объект методом `.create`) и при обновлении объекта (если мы изменяем поля в существующем объекте сеттерами и затем сохраняем его или выполняем обе операции сразу методом `#update_attributes`). Рассмотрим вот такой пример:

```
class Item < ActiveRecord::Base
  ...
  validates :price,  numericality: { greater_than: 0 }
  validates :weight, numericality: { greater_than: 0 }, on: :create
  ...
end
```

В очередной раз обратите внимание, что пара ключ-значение `on: create` присоединяется к хэшу, идущему последним аргументом к методу `validates`. Теперь вы можете попробовать набрать в rails-консоли следующий код и посмотреть на то, какие ошибки будут добавлены в модель:

```
item1 = Item.create(price: 0)
#=> false, потому что валидации на поля price и weight не пройдут

item2 = Item.create(price: 1, weight: 1)
#=> true

item2.update_attributes(price: 0)
#=> false, валидация на price не пройдет

item3 = Item.create(price: 1, weight: 1)
item3.update_attributes(weight: 0)
#=> true, потому что валидация для поля weight игнорируется

item3.update_attributes(price: 0)
#=> false, потому что валидация для поля price не пройдет
```

Сообщения об ошибках

Естественно, после того, как вы "прогнали" валидации, вы не только можете посмотреть возникли-ли какие-то ошибки, но и узнать конкретно, какие именно ошибки возникли для каждого из полей. Для этого существует метод `#errors`, который возвращает специальный объект класса ActiveModel::Errors (очень похожий на хэш), который и содержит всю необходимую нам информацию. Поэтому, чтобы узнать какие ошибки возникли для поля `price`, нам следует выполнить следующий код:

```
item1.errors[:price]
```

результатом выполнения этого выражения в нашем случае (учитывая валидации написанные выше) станет следующий массив:

```
["is not a number", "can't be blank"]
```

Из него видно, что наше поле имеет две ошибки, каждая из которых является результатом выполнения привязанной к полю валидации. Первый элемент массива является результатом выполнения валидации `numericality: { greater_than: 0 }`, а второй — `presence: true`. Как уже было упомянуто выше, первая валидация автоматически выполняет и вторую, так что, возможно, вторая нам не так уж и нужна.

Теперь, зная как получить доступ к списку ошибок валидаций каждой модели, мы сможем выводить все эти сообщения в форме, которая будет повторно загружаться пользователю в случае неправильного ввода данных. Об этом разговор пойдет в главе Контроллеры и ресурсы

Полагаю, что вы обратили внимание на то, что ошибки выведены на английском языке. Это необязательно должно быть так. Для вашего приложения вы можете выбрать дефолтную русскую *локаль*[4], к которой будут привязаны файлы с переводами различных частей

приложения — в том числе ошибок ActiveRecord. За это отвечает модуль интернационализации Rails, который называется I18n[5]. Для некоторых частей Rails на Github-е можно найти готовые переводы, но вам безусловно придется потрудиться и подготовить собственные файлы локализации для тех частей приложения, которые были написаны лично вами. Не "впаивайте" русские символы в код ваших моделей и контроллеров, пользуйтесь модулем интернационализации Rails - I18n [3]

Вот список всех типов валидаций (с примерами использования), которые доступны через метод `validates` ActiveRecord-а:

- **acceptance**

```
validates :i_agree, acceptance: true
```

- **validates_associated** — используется когда нужно при сохранении модели свалидировать не только саму модель, но и связанную с ней другую модель (подробнее об ассоциациях читайте в разделе Ассоциации: связи с другими моделями)

```
has_many :images
validates_associated :images
```

- **confirmation** — если в форме у вас есть два поля, одно из которых называется `email`, а второе `email_confirmation`, то следующая валидация проверит, что значения в этих полях — идентичные:

```
validates :email, confirmation: true
```

- **exclusion** — используется, когда вы хотите, чтобы поле не могло принимать определенное значение.

```
validates :login, exclusion: { in: %w(admin administrator),
          message: "Это имя пользователя зарезервировано" }
```

- **format** — проверка значения в поле *регулярным выражением*[3].

```
validates :login, format: { with: /\A[a-zA-Z0-9]+\z/ ,
          message: "может содерждать только буквы и цифры" }
```

- **inclusion** — позволяет проверить, что значение поля может быть только одним из возможных вариантов.

```
validates :size, inclusion: { in: %w(small medium large),
          message: "%{value} не является правильным размером" }
```

- **length** — проверяет длину поля (если тип данных, который в нем содержится, является строкой)

```
validates :login, length: { minimum: 2, maximum: 30 }
```

- **numericality** — проверяет что число, содержащееся в поле, удовлетворяет условиям.

```
validates :age, numericality: { greater_than: 17, less_than: 100 }
```

- **presence** — проверяет, что значение поля не является `nil`.

```
validates :email, presence: true
```

- **uniqueness** — перед тем как сохранить данные делает запрос в БД и проверяет, нет ли в базе записи, где указанное поле содержит такое же значение.

```
validates: :email, uniqueness: true
```

Посмотреть более подробные примеры различных типов валидаций (официально это называется *validation helpers*) вы можете на сайте Rails Guides в разделе Валидации [2].

[1] Из курса языка Ruby вы должны помнить, что контекстом класса — очень грубо говоря — называется пространство между ключевыми словами `class` и `end`, исключая пространство, которое принадлежит определению инстансных методов — то есть пространство между ключевыми словами `def` и `end`.

[2] Напомню, что в данном случае интерпретатор руби понимает, что это хэш даже не смотря на то, что его не окружают фигурные скобки. Если бы мы хотели подчеркнуть, что второй аргумент является хэшем, мы могли бы написать так: `validates :price, { numericality: { greater_than: 0 }}`, то есть мы окружили бы его фигурными скобками.

[3] Регулярные выражения — обширная тема, имеющая отношения не только к языку Ruby. Введение в регулярные выражения обычно дается в любой книге по Ruby, но если вы вдруг найдете время, я рекомендую вам прочитать прекрасную книгу, целиком посвященную этой увлекательной теме: Книга "Регулярные выражения" (Mastering Regular Expressions) на русском.

[4] Локаль — термин использующийся для обозначения совокупности языковых настроек, которую в данный момент использует приложение. Обычно подразумевается, что локаль приложения может быть изменена без каких либо сложностей и дополнительного внимания со стороны программиста: например если пользователь заходит на сайт с ip в Российской Федерации, то мы устанавливаем локаль в "ru" и он видит руссифицированное приложение (для этого сценария, разумеется, программисту потребуется *предварительно* написать соотвествующий код, проверяюший ip-адрес и назначающий локаль для приложения).

[5] Рассказ об интернационализации в Rails выходит за рамки этой книги, но т.к. сама тема не слишком сложная, с ней успешно можно ознакомиться, используя документацию.

[1] Книга "Регулярные выражения" (Mastering Regular Expressions) на русском.

[2] Описание различных типов стандартных валидаций ActiveRecord на сайте Rails Guides.

[3] Модуль интернационализации Rails I18n

Ассоциации: связи с другими моделями

Данные, которые ваше приложение будет сохранять в БД часто будут между собой связаны. Например Товары должны принадлежать конкретным Категориям, у каждого Пользователя должна быть одна Корзина со своим набором Товаров, и также у Пользователей могут быть Заказы, содержащие Товары. Реляционные базы данных решают этот вопрос за счет того, что каждая таблица может быть связана с другой таблицей так называемым полем *foreign id*. Например чтобы однозначно установить, что данный товар принадлежит Категории с id 7, в записи в таблице items в БД должно быть специальное поле под названием `category_id`.

Вот как это может выглядеть в базе данных. Таблица `categories`:

id	name
7	Мягкие игрушки

Таблица items:

id	name	description	price	weight	category_id
542	Плюшевый слон	Голубого цвета	499	0,2	7

Теперь, мы однозначно можем сказать, что "Плюшевый слон" находится в категории "Мягкие игрушки."

Чтобы Rails автоматически вставляла нужные foreign id в соответствующие записи в таблице, нам придется проделать небольшую работу и написать немного кода в моделях, чтобы объяснить приложению, как именно и между какими моделями существует ассоциативная связь.

Существует несколько типов ассоциативных связей, которые поддерживаются в Rails. Для каждого типа ассоциаций я сначала приведу пример ситуации, когда данная связь нам может понадобиться, а затем покажу как данный тип связи реализуется в Rails.

has_one / belongs_to

Самый простой тип ассоциации легко демонстрируется на следующем примере: в нашем магазине у каждого пользователя должна быть одна и только одна корзина, в которую он будет складывать свои товары. Очевидно, для этого нам прежде всего потребуются модели `User` и `Cart`, которые можно сгенерировать уже знакомой нам командой:

```
rails g model user
rails g model cart
```

Прежде чем написать какой-либо код в нашей модели, давайте для начала разберемся в одном теоритическом вопросе, а именно следует спросить себя: принадлежит ли корзина пользователю или пользователь — корзине? (в нашем случае ответ более менее очевиден, но будьте осторожны — логика стоящей перед вами задачи не всегда однозначно отвечает на этот вопрос). Определившись с ответом — надеюсь вы тоже решили, что Корзина принадлежит Пользователю — мы можем открыть модель `User` и в контексте класса написать там следующий код:

```
class User < ActiveRecord::Base

  ...
  has_one :cart

end
```

Теперь нужно открыть модель `Cart` и написать там следующее:

```
class Cart < ActiveRecord::Base

  ...
  belongs_to :user

end
```

Две строки кода, которые мы добавили в наши модели, сигнализируют Rails о том, что теперь эти модели связаны ассоциативно.

Второе, что нам необходимо сделать, чтобы наша связь заработала — добавить поле с foreign id в миграцию. Для этого нужно открыть миграцию для той модели, которая *принадлежит*, то есть для той модели, где указано `belongs_to`. В нашем случае, это, разумеется, будет миграция для модели `Cart`:

```
class CreateCarts < ActiveRecord::Migration
  def change
    create_table :carts do |t|
      t.integer :user_id
    end
  end
end
```

[1] В примере выше мы добавили поле типа Integer (потому что все id в моделях ActiveRecord являются целыми числами) — оно будет хранить id пользователя, которому принадлежит корзина. Обратите внимание, что название данного поля — `user_id` — является обязательным. Если его назвать иначе, ассоциация не заработает. Ну и не забудьте добавить несколько полей в миграцию для таблицы `users`, например `login` и `email`.

В качестве альтернативного варианта, мы могли бы написать в миграции `t.references :user` — данная запись является просто удобным псевдонимом к тому варианту, который приведен выше. В любом случае, после того как вы прогоните миграцию, в базе данных появится таблица `carts` с полем `user_id`. Собственно, именно это вам на данный момент и следует сделать — запустить команду `rake db:migrate`.

Использование ассоциации

После того, как мы прогнали миграции — работа по созданию ассоциации "Пользователь — Корзина" в нашем приложении закончена и ей можно начинать пользоваться. Мы, естественно, будем на данный момент проверять ее работоспособность в rails-консоли.

Ассоциация has_one/belongs_to добавила в наши модели по одному важному методу, названия которых соответствуют названиям моделей. Во всех объектах класса `User` появился метод `#cart`, а во всех объектах класса `Cart` появился метод `#user`. Чтобы продемонстировать это, нам сначала нужно создать модели:

```
user = User.new(login: "user1")
cart = Cart.new
```

Если сейчас попробовать вызвать эти методы на созданных объектах, то мы получим `nil`:

```
user.cart #=> nil
cart.user #=> nil
```

потому что пока что мы никак не указали нашему приложению, что данная корзина принадлежит данному пользователю. Указать это можно используя соответствующий сеттер:

```
user.cart = cart
```

Теперь нашему приложению известно, что только что созданная нами корзина присвоена пользователю. Правда, в базу данных пока что ничего сохранено не было. Чтобы это сделать, нам достаточно вызвать уже знакомый нам метод `#save`:

```
user.save
```

Это сохранит в базу данных запись как о пользователе, так и о созданной для него корзине. Мы могли бы поступить и наоборот, установив пользователя для корзины и затем сохранив его:

```
cart.user = user
cart.save
```

Эффект был бы тот же.

Устанавливать ассоциации можно не только с помощью сеттеров, но и при помощи хэша, который передается в `.create`, например:

```
cart = Cart.new
User.create(login: "user1", cart: cart)
```

Только в этом случае вам необходимо не забыть добавить в модель `User` строку `attr_accessible :cart`.

Наконец, чтобы установить ассоциацию, нет необходимости передавать объект, можно передать только его id (если речь идет о модели, которая *принадлежит*). Например, можно было бы поступить следующим образом:

```
user = User.create(login: "user1")
Cart.create(user_id: user.id)
```

И снова не забудьте, что вам понадобится прописать `attr_accessible :user_id` в моделе `Cart`. Только в случае, когда вы передаете id, а не модель, вы несете полную ответственность за то, что пользователь с таким id существует в базе данных. Если его не окажется, то попытка получить доступ к пользователю с помощью `cart.user` вернет `nil`. Это важно помнить, потому что часто в вашем коде вы будете обращаться к свойствам ассоциации через соответствующий метод ассоциации в моделе, например:

```
cart.user.login
```

Если `cart.user` — это `nil`, то попытка вызвать на нем метод приведет к следующей ошибке: `NoMethodError: undefined method `login' for nil:NilClass`. Новички часто теряются и не знают, откуда берется эта ошибка — ведь у пользователя есть метод `#login`!

Избежать такой ситуации можно несколькими способами. Один из них, который можно упомянуть сейчас, заключается в необходимости убедиться, что ассоциация обязательно установлена для каждой записи. Для этого можно написать валидацию типа `presence` для атрибута `user` в моделе `Cart`. Кроме того, поскольку у одного пользователя может быть только одна корзина, для этого типа ассоциаций также пригодилась бы валидация типа `uniqueness`, проверяющая уникальность значения поле `user_id`. В целом, это могло бы выглядеть так:

```
class Cart < ActiveRecord::Base
  attr_accessible :user_id
  belongs_to      :user
  validates       :user,    presence: true
  validates       :user_id, uniqueness: true
end
```

[1] Обратите внимание, я удалил из миграции `t.timestamps` — корзине вряд ли нужны поля, хранящие дату ее создания.

has_many / belongs_to

Каждый пользователь в нашем магазине может оформлять заказы. И каждый может сделать не один, а много заказов. При этом сам заказ может принадлежать только какому-то одному пользователю. Таким образом, у нас возникает ситуация очень похожая на ситуацию с Пользователем и Корзиной с тем лишь отличием, что заказов у пользователей может быть несколько, а корзина — только одна. Это как раз тот случай, когда нам понадобится отношение has_many / belongs_to.

С точки зрения реализации данный тип ассоциации практически ничем не отличается от has_one / belongs_to. Мы точно также должны будем добавить в миграцию *принадлежащей* модели поле foreign id и точно так же вызвать в самой моделе метод `belongs_to`:

```
# Файл с миграцией для модели Order

class CreateOrders < ActiveRecord::Migration
  def change
    create_table :carts do |t|
      t.integer :user_id
    end
  end
end


# Файл модели Order, app/models/order.rb

class Cart < ActiveRecord::Base
  ...
  belongs_to :user
end
```

Наконец, вместо `has_one` в моделе `User` мы напишем `has_many`:

```
 class User < ActiveRecord::Base

  ...
  has_many :orders

end
```

Обратите внимание, что первый аргумент к методу (символ с названием модели, `:orders`) указан здесь во множественном числе. Не путайте, единственное число используется, когда мы создаем ассоциацию has_one. Если забыть об этом и написать неправильно — ассоциация не заработает.

После того, как вы прогоните миграцию, ассоциацию можно будет использовать. На

стороне *принадлежащей* модели (`Order`) ассоциация автоматически добавила метод `#user`, который можно использовать точно также, как и с ассоциацией has_one: т.е. можно назначать юзера через сеттер или через хэш передаваемый методам класса модели — `.create` и `.new` — а также получать доступ к связанной модели через геттер `#user`.

Немного иначе дело обстоит с объектами класса `User`, на чьей стороне в данном случае находится выражение `has_many`. В объектах этого класса стал доступен метод `#orders`, который вернет массив из всех заказов, который оформил пользователь. Если создать нового пользователя и попытаться вызвать этот метод сразу, то он вернет пустой массив, потому что этот пользователь еще не оформил ни одного заказа:

```
user = User.create
user.orders # => []
```

"Оформить" новый заказ на этого пользователя, или выражаясь более асбтрактно, связать новый заказ с пользователем, можно несколькими способами. Можно создать заказ и назначить ему этого пользователя через сеттер или хэш с атрибутами:

```
user  = User.create
order = Order.new
order.user = user
order.save
```

Можно воспользоваться методом `&#lt;` , который доступен на массиве возвращаемом методом `User#orders`:

```
user  = User.create
order = Order.new
user.orders << order
user.save
```

Обратите внимание, что в данном случае мы сохраняем объект `user`, а не объект `order`, как в первом примере.

Наконец, мы могли бы воспользоваться специальным методом `User.orders.create`:

```
user  = User.create
user.orders.create
```

Здесь `#create` вызывается на массиве (потому что именно объект класса `Array` возвращается методом `User.orders`). Это довольно необычно, потому что если посмотреть в документацию по Ruby, у класса `Array` нет такого метода. Пусть вас это не смущает: благодаря динамической природе языка Ruby авторы ActiveRecord смогли сделать так, что у объектов класса `Array`, которые возвращаются методами ассоциации has_many, этот метод есть и что он делает именно то, что от него ожидают — создает новый объект

ассоциированного класса и сохраняет его в базу данных.

Теперь, когда мы добавили несколько заказов нашему пользователю и сохранили их в базу, мы можем поочередно их просмотреть и вывести нужную нам информацию. Например:

```
user.orders.each do |o|
  print "\n"
  print "id заказа: #{o.id} | "
  print "id и логин пользователя: #{o.user.id}, #{o.user.login}"
  print "\n"
end
```

В этом примере мы могли бы заменить выражение `o.user.login` на более короткое `#{user.login}`, потому что пользователь ассоциированный с данным заказом — идентичен тому, что содержится в переменной `user`.

Ассоциация автоматически разрывается как только какой-либо заказ будет удален из базы данных. Например:

```
user = User.create
o1 = user.orders.create
o2 = user.orders.create
user.orders.count # => 2
o1.destroy
user.orders.count # => 1
```

Здесь мы видим, что после удаления заказа из БД количество заказов в массиве, возвращаемом `#orders`, автоматически уменьшилось на еденицу.

Однако все не так просто в том случае, если мы хотим удалить пользователя. По умолчанию, Rails не удаляет ассоциации на другом конце has_many и это может стать источником потенциальных ошибок. Например рассмотрим такой код:

```
user = User.create
o1 = user.orders.create
o2 = user.orders.create
user.destroy
o1.user # => nil
o2.user # => nil
o2.user.login # => NoMethodError: undefined method `login'
for nil:NilClass
```

Если в вашем приложении в каком-либо месте вы пытаетесь получить доступ к свойствам модели `User` через ассоциацию на связанной моделе `Order`, то у вас возникнет ошибка, описанная в предыдущем разделе: `NoMethodError: undefined method `login' for nil:NilClass`. Чтобы избежать таких ошибок, необходимо передать методу `has_many` опцию `dependent: :destroy`

```
class User  ActiveRecord::Base
```

```
  ...
  has_many :orders, dependent: :destroy
end
```

Теперь при удалении модели `User` будут автоматически удалены и все связанные с ней модели `Order`[1].

[1] В качестве значения для ключа `dependent` можно указать не только `:destroy`, но также и `:delete_all`. Между ними есть важная разница, которая станет очевидна, как только мы поговорим о коллбэках в моделях ActiveRecord. Подробнее об разнице между двумя опциями можно почитать в Rails Guides.

has_and_belongs_to_many (HABTM)

Любой пользователь в магазине может добавить тот или иной товар в корзину. Причем важно понимать, что поскольку наш магазин виртуальный, то и товары добавляемые в корзину — не являются конкретными единицами товара, а представляют из себя ссылку на конкретный тип товара, например "Плюшевого слона". Когда менджер будет обрабатывать заказ, он посмотрит id товара в заказе, пойдет на склад и достанет из коробки со слонами конкретного плюшевого слона. Но id никак не влияет на то, какой именно слон отправится покупателю (да это и неважно, потому что все слоны — одинаковые). То есть всем плюшевым слонам в магазине соответствует одна единственная запись в таблице `items` в базе данных.

Теперь стоит вспомнить, что у магазина — не один покупатель, а много. У каждого покупателя своя корзина и каждый может "положить" туда плюшевого слона. Таким образом получается, что слон может оказаться в нескольких корзинах. Кроме того, помимо плюшевых слонов в магазине есть и другие товары, которые пользователи могут добавлять себе в корзины — соответственно в одной корзине может быть несколько товаров. Это как раз тот случай, когда нам понадобится ассоциация `has_and_belongs_to_many`, которую сокращенно для удобства принято называть HABTM.

Особенность этой ассоциации заключается в том, что нам потребуется дополнительная таблица в базе данных, которая будет связывать две модели. Дело в том, что в отличие от `has_one` и `has_many` в этой ассоциации мы не можем хранить foreign_id в таблицах `items` или `carts`, потому что в поле для foreign_id может храниться только один id. Наша же ассоциация подразумевает, что их должно быть много. Поэтому мы создадим таблицу с двумя столбцами: `cart_id` и `item_id` которые будут однозначно связывать товары с корзинами пользователей.

В Rails этот вопрос решается созданием дополнительной миграции. В нашем случае, необходимо будет сгенерировать миграцию, для создания этой дополнительный таблицы, которая **должна** называться `carts_items` (команда `rails g migration`

create_carts_items) и заполнить ее следующим кодом:

```
class CreateCartsItems < ActiveRecord::Migration

  def self.up
    create_table :carts_items, id: false do |t|
      t.references :cart
      t.references :item
    end

    add_index :carts_items, [:item_id, :cart_id]
    add_index :carts_items, [:cart_id, :item_id]
  end

  def self.down
    drop_table :carts_items
  end

end
```

Не обращайте внимание на вызов методов add_index, о них мы поговорим чуть позже. Самое главное в этой миграции — два поля для foreign_id: одно хранит id корзины, второе хранит id товара. И стоит также отметить опцию id: false, которую мы передали методу create_table. Дело в том, что для таблицы carts_items не будет существовать отдельной модели в нашем приложении — эта таблица нужна только для связи двух существующих моделей — именно поэтому и поле id в этой таблице не требуется.

После того как мы написали и прогнали миграцию, пора добавить немного кода в наши модели, чтобы связь заработала. Тут все мало отличается от предыдущих типов ассоциаций:

```
class Cart < ActiveRecord::Base
  ...
  has_and_belongs_to_many :items

end

class Item < ActiveRecord::Base
  ...
  has_and_belongs_to_many :carts

end
```

Остается проверить, работает ли наша ассоциация. Предлагаю создать несколько корзин и подобавлять в них товары:

```
cart1 = Cart.create; cart2 = Cart.create
item1 = Item.create(name: "Плюшевый слон", description: "розовый",
price: 100, weight: 1)
item2 = Item.create(name: "Плюшевый крокодил", description: "зеленый",
price: 150, weight: 1)
item3 = Item.create(name: "Плюшевый тигр", description: "желтый",
price: 200, weight: 1)
```

```
cart1.items << item1
cart1.items << item2

cart2.items << item2
cart2.items << item3

cart1.save; cart2.save
```

Теперь убедимся, что товары действительно находятся в наших корзинах:

```
cart1.items # => [#<Item/>1 ...>, #<Item/>2 ...>]
cart2.items # => [#<Item/>2 ...>, #<Item/>3 ...>]
```

Естественно, в какой-то момент вам может потребоваться удалить товар или все товары из конкретной корзины одного пользователя, но при этом важно, чтобы запись о самом товаре осталась в таблице `items` базы данных. В этом случае мы можем воспользоваться двумя методами:

```
# удаляем из корзины один конкретный товар
cart1.items.delete(item1)

# удаляем из корзины все товары
cart1.items.delete_all
```

Основное ограничение здесь заключается в том, что для удаления конкретного товара из корзины вам нужен конкретный объект класса `Item`, который необходимо передать методу `#delete`. В следующем разделе мы посмотрим, как используя тип ассоциации `has_many :through => ...` мы сможем удалять товары из корзины зная только их id.

has_many through

В нашем предыдущем примере с товарами в корзине мы не учли одну вещь: пользователь может положить в корзину несколько одинаковых товаров (например два плюшевых слона). Конечно, мы могли бы, в этом случае, создать две записи в таблице `carts_items` в БД — по одной для каждого добавленного слона — но это не является правильным и общепринятым решением подобной задачи.

Вместо этого мы создадим еще одну модель, которую назовем `Position` (т.е. позиция в корзине или в заказе). Эта модель будет основана на таблице очень напоминающей `carts_items` — там будут поля `cart_id` и `item_id`, но помимо них в таблице также будут два дополнительных поля: `id` и `quantity`. Последнее как раз и будет показывать какое количество единиц товара было добавлено пользователем в корзину. Вот как могла бы выглядеть миграция для модели `Position`:

```
class CreatePositions < ActiveRecord::Migration

  def change

    create_table :positions do |t|
      t.references :cart
      t.references :item
      t.integer    :quantity, default: 1
      t.timestamps
    end
    add_index :positions, [:item_id, :cart_id]
    add_index :positions, [:cart_id, :item_id]

  end

end
```

Напоминаю, что поле `id` создается миграцией автоматически. Прогнав миграцию, нам нужно будет добавить следующий код в наши модели `Cart`, `Item` и `Position`

```
class Item < ActiveRecord::Base
  ...
  has_many :positions
  has_many :carts, through: :positions
end

class Cart < ActiveRecord::Base
  ...
  has_many :positions
  has_many :items, through: :positions
end

class Position < ActiveRecord::Base
  ...
  attr_accessible :cart, :cart_id, :item, :item_id, :quantity
  belongs_to :item
  belongs_to :cart
end
```

Теперь модели `Item` и `Cart` связаны между собой моделью `Position` и для каждого конкретного товара добавленного в конкретную корзину будет существовать своя запись в таблице `positions` — почти как в случае с ассоциацией `has_and_belongs_to_many`, только к таблице прилагается еще и модель ActiveRecord. И сделали мы это затем, чтобы хранить информацию о количестве добавленного в корзину товара. Но лучше показать как это работает на примере. Сделаем то же самое, что и в предыдущем разделе — создадим корзину и добавим в нее товары, только на этот раз добавим несколько единиц одного и того же товара:

```
cart = Cart.create
item1 = Item.create(name: "Плюшевый слон", description: "розовый",
```

```
price: 100, weight: 1)
item2 = Item.create(name: "Плюшевый крокодил", description: "зеленый",
price: 150, weight: 1)
item3 = Item.create(name: "Плюшевый тигр", description: "желтый",
price: 200, weight: 1)

Position.create(cart: cart, item: item1, quantity: 2)
Position.create(cart: cart, item: item2, quantity: 5)
Position.create(cart: cart, item: item3, quantity: 10)

cart.reload         # обновим данные из БД
cart.positions.size # => 3
cart.items.size     # => 3
```

Как видите, нам пришлось немного изменить подход и теперь, чтобы добавить товары в корзину, мы каждый раз создаем новую модель `Position`. Я согласен, что это не самый элегантный подход и у нас может возникнуть проблема: если пользователь последовательно добавит один и тот же товар, будут созданы две идентичные записи в таблице `positions` (точно также, как произошло бы в случае с ассоциацией HABTM) поэтому предлагаю вам создать кастомный метод для добавления товаров в Корзину, который сможет решить эту проблему. Откроем класс `Cart` и добавим туда такой код:

```
class Cart < ActiveRecord::Base

  ...

  def add_item(i)
    if items.include?(i)
      positions.where(item_id: i.id).first.increment!(:quantity)
    else
      positions.create(item: i, quantity: 1)
    end
  end

end
```

Рассмотрим что тут происходит. `if items.include?(i)` проверяет, была ли хоть одна позиция с таким товаром добавлена в корзину ранее. Если да, тогда нам необходимо найти такую позицию — это делает код `positions.where(item_id: i.id).first`[1] — и затем обновить поле `quantity` увеличив его значение на единицу с помощью метода `#increment!`[2]. Если же такой товар ранее не был добавлен в корзину, то мы просто создаем позицию с таким товаром для данной корзины[3]. Проверим работоспособность нашего метода:

```
cart = Cart.create
cart.add_item item1
cart.add_item item1
cart.add_item item1
cart.reload                  # обновим данные из БД
```

```
cart.items.size                # => 1
cart.positions.size            # => 1
cart.positions.first.quantity # => 3
```

Наконец, как я и обещал в предыдущем разделе, у нас появилась возможность удалить конкретный товар из корзины зная только id этого товара, но не располагая моделью этого товара (это будет удобно в дальнейшем, когда мы будем создавать фронтэнд для нашего приложения). Вот пример того, как это можно сделать:

```
cart.positions.where(item_id: 99).first.destroy
```

Ничего страшного, если пока что вы не понимаете как работает поиск с помощью метода `where` — мы вскоре вернемся к этому. Главное, что код `.where(item_id: 99).first` вернет нам модель `Position`, на которой мы можем вызвать уже знакомый нам метод `destroy`. Если товар с таким id был добавлен в корзину, он будет из нее удален, но вот если товар с таким id не был добавлен в Корзину или был удален из нее ранее, у нас возникнет ошибка `NoMethodError: undefined method `destroy' for nil:NilClass`. Решение будет достаточно простым — специальный метод `#try`[4] доступный любому объекту в Rails приложении позволит "умолчать" об этой ошибке, приложение просто проигнорирует ее:

```
cart.positions.where(item_id: 99).first.try(:destroy)
```

Учитывая то, что этот код стал довольно длинным, мы могли бы изобрести для него *обертку*[5] и добавить в класс `Cart` метод `#remove_position`:

```
class Cart < ActiveRecord::Base

  ...
  def remove_position(position_to_remove_id)
    positions.where(item_id: position_to_remove_id).first.try(:destroy)

  end

end
```

В качестве домашнего задания читателю предлагается написать метод для удаления одной единицы товара. Если в корзину было добавлено 10 плюшевых слонов и я хотел бы удалить только одного, у корзины должен быть специальный метод `#remove_item`, который ищет позицию по id товара и затем уменьшает поле `quantity` этой позиции на единицу. В случае когда поле `quantity` становится равно 0, позиция удаляется из базы данных совсем. Подсказка: метод будет очень похож на созданный нами ранее `#add_item`.

[1] Подробнее о языке запросов для моделей ActiveRecord можно прочитать далее в этой главе, в разделе Язык запросов для моделей ActiveRecord.

[2] Описание метода `#increment!` можно найти в Rails API.

[3] Выражение `positions.create` работает, потому что у массива, который возвращает метод `#positions` есть метод `#create` добавленный программистами Rails — волшебство! И работает он так же, как метод класса `Position.create`, только атрибут `cart` заполняется автоматически. Угадайте чем.

[4] Описание метода `#try` можно также найти в Rails API.

[5] "Оберткой" (wrapper) называют метод, который позволяет эффективнее или удобнее выполнить операцию, которую обычно выполняет другой или ряд других методов.

Полиморфные ассоциации

Нашему приложению потребуется следующий функционал: пользователи должны иметь возможно загружать к себе в профайл изображения (аватарки), а администраторы магазина должны иметь возможность добавлять к каждому товару фотографии этого товара. То есть нам абсолютно точно потребуется новая модель `Image`[1], но остается вопрос — как правильно привязать ее к моделям `User` и `Item`.

Подходом "в лоб" было бы создание ассоциации `has_many :images` для каждой из двух моделей, но тогда в таблице `images` в базе данных у нас было бы два поля — `user_id` и `item_id` — одно из которых было бы всегда пустым. Кроме того, если бы мы захотели добавить изображения к какой-нибудь другой модели — например мы бы создали модель "Новость" и захотели бы присоединять к каждой новости изображение — нам бы пришлось писать новую миграцию и добавлять в таблицу `images` поле `news_id`. И так с каждой новой моделью, которой потребовалось бы изображение.

Более логичным решением в данном случае было бы использовать полиморфную ассоциацию. В этом случае, в таблице `images` у нас будет два поля: `imageable_id` и `imageable_type`, которые уникально идентифицировали бы класс и id связанной с изображением записи. Посмотрим на миграцию для модели `Image`:

```
class CreateImages < ActiveRecord::Migration
  def change
    create_table :images do |t|
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps
    end
  end
end
```

Вас наверное смутило, почему названия полей начинаются с `imageable_`. Чтобы объяснить это, мне необходимо сначала показать вам что мы напишем в моделях `Image`, `User` и `Item`:

```
class Image < ActiveRecord::Base

  attr_accessible :imageable, :imageable_id, :imageable_type
  belongs_to      :imageable, polymorphic: true

end

class User < ActiveRecord::Base
  ...
  has_many :images, as: :imageable
end

class Item < ActiveRecord::Base
  ...
  has_many :images, as: :imageable
end
```

Для модели `Image` фактически не существует никаких связанных пользователей или товаров. Модель думает, что она связана с неким абстрактным объектом, носящим имя `imageable` — то есть "способный иметь изображения". Обычно полиморфные ассоциации так и называют, добавляя в конец слова суффикс -able. Например, если бы мы делали полиморфную ассоциацию для модели `Comment` (комментарий) и захотели бы сделать возможным комментирование не только товаров в магазине, но и новостей, то мы назвали бы такую ассоциациию `commentable`.

Работать с полиморфными ассоциациями можно также, как с обычными ассоциациями `has_many`. Нужно только не забыть правильное имя метода, который приведет вас к ассоциированной модели. В нашем случае, если вы, используя объект `Image`, хотите получить доступ к объекту `Item`, то правильно будет вызвать метод `#imageable`:

```
item        =  Item.last
image       =  Image.create
item.images << image

# Правильно
image.imageable

# Неправильно
image.item # => NoMethodError: undefined method `item' for
&#lt;Image:0x000000027a91d8>
```

Откуда Rails знает, что `imageable` для данного изображения — это именно объект `Item`, а не `User`? Потому что при создании в поле `imageable_type` было автоматически записано имя класса. Попробуйте вывести его и увидите:

```
image.imageable_type # => "Item"
```

Ну а так как в поле `imageable_id` содержится id конкретной записи `Item`, то Rails не составляет труда выбрать правильную таблицу (`items`) и найти там эту запись по id.

Разумеется, обратиться к массиву объектов `Image` используя объект класса `Item` можно также, как и с ассоциацией `has_many`:

```
item.images
```

[1] Чтобы оставаться в фокусе и не отвлекаться, на данном этапе мы не будем обсуждать сам процесс загрузки изображения (вы можете посмотреть как это делается почитав документацию к гему paperclip). Будем подразумевать, что модель `Image` волшебным образом всегда знает откуда достать необходимое изображение.

Другие типы ассоциаций

Помимо перечисленных в предыдущих разделах типов ассоциаций существует еще несколько типов, с которыми вы, может быть, столкнетесь, когда вам нужно будет решить ту или иную задачу. Для каждого такого типа я приведу в пример задачу, которая будет решаться, небольшой пример кода и ссылки на источники, где о данном типе ассоциации можно почитать подробнее. Я замечу одну вещь: нижеперечисленные типы ассоциаций не являются сильным местом Rails, поэтому тяжесть их реализации в основном ложится на разработчика конкретного приложения (то есть вас).

Single Table Inheritance

В нашем интернет магазине может быть несколько типов товаров. Хорошо, если поля у всех товаров — одинаковые. Но что если мы решим продавать мониторы и нам будет нужно сделать поиск по мониторам, который искал бы мониторы только с определенной диагональю? Очевидно, что нам нужно поле `screen_size` по которому база данных будет выполнять поиск. Также очевидно, что это поле должно быть у мониторов, но не у, например, чайников.

Самое простое решение в данном случае — воспользоваться Single Table Inheritance (STI), создать два класса — `Item` и `Display`, причем последний будет наследоваться от `Item`:

```
class Item < ActiveRecord::Base
  ...
  attr_accessible :price, :weight, :name, :description
end

class Display < Item
  ...
  attr_accessible :screen_size
end
```

Обе модели, однако, будут использовать одну и ту же таблицу в базе данных- `items`. Поэтому, если мы создадим обычный объект класса `Item` (а не объект класса `Display`), то

поле `screen_size` в соответствующей ему записи в БД останется равным значению `null`. Несложно представить, что если у вас будет много разных товаров с разными полями (много классов, наследующихся от `Item`), в таблице `items` будет очень много разных полей, и для каждой записи большинство из этих полей будут иметь значение `null`. Поэтому STI является относительно неуклюжим решением стоящей перед нами задачи.

Multiple Table Inheritance

Несколько ближе к решению предыдущей задачи подходит концепция Multiple Table Inheritance (MTI). Здесь каждому классу, который наследуется от класса `Item` будет соответствовать своя таблица в базе данных. Причем в этой таблице будут присутствовать поля специфичные только для наследующейся модели. В таблице `items` в то же время, будут содержаться только поля, которые являются общими для всех товаров.

В сети существует несколько мануалов по решению этой задачи. Однако, когда я столкнулся с похожей задачей, мне на глаза попался неплохой гем `Heritage` (который, впрочем, мне пришлось немного допилить). Моя версия гема Heritage выложена на GitHub-е; чтобы установить ее, не забудьте указать в Gemfile источник:

```
gem "heritage", "0.3.4", :git => "git://github.com/snitko/Heritage
git"
```

После установки гема вы сможете написать в своих моделях следующее:

```
class Item < ActiveRecord::Base
  ...
  acts_as_predecessor
end

class Display < ActiveRecord::Base
  ...
  acts_as_heir_of :item
end
```

Обратите внимание, что оба класса все равно наследуются от `ActiveRecord::Base`. И не забудьте сгенерировать и прогнать миграции в соответствии с документацией к гему `Heritage`.

Полиморфный has_many through

Очевидно, что модели `Cart` и `Order` в нашем приложении должны быть связаны с моделью `Item` связью `has_many :through`. То есть, переводя на русский — товары могут быть добавлены как в корзину так и в заказ, и оба в данном случае будут являться как бы контейнерами для товаров. В разделе has_many through мы уже написали код для связи

модели `Item` с моделью `Cart` — точно такой же код мог бы быть написан и для связи с `Order`. Но тогда получается, что в моделе `Position`, через которую происходит связь, нам нужны два поля: `order_id` и `cart_id` и одно из них всегда будет пустым. То есть, это явный кандидат на полиморфную ассоциацию.

К сожалению Rails не поддерживает совмещение двух данных типов ассоциаций автоматически, но этого несложно добиться следующим кодом:

```
class Item < ActiveRecord::Base
  ...
  has_many :positions
  has_many :carts,  through: :positions, source: :container,
                    source_type: "Cart"
  has_many :orders, through: :positions, source: :container,
                    source_type: "Order"
end

class Cart < ActiveRecord::Base
  ...
  has_many :positions, as: :container
  has_many :items, through: :positions
end

class Order < ActiveRecord::Base
  ...
  has_many :positions, as: :container
  has_many :items, through: :positions
end

class Position < ActiveRecord::Base
  ...
  attr_accessible :container, :container_id, :item, :item_id,
  :quantity
  belongs_to :container, polymorphic: true
end
```

Разумеется, вам нужно предварительно создать соответствующую миграцию для модели `Position`.

[1] *Single Table Inheritance with Rails*

[2] *Multiple Table Inheritance in Rails 3*

[3] *HOW TO create Rails 3 Polymorphic has_many :through Relationships*

[1] *Ассоциации ActiveRecord, описание на RailsGuides*

Коллбэки

Коллбэками (callbacks) в программировании называют специальные методы тело которых выполняется в тот или иной момент жизни объекта — то есть, можно сказать, при наступлении того или иного события (event). В Rails для объектов ActiveRecord существует несколько событий, к которым можно привязать коллбэки: создание, сохранение, обновление, удаление, валидация, загрузка из БД и инициализация. Прежде чем посмотреть на то, как создать коллбэк нам необходимо разобраться, что означает каждое из перечисленных событий и когда оно наступает.

- **Создание (create)** — наступает, когда запись о новом объекте создается в базе данных. Например, если мы сначала инициализировали объект и затем сохранили его, или если мы инициализировали и сохранили объект вызовом метода `.create`:

  ```
  item1 = Item.new(...)
  item1.save # произошло событие "создание"

  item2 = Item.create(...) # и здесь произошло событие "создание"
  ```

  Следует отметить, что событие произойдет только в том случае, если у нас не было ошибок валидации и запись действительно была добавлена в БД. Если ошибки валидации произошли, то коллбэки, привязанные к событию "создание" не выполнятся.

- **Сохранение (save)** — наступает при любом сохранении объекта, будь то сохранение нового объекта (т.е. создание записи в БД) или обновление уже существуещего объекта и записи в БД, которая ему соответствует. Ниже приведены примеры кода, которые вызовут данное событие (опять-таки, подразумевается, что валидации пройдены успешно):

  ```
  item1 = Item.new(...)
  item1.save # произошло событие "сохранение"

  item2 = Item.create(...) # и здесь произошло событие "сохранение"

  item3 = Item.first
  item3.update_attributes(name: "First Item") # и здесь тоже произошло
  событие "сохранение"

  item3 = Item.last
  item3.name = "Last Item"
  item3.save # снова произошло событие "сохранение"
  ```

- **Обновление (update)** — событие, которое наступает только когда сохраняется уже существующий объект:

```
item1 = Item.first
item1.update_attributes(name: "First Item") # произошло событие
"обновление"

item2 = Item.last
item2.name = "Last Item"
item2.save # снова произошло событие "обновление"
```

Как и в случае с двумя предыдущими событиями, данное событие произойдет только если валидации прошли и запись в БД действительно была обновлена.

- **Удаление (destroy)** — событие наступающее при удалении объекта методом `#destroy`, но не методом класса `.delete`. Например:

```
Item.first.destroy # произошло событие "удаление"
Item.delete(1)     # событие "удаление" НЕ ПРОИЗОШЛО, хотя запись
была удалена из БД
```

Обратите внимание, что метод `#destroy` — это метод объекта и вызывается на уже существуещем объекте, в то время как метод `#delete` — это метод класса принимающий в качестве аргумента id записи, которую нужно удалить из БД.

- **Валидация (validation)** — событие, которое наступает когда прогоняются валидации (независимо от того успешно ли они пройдут или возникнут ошибки). Поскольку валидации прогоняются перед каждым сохранением объекта, то это событие возникнет во всех тех же случаях, которые были указаны для события "сохранение". Плюс к этим случаям, существует еще один, когда мы можем запустить валидации вручную и, соответственно, инициировать и само события "валидация":

```
item1 = Item.new
item1.valid? # произошло событие "валидация", но НЕ событие "сохранение"
```

- **загрузка из БД** — событие которое произойдет в случае, когда объект инициализируется по существующей записи в БД, то есть, говоря проще, если мы нашли объект с помощью языка запросов ActiveRecord[1]. Вот несколько примеров (событие происходит на каждой строке в приведенном коде):

```
Item.last
Item.where(id: 1).first
Item.where("price > ?", 100).to_a # здесь событие произошло
для каждого найденного товара
```

- **Инициализация (initialize)** — событие происходящее после того как создан сам объект (но необязательно запись для него) или когда этот объект загружен из БД (т.е. одновременно с предыдущим событием). Часто коллбэк `after_initialize`

используют как заменитель *конструктора*[2] в объектах ActiveRecord, т.к. конструктор в ActiveRecord переопределять нельзя. Вот примеры кода, когда произойдет событие "инициализация":

```
Item.last
Item.new(...)
Item.create(...)
```

К каждому из перечисленных событий можно привязать коллбэк, который будет выполняться до, после или "вокруг" события. Давайте сначала посмотрим на пример, а затем обсудим детали его реализации. Предположим, что мне необходимо обновлять счетчик добавленных в категорию товаров, чтобы в браузере у пользователя возле каждой категории в нашем магазине выводилось количество товаров. Конечно, мы могли бы выполнять по одному дополнительному запросу в БД — `Item.where(category_id: c.id).count`, но если на странице показан список из нескольких категорий, это означает, что для каждой категории нужно выполнить по одному такому запросу — и внезапно мы получаем ситуацию с n+1 запросов к базе данных[3]. Чтобы этого избежать, мы добавим в модель `Category` поле `items_count` и будем обновлять его каждый раз, когда мы будем создавать или удалять товар. И здесь нам пригодятся коллбэки:

```
class Item < ActiveRecord::Base

  ...

  after_create  :increment_category_counter
  after_destroy :decrement_category_counter

  private

    def increment_category_counter
      category.inc!(:items_count)
    end

    def decrement_category_counter
      category.inc!(:items_count, -1)
    end

end
```

Здесь мы сделали две вещи: создали два приватных метода в классе `Item`, которые соответственно увеличивают и уменьшают на единицу поле `items_count` в `Category` и сохраняют изменения в БД (метод `#inc!` обновляет счетчик и сразу сохраняет модель). И затем (я говорю "затем" потому что это последовательность того, как мы писали код, а не отражение того, как он расположен в классе) мы регистрируем эти два метода в качестве коллбэков на определенные события. В данном случае, один из методов будет выполняться после создания товара, а другой — после удаления.

Таким образом, чтобы зарегистрировать метод как коллбэк, который должен выполниться перед, после или вокруг[4] какого-то события, нам необходимо вызвать специальный метод класса начинающийся на `before_`, `after_` или `around_` соответственно и передать ему в виде символа название метода, который мы хотим сделать коллбэком. Вот полный список наиболее часто используемых методов, с помощью которых можно зарегистрировать коллбэки:

- before_validation
- after_validation
- before_create
- after_create
- after_save
- before_save
- before_update
- after_update
- before_destroy
- after_destroy

Полный список методов для регистрации коллбэков [1] можно найти в Rails Guides.

Есть еще один способ создания коллбэков. Методу регистрации коллбэка можно вместо названия метода передать блок, который и будет содержать необходимый код. Например предыдущий пример с обновлением счетчиков можно было бы переписать следующим образом:

```
class Item < ActiveRecord::Base

  ...

  after_create do
    category.inc!(:items_count)
  end

  after_destroy do
    category.inc!(:items_count, -1)
  end

end
```

Как видите, когда кода в коллбэке немного, проще сделать именно так, чем создавать отдельный метод.

Стоит сказать, что мы можем добавлять к регистрации коллбэков условия. Например

предположим, что мы хотели бы отправлять пользователю, оформившему заказ, email с подтверждением, но мы должны делать это только в том случае, если пользователь отметил соответствующий флажок в настройках (и поле `User#send_confirmations`, которое мы не забудем добавить в таблицу `users` с помощью миграций, установлено в значение true для пользователя, создавшего заказ). Эту задачу можно было бы решить так:

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

  after_create :send_confirmation_email, if: -> { self.user
  send_confirmations }

  private

    def send_confirmation_email
      OrderMailer.confirmation(user.email, order).deliver
    end

end
```

Здесь при регистрации коллбэка мы передали `after_create` дополнительную опцию `if` — обратите внимание, что `if` в данном случае не ключевое слово, а ключ в хэше с опциями. Далее, в качестве значения для этого ключа мы передали *блок lambda*[5] от результата выполнения которого зависит, будет ли вызван код коллбэка и, соответственно, будет ли отправлен email. Чтобы было понятней, позволю себе привести альтернативный способ записи регистрации этого коллбэка, который поможет вам увидеть как бы этот код был написан в Ruby 1.8:

```
after_create :send_confirmation_email,
             { :if => lambda { self.user.send_confirmations } }
```

Пожалуйста убедитесь, что вы понимаете: первые фигурные скобки — это ограничители хэша, вторые — это ограничители блока lambda.

К сожалению в рамках этой книги у меня нет возможности привести достаточное количество примеров использования коллбэков, но понимание этого обязательно придет с опытом. Есть еще несколько тем касающихся коллбэков в ActiveRecord, которые мы не покрыли в этом разделе, поэтому в списке литературы ниже я привел ссылки на соответствующие материалы.

[1] О языке запросов ActiveRecord мы поговорим в разделе Язык запросов для моделей ActiveRecord этой главы.

[2] Напомню, что конструктором в ООП языках програмирования называют специальный метод, который выполняется при инициализации объекта.

[3] n+1 — это довольно распространенная и известная проблема, которой посвящено огромное количестве материалов.

[4] Подробнее про `around_` коллбэки читайте на StackOverflow.

[5] Если вы не уверены, что понимаете что такое lambda в Ruby, обязательно вернитесь к объяснению этого момента в вашей книге по языку Ruby или поищите объяснение в гугле.

 *[1] Список методов для регистрации коллбэков, Rails Guides.*

 *[2] Условия для коллбэков, Rails Guides.*

 *[3] Классы коллбэков для использования в нескольких моделях, Rails Guides.*

 *[4] Observers как альтернатива коллбэкам, Rails Guides.*

 *[5] Скринкаст на русском, посвященный коллбэкам в ActiveRecord*

Язык запросов для моделей ActiveRecord

После того как мы создали несколько товаров в магазине, нам безусловно придется выводить эти товары в каталоге и также находить товары по id и показывать подробную информацию о каждом товаре на отдельной странице. Чтобы это делать, нам необходимо научиться делать запросы к базе данных и получать назад готовые модели со всей необходимой информацией. Хорошая новость состоит в том, что нам не потребуется знать язык SQL для того чтобы делать запросы к БД. Нам достаточно будет понять как работают несколько основных методов класса `ActiveRecord::Base`, которые как раз и отвечают за формирование запроса к БД и возврат готовых моделей. С некоторыми этими методами мы уже сталкивались ранее в этой книге.

Получение объектов из БД

Самый простой способ получить один или несколько объектов из БД[1] — это воспользоваться методами класса `.first`, `.last` и `.all`. Рассмотрим каждый из них:

- `.first` вернет нам первый найденный объект. По умолчанию сортировка происходит по полю `created_at` в прямом порядке (от самого малого значения к самому большому), а это значит что метод вернет нам самый старый найденный объект.

- `.last` ведет себя почти также, как и предыдущий метод, но возвращает нам последний найденный объект.
- Наконец `.all` просто вернет массив из всех найденных объектов. Будьте осторожны, если в вашей таблице, соответствующей данной модели, очень много записей, то `.all` попытается вернуть вам все объекты и это может привести к нехватке памяти на сервере; используйте метод `.limit` речь о котором пойдет ниже.

Теперь можно продемонстрировать как работают эти методы.

```
# Удалим, для данного примера, все товары,
# которые мы добавляли ранее.
# Если этого не сделать, результаты будут совсем другие.
Item.delete_all

# Сначала создадим несколько товаров для примера
Item.create(name: "First item", ...)
Item.create(name: "Second item", ...)
Item.create(name: "Third item", ...)

# Найдем первый товар и вызовем на нем метод #name,
# чтобы убедиться, что это действительно первый объект.
Item.first.name # => "First Item"

# Найдем последний товар и также вызовем на нем метод #name
Item.last.name # => "Third item"

# Найдем все товары и с помощью Array#map превратим массив из товаров
# в массив состоящий только из имен товаров
Item.all.map { |i| i.name }
# => ["First item", "Second item", "Last item"]
```

Поэкспериментируйте с этим кодом самостоятельно. Убедитесь что вы понимаете, что теперь можно делать с полученным объектом все что угодно. Например, записать его в переменную, обновить поля и сохранить в базу:

```
item1 = Item.first
item1.name # => "First item"
item1.name = "My first item"
item1.save
item1.name # => "My first item"
```

Следующий метод, которым мы можем воспользоваться, чтобы найти объекты это метод `.find`. Особенность этого метода заключается в том, что он принимает в качестве аргумента id записи в БД, которую необходимо найти. Если такая запись существует, он вернет вам готовый объект. Если же запись с таким id отсутствует, то произойдет exception `ActiveRecord::RecordNotFound`. Например:

```
# Создадим новый товар для примера
new_item = Item.create(name: "Fourth item", ...)

# Узнаем его id. Предположим, что это в нашем
# примере это 4 - в вашем приложении это может быть другое число.
new_item.id # => 4

#Найдем товар с id == 4 в БД
Item.find(4).name # => "Fourth item"
```

Метод `.find` также может принимать массив из id — тогда будут найдены все записи, соответствующие id в этом массиве. Например:

```
# Создадим еще один товар, предположим, что его id будет равен 5
another_new_item = Item.create(name: "Fifth item", ...)
another_new_item.id # => 5

# Найдем товары с id 4 и 5 в БД
Item.find([4,5]).map { |i| i.name }
# => ["Fourth item", "Fifth item"]
```

Теперь вы можете выполнять простой поиск объектов в базе данных. Этого уже достаточно, чтобы вывести каталог товаров и показавать отдельные товары на отдельно отведенной этому товару странице. Но пока что мы еще не научились производить поиск и сортировку объектов по различным полям, которые есть у вашей модели.

[1] Когда я говорю "получить объекты из БД" это на самом деле означает следующее: сделать запрос в базу данных, найти записи и затем сделать из этих записей привычные нам объекты ActiveRecord. Rails делает это все за нас, поэтому мы можем договориться об этой абстракции. Но главное все-таки понимать, что модели нашего приложения на самом деле не хранятся в БД, они просто связаны с таблицами в которых существуют записи, описывающие каждую такую модель.

Условия поиска и метод where

В нашем магазине мы хотим дать пользователям возможность искать товары, цена которых не превышает определенную сумму. Это означает, что нам надо сделать такой запрос, который позволит нам произвести поиск по полю `Item#price`. Для этого и предназначен метод `.where`, который принимает в качестве аргумента хэш, где ключами являются названия полей нашей модели, а значениями для ключей — те условия, с учетом которых будет происходить поиск в БД. В первую очередь давайте рассмотрим относительно простой вариант, где мы будем искать все товары, цена которых *равна* 400 (обратите внимание, не *меньше*, а именно равна). Для начала создадим несколько товаров с разными ценами:

```
Item.create(name: "Item priced at 100", price: 100, ...)
Item.create(name: "Item priced at 200", price: 200, ...)
```

```
Item.create(name: "Item priced at 300", price: 300, ...)
Item.create(name: "Item priced at 400", price: 400, ...)
Item.create(name: "Second item priced at 400", price: 400, ...)
Item.create(name: "Item priced at 500", price: 500, ...)
```

И теперь найдем все товары, цена которых равна 400:

```
items = Item.where(price: 400)
items.class # => ActiveRecord::Relation
```

Сейчас нам стоит обратить особое внимание на вторую строку. Казалось бы, в переменной `items` должен находиться массив из найденных товаров и вызов метода `#class` должен был бы вернуть объект класса `Array`. На самом деле метод `.where` возвращает не массив из найденных объектов, а специальный объект `ActiveRecord::Relation`, который содержит в себе SQL-запрос, сгенерированный за вас `ActiveRecord`-ом на основе данных, которые вы передали этому `.where`. Этот запрос пока что еще не был выполнен. Чтобы выполнить запрос и найти объекты в базе данных, нам необходимо вызвать на этом специальном объекте какой-нибудь метод, который обычно есть у объектов класса `Array`, например уже знакомый нам `#map`:

```
Item.where(price: 400).map { |i| i.name }
# => ["Item priced at 400", "Second item priced at 400"]
```

Когда объект `ActiveRecord::Relation` видит, что на нем вызывают метод массива, он понимает, что надо выполнить запрос и вернуть массив с найденными объектами (или пустой массив, если не найдено ничего). Таким образом получается, что метод `#map` вызывается уже на массиве. Эта запутанная ситуация становится возможна благодаря динамической природе языка Ruby, но чтобы картина стала более понятной, мы могли бы написать код несколько более развернуто:

```
Item.where(price: 400).to_a.map { |i| i.name }
# => ["Item priced at 400", "Second item priced at 400"]
```

Здесь мы добавили к цепочке метод `#to_a`, который тоже заставил `ActiveRecord::Relation` выполнить запрос и вернуть массив. На практике, оба варианта абсолютно одинаковы.

Теперь немного изменим нашу задачу и попытаемся найти товары, цена которых не превышает 400 рублей. Чтобы это сделать, нам нужно передать методу `.where` вместо хэша специальную строку, которая в дальнейшем станет частью SQL запроса. В этом нет ничего сложного, если внимательно присмотреться:

```
Item.where("price <= 400").map { |i| i.name }
```

Такой код вполне имеет право на существование, но только в том, случае, если мы будем

указывать число 400 вручную. А поскольку мы почти всегда будем получать условия поиска от пользователя — то есть они будут приходить из формы, в которую пользователь ввел значения — нам необходимо принять меры, чтобы хакеры на смогли выполнить *sql-injection*[1]. К счастью, Rails способна принять эти меры за нас. Все что нам нужно сделать, это написать следующий код:

```
# Представим, что значение в этой переменной пришло к нам из формы
пользователя.
item_price = 400

# Выполним безопасный запрос
Item.where("price <= ?", item_price)
```

Обратите внимание на знак вопроса ? — он будет автоматически заменен на второй аргумент метода `.where`, то есть на значение переменной `item_price`. Отличие от примера выше здесь состоит в том, что Rails автоматически проверяет все значения аргументов метода `.where` на вредоносный код и в случае если таковой присутсвует — удаляет его.

Мы можем добавить несколько условий к нашим запросам, причем в обоих типах вызова метода `.where`. Вот как это делается:

```
# Найти все товары с ценой равной 400 и весом равным 10.
Item.where(price: 400, weight: 10)

# Найти все товары с ценой не больше 400 и весом меньше 10.
Item.where("price <= ? AND weight  ?", 400, 10)
```

Использование ключевого слова "AND" обусловлено здесь тем, что внутри кавычек у нас находится часть SQL-запроса, и ключевое слово "AND" является частью языка SQL. Что оно означает догадаться несложно. Если вы, в дальнейшем, немного разберетесь в языке SQL, у вас появится возможность использовать всю мощь вашей базы данных, чтобы выполнять относительно сложные запросы с помощью все того же метода `.where`. Не забывайте использовать знаки ? и последующие аргументы к этому методу, чтобы предотвратить sql-injections, способные поставить под угрозу сохранность ваших данных в БД.

Внутри строки передаваемой методу `.where` мы можем использовать не только числа или строки, но также и даты. Например вот так я смогу найти все товары, которые были добавлены в магазин в течение дня:

```
Item.where("created_at > ?", 1.day.ago)
```

Выражение `1.day.ago` работает, потому что Rails добавляет ко всем объектам класса `Fixnum` несколько полезных методов (в т.ч. метод `#day`), позволяющих преобразовывать их в промежутки времени (измеряется количеством секунд). И затем на этих промежутках (они, кстати, тоже являются объектами класса `Fixnum`) мы можем вызвать метод `.ago`, который

сконвертирует промежуток в дату.[2]

Еще один интересный момент: когда мы передаем методу `.where` хэш, мы необязательно должны указывать в качестве значения для каждого ключа число или строку — мы можем также указать массив или `Range`. Например, если мы хотим найти товары с id с 1-ого по 5-ый, то возможно написать это двумя способами:

```
Item.where(id: [1,2,3,4,5])
Item.where(id: 1..5)
```

Наконец, еще одна важная особенность заключается в том, что объект `ActiveRecord::Relation`, возвращаемый методом `.where` позволяет вызвать на себе еще один метод `.where`. Таким образом мы получаем возможность конструировать наш запрос, постепенно наращивая различные условия и в конце концов выполняя его, когда мы посчитаем нужным:

```
# Вариант в одну строку
Item.where(price: 400).where(weight: 10).to_a

# Вариант с переменной
items = Item.where(price: 400)
items = items.where(weight: 400)
items.to_a # только на этой строке запрос будет выполнен
```

Скорее всего на первом этапе этот функционал вам не понадобится, но при создании более сложных условий и появлении необходимости их компоновки, возможность собирать запрос из нескольких частей становится очень удобной.

[1] sql-injection — вредоносный запрос к базе данных, который может выполнить хакер благодаря уязвимости в программе, в частности в веб-приложении. Становится возможным, когда вводимые пользователем данные (например данные переданные программе через форму) используются без какой-либо обработки в sql-запросе. Хакер воспользовавшись этой уязвимостью может прочитать данные из БД, а также модифицировать или удалить данные.

[2] Обсуждение методов, которые Rails добавляет в классы `Fixnum`, `Date` и `Time` для удобной работы с датой и временем, выходят за рамки этой книги, но вы можете посмотреть на их список в документации Rails API. Ссылки приведены в материалах к разделу Язык запросов для моделей ActiveRecord.

Сортировка, ограничение количества получаемых записей и постраничный вывод

Пользователи вашего магазина наверняка захотят просматривать каталог отсортировав товары в нем по различным полям — например, по цене или по дате добавления товара. Чтобы дать им такую возможность, нам необходимо познакомиться с тем, как использовать специальный метод `.order`, который доступен как на классах унаследовавшихся от `ActiveRecord::Base` так и на объектах класса `ActiveRecord::Relation` (что означает, что мы можем присоединять его к цепочке `.where`). Если нам требуется вывести товары в

порядке возрастания их цены, мы можем написать такой код:

```
Item.order("price")
```

Чтобы указать обратный порядок, необходимо добавить ключевое слово языка SQL `DESC`. Следующий код покажет сначала самые дорогие товары:

```
Item.order("price DESC")
```

Естественно, гораздо логичнее было бы применить обратный порядок сортировки, когда нам нужно найти самые новые товары. Здесь "самой большой" является самая недавняя дата, поэтому такой код сработает:

```
Item.order("created_at DESC")
```

Наконец, сортировку можно производить по нескольким полям. В следующем запросе объекты сначала будут отсортированы по первом полю, а затем внутри каждой группы — по второму полю:

```
Item.order("price ASC, created_at DESC")
```

В данном коде мы сначала сортируем товары по цене, таким образом самые дешевые окажутся в начале возвращаемого массива. Затем, если у нас будет два товара с одинаковой ценой, то тот, который был создан позже (то есть тот, который поновее) будет стоять впереди.

Если в вашем магазине много товаров, вы вряд ли захотите демонстрировать их всех на одной странице — время загрузки такой страницы отпугнет любого покупателя. Поэтому важно научиться устанавливать лимит количества объектов, которые будут найдены в таком запросе. Здесь нам пригодится метод `.limit`. Этот код найдет только 10 товаров (или меньше, если количество товаров в магазине меньше 10):

```
Item.limit(10)
```

Немного усложним его и сделаем так, чтоб наш запрос нашел товары цена которых больше 100, отсортировал их по дате добавления, поставив на первое место самые новые, и затем вывел первые 10 из них:

```
Item.where("price > 100").order("created_at DESC").limit(10)
```

Конечно, само по себе ограничение количества выводимых товаров не является способом разбить вывод на несколько страниц. Для того, чтобы в дальнейшем делать постраничный вывод товаров, вам придется воспользоваться гемом will_paginate ставшим де-факто

стандартом для решения проблемы постраничного вывода. Подключив гем к вашему приложению, вы сможете написать следующий код:

```
Item.paginate(per_page: 10, page: 1)
```

который означает, что вы хотите получить 10 товаров из БД и номер текущей страницы — 1. Безусловно, метод `#paginate` можно добавлять к цепочке `.where`, например так:

```
Item.where("price > 100").order("created_at DESC").paginate(per_page:
10, page: 1)
```

Обратите внимание, что использование метода `#limit` при этом не требуется.

Загрузка ассоциаций

В нашем приложении у каждого товара может быть изображение. Как вы помните, изображение представляет из себя модель `Image` связанную ассоциативной связью с моделью `Item`. Предположим, мы хотим найти все товары и для каждого товара вывести изображение. Рассмотрим такой код:

```
items = Item.all
items.each do |item|
  p item.image
end
```

Наш пример заработает, но проблема заключается в том, что внутри блока у нас будет происходить дополнительный запрос к БД, который будет пытаться найти записи в таблице `images`, соответствующие данному товару. В итоге, каждый раз, когда будет выполняться тело блока — а оно будет выполняться ровно столько раз, сколько будет найдено товаров — Rails будет делать дополнительный запрос к БД, чтобы найти изображения для данного товара. Это означает, что мы вновь столкнулись с проблемой n+1. Чтобы решить ее, нам надо загрузить связанные с товарами изображения вместе с самими товарами, в одном запросе. Сделать это нужно так:

```
items = Item.includes(:images)
items.each do |item|
  p item.images
end
```

Теперь, при выполнении блока и обращении к ассоциации `images` дополнительный запрос к БД производиться не будет.

Загружать можно не одну, а несколько ассоциаций. Например для пользователя, мы могли бы подгрузить все заказы, а также его изображения:

```
User.includes(:orders, :images)
```

Наконец, иногда вам может понадобится загрузить ассоциации к другой ассоциированной модели. Тогда наш код немного усложняется:

```
User.includes(:orders => [{ :items => :images }, :images])
```

В этом случае к каждому пользователю будут загружены изображения, а также заказы, к каждому из которых будут загружены все товары и изображения для данных товаров.[1] Это нетривиальный запрос, смысл которого может стать понятен вам не сразу. Не волнуйтесь, делать такие запросы как правило приходится не часто.

В целом, данный подход, когда мы загружаем связанные модели заранее, называется *eager loading* [4]. Особенность здесь состоит в том, что вы не увидите никакой ошибки, если *eager loading* не произошло (если вы, например, забыли использовать метод `.include`), но запросов к базе данных у вас станет значительно больше. Чтобы контролировать количество выполняемых к БД запросов, я рекомендую вам установить гем rails-footnotes [10], который помимо всего прочего, показывает внизу страницы количество выполненных запросов к БД.

[1] Модель `Images` ассоциирована в нашем приложении как с моделью `User`, так и с моделью `Item`. Загрузка изображений для пользователя вовсе не означает автоматическую загрузку изображений для товаров. Чтобы изображения загрузились и для товаров тоже, мы написали `{ :items => :images }`.

Scopes

Scopes — это удобный способ выполнять самые распространенные запросы по поиску объектов. Рассмотрим такой код:

```
class Item < ActiveRecord::Base
  ...
  scope :cheapest, where("price  100")
end
```

Здесь мы определили scope `cheapest`. Теперь используя его мы сможем найти и вывести все товары, стоимость которых меньше 100:

```
Item.cheapest.each do |i|
  puts "Стоимость товара #{i.name} составляет #{i.price}"
end
```

Дело немного усложняется, если вы захотите создать scope для поиска товаров, которые были созданы сегодня. Очевидный, но **неправильный** способ сделать это будет выглядеть так:

```
class Item < ActiveRecord::Base
  ...
  scope :newest, where("created_at > ?", 1.day.ago)
end
```

К сожалению, код в контексте класса выполняется один раз — при загрузке приложения. Если ваше приложение проработает на сервере 10 дней подряд без перезагрузки, то данный scope будет находить все товары за последние 11 дней (1 + 10), а никак не за последний день. Чтобы избежать этого подводного камня, необходимо использовать блок lambda, внутри которого `1.day.ago` будет вычисляться каждый раз при вызове scope:

```
class Item < ActiveRecord::Base
  ...
  scope :newest, -> { where("created_at > ?", 1.day.ago) }
end
```

Подробнее про scopes и что еще можно с ними делать очень внятно описано в Rails Guides [5].

***

В этом разделе я не успел осветить ряд менее важных тем, связанных с языком запросов ActiveRecord, в частности:

- Загрузка отдельных полей, а не модели целиком [6]
- Join таблиц [7]
- Вычисления при поиске в БД [8]

Ссылки на соответствующую документацию приведены в списке литературы к этому разделу.

[1] *Документация по добавленным методам в класс Time, Rails API*

[2] *Документация по добавленным методам в класс Date, Rails API*

[3] *Документация по добавленным методам в класс Numeric, Rails API*

[4] *Eager loading associations, Rails Guides*

[5] *Описание scopes, Rails Guides*

[6] *Загрузка отдельных полей, а не модели целиком, Rails Guides*

[7] *Join таблиц, Rails Guides*

[8] *Вычисления при поиске в БД, Rails Guides*

[9] *Полная документация по языку запросов ActiveRecord, Rails Guides*

[10] *Гем rails-footnotes для получения различной информации о работе приложения через браузер.*

Миграции, часть 2: добавление столбцов и индексов

Добавление новых атрибутов в модель (или столбцов в таблицу)

В процессе работы над вашим приложением вам может понадобится добавить какие-нибудь атрибуты в модель. Например для модели `Item` может понадобиться атрибут, который будет указывать, является ли товар новым или б/у. Вполне вероятно, что ваш магазин уже будет задеплоен на продакшен сервер, уже содержит товары и обслуживает реальных пользователей. А это значит, что мы не можем себе позволить полностью откатить все миграции, добавить в миграцую создания таблицы `items` еще одно поле и прогнать миграции заново — все данные в нашем магазине потеряются. Выход заключается в том, чтобы сгенерировать новую миграцию, которая будет просто добавлять 1 столбец соответствующий новому атрибуту в таблицу `items`, не трогая при этом все другие данные.

Чтобы сгенерировать новую миграцию, выполним в терминале команду `rails g migration` и передадим ей в качестве аргумента название миграции:

```
rails g migration add_used_column_to_items
```

Обратите внимание на название миграции. Оно записано snake_case-ом и явно следует определенному стилю именования. Придерживаться его не обязательно, но обычно так поступают, чтобы взглянув на название файла с миграцией можно было сразу понять, про что он. В данном случае очевидно, что автор этой миграции хочет добавить столбец `used` в

таблицу `items`.

Откроем файл с миграцией. Вы увидите там следующий код:

```
class AddUsedColumnToItems  ActiveRecord::Migration
  def change
  end
end
```

Теперь нам предстоит заполнить тело метода `#change`. Чтобы добавить поле в таблицу, используется специальный метод `#add_column`, который мы и вызовем внутри метода `#change`:

```
class AddUsedColumnToItems  ActiveRecord::Migration
  def change
    add_column :items, :used, :boolean, default: false
  end
end
```

Первый аргумент, передаваемый методу `#add_column` — это название изменяемой таблицы, второй аргумент — название добавляемого столбца, третий аргумент — тип столбца, ну а четвертый аргумент является хэшем из опций, который мы обсуждали в первой части нашего разговора про миграции. В данном случае, мы передаем только одну опцию, которая устанавливает значение по умолчанию для всех вновь создаваемых товаров.

Прогнав миграцию командой `rake db:migrate` мы фактически добавим атрибут в нашу модель. Если по какой-то причине вы вдруг передумали или захотели исправить значение по умолчанию с `false` на `true`, вы можете откатить последнюю миграцию назад[1] с помощью команды `rake db:rollback`. Rails, в этом случае, снова прочтет метод `#change` в данной миграции, но все действия выполняемые в нем будут инвертированы, то есть выполнятся обратные им эквиваленты. Этот процесс инвентирования миграций можно описать вручную, если вместо метода `#change` добавить в миграцию два других метода: `#up` и `#down`:

```
class AddUsedColumnToItems  ActiveRecord::Migration

  def up
    add_column :items, :used, :boolean, default: false
  end

  def down
    remove_column :items, :used
  end

end
```

Код говорит сам за себя.

Добавление индексов

Когда база данных производит поиск по тем или иным полям, ей приходится сильно трудиться, перебирая каждую запись и сопоставляя поля и их значения из запроса с каждой записью. Эта операция может занять много времени, особенно если в таблице в БД очень много записей. Чтобы облегчить базе данных задачу, можно воспользоваться индексами. Индексы — это специальные обекты внутри самой БД, которые формируются из одного или нескольких столбцов в таблице и за счет оптимизации своей структуры позволяют значительно ускорить поиск. Естественно, было бы наивно предполагать, что если проиндексировать каждый столбец в таблице, любые запросы будут работать быстрее — это не так. Ваша задача — идентифицировать поля, которые являются наиболее вероятными кандидатами на индексирование и создать для них индексы в файлах миграций.

Сначала рассмотрим небольшой пример, а затем поговорим о том, какие поля необходимо индексировать. Вот пример нашей миграции для модели `Item` с добавленным индексом для столбца `name`:

```
class CreateItems  ActiveRecord::Migration

  def change
    create_table :items do |t|
      t.string :name
      t.text   :description
      t.float  :price
      t.float  :weight
      t.timestamps
    end
    add_index :items, :name
  end

end
```

Здесь мы использовали метод #add_index [2], который позволил нам добавить в таблицу `items` индекс для `name`. Обратите внимание, что метод был вызван после того, как завершился блок метода `create_table`. Это означает, что мы могли бы добавить индекс и в отдельной миграции позже, а не только при создании таблицы.

Другой пример — добавление индекса для столбца `email` в таблице `users` (здесь мы как раз создали для его добавления отдельную миграцию):

```
class AddIndexForEmailInUsers  ActiveRecord::Migration

  def change
    add_index :users, :email, unique: true
  end

end
```

Заметьте, что третий аргумент к методу `add_index` — это хэш, в котором есть одна опция, `unique: true`. Она говорит базе данных, что в этом столбце не может содержаться два одинаковых значения. Если мы попытаемся создать пользователя с существующим email-ом, произойдет ошибка БД (именно поэтому в моделе `User` следует также написать валидацию для поля email, проверяющую его уникальность, чтобы ошибка была поймана на уровне приложения!). Так или иначе, теперь поиск по уникальному индексу будет происходить еще быстрее.

Наконец, нам может понадобится создать индекс сразу по двум полям. Это может пригодиться в случае нашей полиморфной ассоциации для модели `Image`:

```
class CreateImages  ActiveRecord::Migration

  def change

    create_table :images do |t|
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps
    end

    add_index :images, [:imageable_id, :imageable_type]

  end

end
```

Теперь поиск изображений связанных с моделями `User` и `Item` будет происходить быстрее.

На всякий случай обращу внимание на, вроде бы, очевидный факт — создание индексов является частью миграции. Если миграция, приведенная выше, была запущена ранее, но без строки о добавлении индекса, то когда вы допишите в эту же самую миграцию строку `add_index :images, [:imageable_id, :imageable_type]` это не добавит желаемый индекс в БД. В этом случае, вам скорее всего нужно будет сгенерировать отдельную миграцию, добавить туда вызов метода `#add_index` и затем прогнать ее. Только если миграция не была запущена ранее и вы создаете индексы одновременно с созданием модели можно рассчитывать, что индексы создадутся в БД. То же самое, кстати, имеет отношение и к добавлению новых столбцов. Иначе говоря, если вы на запустили `rake db:migrate` и не увидели в терминале подтверждения о том, что индексы или столбцы добавлены — вы что-то напутали.

Какие поля нужно индексировать?

Ниже я собрал несколько пунктов, которые помогут вам определить, следует ли индексировать поле. Чем больше ЗА и чем меньше ПРОТИВ вы насчитаете из этого списка при принятии решения, тем вероятнее необходимость индексирования.

СЛЕДУЕТ индексировать поля:

- Которые часто используются при поиске в выражениях `find` или `where`
- Имеют высокую селективность, т.е. чем меньше повторяющихся значений — тем лучше. Хороший пример: поле `Item#name`, потому что наименования товаров обычно отличаются. Плохой пример: поле `Item#used`, потому что там может быть только два значения — `true` и `false`.
- Хранящие foreign_key, например поле `Position#cart_id` и `Position#item_id`.

НЕ СЛЕДУЕТ индексировать поля:

- По которым редко или вообще не производится поиск.
- Которые имеют низкую селективность (например могут содержать только значения `true` или `false`).
- Которые часто модифицируются. Обновление модели и сохранение ее в БД будет происходить дольше, если для измененного поля существует индекс. Например, скорее всего будет быссмысленно создавать индекс для поля `updated_at`, т.к. оно обновляется каждый раз при сохранении изменений в моделях.

Стоит отметить, что нет необходимости вручную создавать индекс для поля `id`, которое есть у всех моделей — миграции Rails сделают это за вас.

[1] Учитывая, конечно, что вы пока что еще не задеплоили новую версию приложения с последней миграцией на продакшен сервер и все изменения происходили локально.

[2] Обязательно прочитайте подробнее про то, что такое индексы и как их применять [1].

[1] Что такое индексы в базе данных, Wikipedia (на русском)

[2] Документация к методу #add_index, Rails API

[3] Полная документация по миграциям, Rails Guides

Можно ли обойтись без ActiveRecord? Альтернативные ORM/ODM и NoSQL.

В начале этой главы мы говорили о том, что ActiveRecord — это ORM-библиотека, которая является прослойкой между Rails-приложением и базой данных. Несмотря на то, что ActiveRecord работает со многими БД, включая MySQL, PostgreSQL, SQLite, SQL Server,

Sybase, и Oracle — она не умеет работать с нереляционными БД, т.н. NoSQL, такими как MongoDB или Redis. Подробное описание отличий реляционных БД от NoSQL БД выходит за рамки этой книги, но в данном случае достаточно будет упомянуть, что NoSQL БД гораздо эффективнее справляются с хранением большого объема данных, но плохо умеют выполнять сложные поисковые запросы.

Для работы с такими базами данных, самой популярной из которых на данный момент является MongoDB, были созданы отдельные библиотеки, которые, тем не менее, очень напоминают ActiveRecord. Чтобы их использовать, как правило достаточно установить соответствующий руби-гем и наследовать все создаваемые модели не от ActiveRecord::Base, а от другого класса.[1] Более того, эти библиотеки стараются свести к минимуму неудобства при переходе от ActiveRecord и даже используют некоторые модули Rails для своей работы: так в Mongoid — библиотеке для работы с MongodDB — используется модуль `ActiveModel::Validations`, поэтому все то, что вы изучили в этой книге про валидации можно применить и при работе с Mongoid. Вот код двух Mongoid-моделей и пример их использования:

```
class Artist
  include Mongoid::Document
  field :name, type: String
  embeds_many :instruments
end

class Instrument
  include Mongoid::Document
  field :name, type: String
  embedded_in :artist
end

syd = Artist.where(name: "Syd Vicious").between(age: 18..25).first
syd.instruments.create(name: "Bass")
syd.with(database: "bands", session: "backup").save!
```

Помимо библиотек для работы с NoSQL существуют также альтернативы ActiveRecord, которые отличаются в использовании, но работают со всеми теми же базами данных. Одной из таких библиотек является DataMapper. На данный момент я бы не стал рекомендовать использовать ее, но знать о существовании таких библиотек полезно. Ruby-сообщество постоянно развивается, сам ActiveRecord меняется, поэтому "зависать" на выученном однажды синтаксисе не стоит. Не исключено, что когда-нибудь ActiveRecord и вовсе потеряет свою популярность и в Rails начнут использовать по умолчанию какую-нибудь другую ORM-библиотеку.

[1] Строго говоря, эти библиотеки не всегда используют наследование. Библиотека Mongoid, например, требует делать инклуд модуля в модель, вместо наследования.

Пример модели

Ниже приведен относительно сложный пример нескольких моделей (пример взят НЕ из нашего воображаемого магазина, а из реального Rails-приложения). Убедитесь, что вы понимаете весь код — в нем нет почти ничего, чего бы мы не разобрали в этой главе. Исключение составляет использование метода `write_attribute` [1], о котором можно почитать в документации по Rails.

Модель User

```
class User < ActiveRecord::Base

  ROLES = { admin: 10, moderator: 5, visitor: 0 }

  attr_accessible :login, :email, :password, :password_confirmation,
  :name, :bio, :short_bio, :website, :send_news, :remember_me,
  :notifications

  has_many :screencasts
  has_many :sections
  has_many :subscriptions, dependent: :delete_all
  has_many :questions
  has_many :answers
  has_many :votes
  has_many :comments

  validates :login, length: { maximum: 25, minimum: 2 }, format:
  /[a-zA-Z0-9_]/, allow_nil: true, on: :create
  validates :short_bio, length: { maximum: 100 }
  validates :bio,       length: { maximum: 2000 }
  validates :login,     uniqueness: true, allow_nil: true
  before_validation :prepare_website

  def is_admin?
    role >= 10
  end

  def display_name
    name || login
  end

  private

    def prepare_website
      unless website =~ /^http:\/\//
        write_attribute(:website, "http://#{website}") unless
        website.blank?
      end
    end


end
```

Миграция для модели User

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|

      t.string :name
      t.string :bio
      t.string :short_bio
      t.string :website
      t.integer :role, :default => 0

      t.boolean :send_news, :default => false

      t.string :login
      t.string :email, :null => false
      t.string :encrypted_password, :null => false, :default
      => ""

      t.string   :reset_password_token
      t.datetime :reset_password_sent_at

      t.datetime :remember_created_at

      t.timestamps
    end

    add_index :users, :login, :unique => true
    add_index :users, :email, :unique => true
    add_index :users, :reset_password_token, :unique => true

  end
end
```

Модель Vote

```
class Vote < ActiveRecord::Base

  attr_accessible :user, :votable_id, :votable_type, :units

  belongs_to :votable, polymorphic: true
  belongs_to :user

  validates :votable, :user, presence: true
  validate  :validate_uniqueness_of_vote
  validate  :validates_voter_is_not_votable_owner

  after_create do
    votable.increment!(:votes_count, units)
    votable.user.increment!(:votes_count, units)
  end

  after_destroy do
    votable.increment!(:votes_count, -units)
    votable.user.increment!(:votes_count, -units)
  end

  def units=(n)
    if n.to_i > 0
```

```
      write_attribute(:units, 1)
    else
      write_attribute(:units, -1)
    end
  end

  private

    def validate_uniqueness_of_vote
      if vote = Vote.where(user_id: user.id, votable_id: votable_id,
      votable_type: votable_type).first
        write_attribute(:units, vote.units)
        errors.add(:base, I18n.t('models.vote.errors.uniqueness'))
      end
    end

    def validates_voter_is_not_votable_owner
      errors.add(:base, I18n.t('models.vote.errors.ownership',
      :votable => I18n.t("activerecord.models.#{votable.class
      to_s.underscore}"))) if votable.user == user
    end

end
```

Миграция для модели Vote

```
class CreateVotes < ActiveRecord::Migration
  def change
    create_table :votes do |t|
      t.integer    :units, default: 1
      t.references :user
      t.references :votable, polymorphic: true
      t.timestamps
    end
    add_index :votes, [:user_id, :votable_id, :votable_type],
    unique: true, name: "index_votes_on_user_and_votable"
  end
end
```

[1] Документация по методу #write_attribute, доступного в моделях ActiveRecord.

Глава 6. Контроллеры и ресурсы

Эта глава будет в основном посвящена контроллерам в Rails — связующему звену между views и моделями. Нам, однако, не удастся обойтись без примеров форм, которые пользователь будет отправлять из браузера и данные из которых будут поступать в тот или иной контроллер. Чтобы научиться работать с контроллерами, нам придется параллельно знакомиться и с views. Мы не будем затрагивать views слишком глубоко — им посвящена отдельная глава, но простейшие шаблоны вы научитесь создавать уже в этой главе. Для изучения этой главы вам понадобится открытый браузер и базовые знания html. И, конечно же, запущенное терминале Rails-приложение[1].

[1] Если вы уже успели это забыть, то напоминаю, что Rails-приложение запускается командой `rails s`, если вы находитесь в папке с приложением.

Типы HTTP-запросов

Прежде чем продолжить, вынужден сделать небольшое отступление, чтобы рассказать про http-глаголы и типы запросов. Как вы, надеюсь, понимаете, когда вы щелкаете по ссылке, отправляете форму или даже набираете адрес в адресной строке браузера и нажимаете клавишу `Enter` — браузер делает запрос на сервер. Браузеры могут делать два разных типа запросов на сервер — GET и POST. Собственно, эти два слова и называют "глаголами" — именно они и определяют тип запроса. Теоретически, GET запросы предназначены для получения данных с сервера, а POST запросы — для отправки данных. На практике сервер, конечно же, всегда что-нибудь отвечает, а браузер всегда что-нибудь посылает (например тот же url). Чтобы не усложнять картину, можно подразумевать, что POST запросы предназначены для тех случаев, когда мы пытаемся отправить на сервер больше данных, чем хотим получить — например, когда мы добавляем новый товар в магазин мы отсылаем на сервер POST запрос с данными о новом товаре и, возможно, его изображением, а назад получаем короткое сообщение о том, что товар создан.

На самом деле, документация к протоколу HTTP описывает не только GET и POST запросы, но и еще несколько других типов — в частности PUT и DELETE. Что они означают мы узнаем немного позже, в разделе CRUD и доступ к экшенам в RESTful контроллере. Сейчас же скажу, что браузеры их не поддерживают, но авторы Rails нашли способ обойти это ограничение и Rails успешно понимает эти типы запросов.

Чем занимается и из чего состоит контроллер?

Контроллер — это связующее звено между моделью и тем, что пользователь видит в браузере, то есть представлением (views). Когда пользователь заполняет форму и нажимает кнопку "создать товар", браузер отправляет запрос на сервер, в наше Rails-приложение. В зависимости от того, на какой url был отправлен запрос, а также какой *http-глагол* был использован (GET, POST, PUT, DELETE) — приложение решает какой контроллер и какой его экшен ответственен за то, как поступить с запросом. А дальше, в самом контроллере, мы сможем разобрать пришедшие от пользователя данные и начать на их основе взаимодействовать с моделями нашего приложения — точно так же, как мы делали это в rails-консоле, в главе посвященной моделям.

Сам контроллер в Rails-приложении представляет из себя специальный класс, название которого заканчивается на `Controller` и который наследуется от класса `ApplicationController`. Все контроллеры в вашем приложении будут находится в папке `app/controllers` и название файлов каждого из контроллеров идентично названию класса в нем содержащегося — только записанного в snake_case.

Как правило контроллеры в Rails-приложении соответствуют существующим моделям. Например, если в нашем приложении есть модели `Item` и `User`, то скорее всего, у нас в приложении будут и контроллеры `ItemsController` и `UsersController` (обратите внимание на то, что в большинстве случаев первое слово в названии контроллера идет во множественном числе — почему это так мы сможем понять немного позже). Однако это вовсе не является правилом. В нашем приложении могут существовать модели, для которых не будет контроллера, например нам вряд ли понадобится отдельный контроллер для модели `Position`, т.к. необходимые операции по добавлению и удалению товаров из корзины будет выполнять контроллер `CartsController`. В то же время, в приложении может существовать контроллер, который не имеет отношения к какой-либо модели: например для администраторов магазина можно было бы создать контроллер `SettingsController`, который бы позволял изменять различные настройки магазина и сохранять их напрямую в специальный текстовый файл с настройками.

Чтобы создать новый контроллер, нам необходимо воспользоваться генератором. Создадим контроллер, который назовем `ItemsController`. Для этого наберем следующую команду:

```
rails g controller items
```

После этого, если вы откроете файл `app/controllers/items_controller.rb` вы увидите там следующие две строки кода:

```
class ItemsController < ApplicationController
end
```

Пока что наш контроллер полностью пустой.

Чтобы сделать его хоть немного полезным, мы могли бы добавить в него *экшен* — экшеном в контроллерах Rails называют все публичные методы, потому что любой публичный метод в контроллере можно назначить ответственным за тот или иной запрос, приходящий от пользователей. Например, мы могли бы создать экшен `#hello` вот так:

```
class ItemsController < ApplicationController

  def hello
    # Следующая строка выведет в браузер приветствие
    # (пока никакого html, просто текст)
    render text: "This is ItemsController speaking, hello!"
  end

end
```

В следующем разделе мы посмотрим, как нам с помощью браузера попасть на этот экшен и увидеть на экране приветствие (пока у вас этого сделать не получится).

Наконец, в контроллер можно добавлять приватные методы, которые вы можете использовать внутри экшенов. Зачем это может понадобиться мы рассмотрим немного позднее. Но прежде чем двинуться дальше, я хотел бы прояснить еще один теоритический момент.

С точки зрения языка Ruby — класс `ItemsController` — это такой же класс, как и, например, модель `Item`. Однако экземпляры этого класса создает сама Rails, внутри своего исходного кода. В то же время, мы будем постоянно создавать много разных экземпляров наших моделей самостоятельно. Не удивляйтесь этому факту и пусть вас не вводит в заблуждение то, что формально вы лично нигде не вызываете `ItemsController.new` — повторюсь, это делает сама Rails, в нужном ей месте. Ваша задача, как прогаммиста — описать контроллер, наполнить его экшенами и другим необходимым кодом, который затем будет выполняться, когда поступит соответствующий запрос.

URL и обращение к экшенам контроллера, routes

Как уже упоминалось ранее, Rails знает в какой контроллер и в какой экшен направлять

каждый запрос, который приходит из браузера. Вернее сказать, Rails будет об этом знать, когда мы объясним это в специальном файле, который называется `config/routes.rb`. По-русски обычно так и говорят: "рауты"[1]. Этот файл как раз и отвечает за назначение url-ов и типов http-запросов тому или иному контроллеру.

Если открыть такой файл во вновь созданном приложении, то вы увидите, что в нем содержится большое количество комментариев объясняющих, как пользоваться этим файлом, но ни одной конкретной инструкции, которую Rails смогла бы интерпретировать. В нашем интернет-магазине искомый файл будет выглядеть так:

```
StoreApp::Application.routes.draw do
  # Здесь будет много закомментированных строк
end
```

Именно внутри блока, который передается методу `#draw` нам и предстоит писать инструкции. Для начала поместим туда следующую инструкцию:

```
StoreApp::Application.routes.draw do
  match ':controller(/:action(/:id))'
end
```

Данная инструкция по умолчанию добавлялась в каждое новое Rails-приложение в старых версиях Rails, пока авторы фреймворка не решили, что это нежелательный способ работы с приложением и что нам нужно переходить на RESTful контроллеры. Нам же, как новичкам в Rails, эта строка поможет разобраться с основами работы контроллеров. После того, как мы разберемся с основами и перейдем к RESTful контроллерам — можно смело удалять эту строку из файла и заменять ее на инструкции по созданию ресурсов[2].

Но пока что вернемся к нашей добавленной инструкции. `match` — это специальный метод, который, собственно, и устанавливает соответствие между url-ом и контроллером и принимает в качестве аргумента строку с инструкцией. Строка представляет из себя url, в который можно вставлять специальные placeholder-ы, которые будут означать определенные вещи. Далее мы разберем эту строку на части и поймем что каждая из них значит:

- `:controller` означает, что первое слово в url, идущее до символа / — это название контроллера, которому будет передан запрос.
- `:action` означает, что второе слово идущее в url после первого символа / — это название экшена.
- `:id` означает, что третье слово будет помещено в хэш `params` с ключом `:id`[3].
- `()` скобки, означают, что указанная часть url — необязательна.

Таким образом, теперь, если вы наберете в адресной строке браузера

`http://localhost:3000/items/hello`, то ваш запрос отправится в экшен `#hello` контроллера `ItemsController`, который мы создали в предыдущем разделе, и вы увидите на экране вашего браузера текст "This is ItemsController speaking, hello!". Если же попробовать набрать в адресной строке `http://localhost:3000/users/hello`, например, то вы получите ошибку, потому что пока что контроллера `UsersController` просто не существует.

[1] Часто можно услышать такую фразу как "добавить раут для контроллера" или "обновить рауты". Это означает, что нужно открыть файл `config/routes.rb` и отредактировать его соответствующим образом. Обращаю внимание: слово произносится именно как "рауты", не "руты".

[2] Это мы начнем обсуждать чуть ниже, в разделе Ресурсы и RESTful контроллеры

[3] О том, что такое хэш `params` читайте ниже в разделе Передача данных от пользователя контроллеру

Передача данных от пользователя контроллеру

Мы научились обращаться к определенным экшенам контроллера с помощью ввода соответствующего url-а в адресной строке браузера. Но что если пользователь хочет не только получить приветствие, но также отправить в контроллер какие-нибудь данные? Как например бывает, когда у нас есть форма, состоящая из нескольких полей. Когда пользователь нажмет кнопку "Отправить", это заставит браузер сделать запрос на определенный url и браузер отправит вместе с самим запросом все данные, которые пользователь ввел в эту форму. Остается разобраться, как эти данные получить в экшене нашего контроллера.

Прежде чем мы приступим к этому, я хотел бы для начала показать, как отправлять данные из браузера, даже если у вас нет никакой формы. Рассмотрим следующий url:

```
http://localhost:3000/items/hello?first_name=Vincent&family_name=Vega
```

Все, что находится в этом url после символа `?` называется *переменными в url*. В данном случае, у нас есть две переменные: 1) `name` со значением `Vincent` 2) `family_name` со значением `Vega`. Пары переменная-значение разделены символом `.` Эти переменные с их значениями отправятся прямиком в контроллер и ниже я покажу, как можно получить к ним доступ.

Когда пользователь отправляет форму — происходит ровно то же самое, только, как правило, формы отправляются с помощью POST, а не GET запроса. Браузеры устроены таким образом, что они показывают переменные в адресной строке только в том случае, если они отсылают на сервер GET запрос. Нас это не должно сильно волновать, но мне необходимо было сделать это замечание, чтобы было понятно, почему когда мы отправляем

форму или вручную прописываем переменные в адресной строке — мы на самом деле делаем примерно одно и то же.

И так представим, что мы отправили форму (или написали вышеприведенный url с переменными и нажали клавишу `Enter`). Запрос отправился в контроллер, в котором нам нужно получить переданные в переменных значения. Сделать это можно с помощью специального хэша `params`[1]. Например, чтобы получить доступ к имени, переданному в форме, нужно всего-лишь обратиться к соответствующему элементу этого хэша: `params[:first_name]`. Зная это, предлагаю вывести кастомизированное приветствие — вместо того, которое у нас было раньше:

```
class ItemsController < ApplicationController

  def hello
    render text: "This is ItemsController speaking, hello #{params[:first_name]}
    #{params[:family_name]}!"
  end

end
```

В этом примере мы использовали интерполяцию[2] и вставили соответствующие значения прямо в текст. Таким образом, обратившись по url-у приведенному выше, вы увидите в своем браузере следующий текст: "This is ItemsController speaking, hello Vincent Vega!"

Естественно, в большинстве случаев, нам нужно будет не просто выводить приветствия в браузер, а что-то делать с данными, которые приходят в контроллер. Например, когда администратор магазина заполнит форму по созданию нового товара, нам нужно будет использовать данные из хэша `params`, чтобы создать новую модель `Item`. Мы скоро займемся этим, но чтобы сделать это грамотно, сначала нам нужно будет разобраться с тем, что такое ресурсы.

[1] Строго говоря, `params` — это метод, доступный внутри любого метода в наших контроллерах и возвращающий хэш с отправленными в контроллер данными. Для удобства, мы далее будем говорить "хэш params" или просто "params".

[2] Если вы не знаете что такое интерполяция и не понимаете код в этом примере, обязательно прочтите об этом термине в руководстве по языку Ruby или в любой книге, посвященной этому языку.

Render и Redirect

В контроллерах Rails существует два очень важных метода, которыми новички должны обязательно уметь пользоваться и понимать разницу между ними. Эти методы называются `#render` и `#redirect_to`.

Render

В предыдущих разделах мы уже использовали метод `#render` для того, чтобы вывести в браузер текст. В реальности, вам довольно редко может понадобиться выводить просто текст — т.к. это веб приложение, вам нужна будет html-страница. Поэтому вместо хэша с опцией `:text`[1] можно передать методу `#render` просто строку с именем шаблона, который следует вывести (или еще говорят "отрендерить") в браузер. Например вот так:

```
class ItemsController < ApplicationController

  def hello
    render "hello"
  end

end
```

Если сейчас попробовать обратиться к этому экшену через браузер по адресу `http://localhost:3000/items/hello` вы увидите сообщение "Template is missing". Ничего удивительного, т.к. мы еще не успели создать шаблон, который должен рендериться. Чтобы это сделать, нужно перейти в папку `app/views/items/` и создать там файл `hello.html.erb`. Естественно, если оставить файл пустым, то и экран в браузере будет пустым. Поэтому предлагаю заполнить файл чем-нибудь полезным и посмотреть, как работают шаблоны в Rails:

```
<h1>This is ItemsController speaking</h1>
<p>
  Hello, <%= params[:name] %>
  <%= params[:family_name] %>
</p>
```

Последовательность символов <%= %> означает, что мы хотим вставить в шаблон Ruby-код. Все, что находится между этими символами считается Ruby-кодом и результат его выполнения подставляется в шаблон. В данном случае, мы сделали две такие вставки которые позволили нам вывести значение наших ключей с фамилией и именем из хэша params[2].

В шаблонах также доступны все инстансные переменные, которые были инициализорованы во время выполнения экшена в контроллере. Например, чтобы вывести количество товаров, содержащихся в нашем магазине, мы могли бы написать следующий код в контроллере:

```
class ItemsController < ApplicationController

  def hello
    # Сохраним в инстансной переменной количество товаров
    @items_count = Item.count
```

```
    render "hello"
  end

end
```

и затем в шаблоне используем эту переменную:

```
<h1>This is ItemsController speaking</h1>
<p>
  Hello, <%= params[:name] %>
  <%= params[:family_name] %>
</p>
<p>We have <%= @items_count %> items in our store!</p>
```

Стоит обратить особое внимание на две вещи. Первая: переменная должна быть именно инстансной (начинаться с символа @), чтобы быть доступной в шаблоне. Например такой код в контроллере выведет вам шаблон БЕЗ данных о количестве товаров:

```
class ItemsController < ApplicationController

  def hello
    items_count = Item.all.count # переменная items_count - локальная
    render "hello"
  end

end
```

Вторая вещь: в шаблоне доступны только те инстансные переменные, которые были инициализированы при выполнении данного запроса. Например, если бы у нас в контроллере был еще один экшен, в котором мы бы создали другую инстансную переменную, то она была бы доступна *только* при обращении к этому конкретному экшену. Например:

```
class ItemsController < ApplicationController

  def hello
    @items_count = Item.count
    render "hello"
  end

  def goodbye
    @logout_time = Time.now
  end

end
```

Переменная `@logout_time` будет недоступна в шаблоне `hello.html.erb`, если мы попытаемся ее использовать. Проверьте — добавьте ее в шаблон:

```
<h1>This is ItemsController speaking</h1>
<p>
```

```
  Hello, <%= params[:name] %>
  <%= params[:family_name] %>
</p>
<p>We have <%= @items_count %> items in our store!</p>
<p>Logout time is <%= @logout_time %></p>
```

При обращении по адресу `http://localhost:3000/items/hello` на экране браузера появится страница, в которой после фразы "Logout time is" будет пустое место.

Нам необязательно вызывать метод `#render` в каждом экшене. По умолчанию, Rails будет рендерить шаблон, который соответствует названию экшена и находится в папке, имя которой совпадает с именем контроллера. Например, если убрать из экшена `#hello` строку `render "hello"`, то не изменится ровным счетом ничего — программа будет работать точно также, как и работала, и отрендерит шаблон `hello.html.erb` из папки `app/views/items`. Только если вы хотите отрендерить шаблон, который носит другое имя (а это нам обязательно понадобится в дальнейшем), можно явно вызвать метод `#render`. Например, если мы переименуем наш шаблон в `greeting.html.erb`, то внутри нашего экшена мы могли бы написать:

```
class ItemsController < ApplicationController

  def hello
    @items_count = Item.all.count # переменная items_count - локальная
    render "greeting"
  end

end
```

Обратите особое внимание, что инициализация переменной `@items_count` произошла ДО вызова метода `#render`. Это важно, иначе переменная не будет доступна внутри шаблона. Именно поэтому метод `#render` обычно вызывается в самом конце экшена.

Redirect

Помимо метода `#render`, в контроллерах есть еще и метод `#redirect_to`. В отличие от метода `#render`, `#redirect_to` не рендерит никакой шаблон. Вместо этого он посылает в браузер инструкцию сделать еще один запрос на новый url. Url генерируется методом автоматически (хотя его и можно указать вручную, обычно этого не требуется).

Рассмотрим в качестве примера экшен `#sign_in`, который будет делать редирект на метод `#hello` в том случае, если логин и пароль — верные:

```
class ItemsController < ApplicationController

  ...
```

```
  def sign_in
    if params[:login] == "admin" &&
    params[:password] == "my_super_secure_password"
      redirect_to action: "hello"
    else
      render text: "Access denied", status: 403
    end
  end

end
```

Если мы обратимся по адресу `http://localhost:3000/items/sign_in` не передав никаких переменных в строке адреса, мы получим ошибку 403 и сообщение "Access denied". Если же мы передадим в строке адреса браузера переменные с правильными значениями — `http://localhost:3000/items/sign_in?login=admin&password=my_super_secure_password`, то браузер сделает редирект на адрес `http://localhost:3000/items/hello`. Произойдет это потому, что мы передадим методу `#redirect_to` хэш с ключем `action` значением для которого станет название экшена — того экшена, чей url нужно было сгенерировать.

Возникает вопрос: почему в данном случае нельзя было использовать `#render` вместо `#redirect_to`? Дело в том, что, большинство форм отправляются с помощью POST запросов. В нашем случае, если бы мы все немного усложнили, у нас скорее всего тоже была бы форма с двумя полями — логин и пароль — и она бы отправлялась на адрес экшена `#sign_in` в виде POST-запроса. Пользователь нажавший на кнопку "Войти" и попавший на отрендеренную страницу с приветсвием, мог бы случайно нажать на кнопку "Обновить" в браузере и тогда запрос отправился бы в экшен `#sign_in` снова. Теоретически это могло бы привести к ошибке, т.к. мы попытались бы залогинить уже залогиненного пользователя. Еще более неприятная ситуация могла бы возникнуть, если бы это была форма по созданию товара — нажатие кнопки "Обновить" в браузере после создания товара привело бы к повторному созданию такого же товара! Чтобы избежать этого, мы делаем редирект на другую страницу. В случае редиректа, нажатие кнопки "Обновить" приведет к отправке запроса на адрес `http://localhost:3000/items/hello`, а не на `http://localhost:3000/items/sign_in`. Следовательно, повторной попытки залогинить пользователя не произойдет.[3]

Стоит также отметить, что так как в случае использования метода `#redirect_to` происходит повторный запрос к серверу и контроллер выполняет код того или иного экшена заново, то все инстансные переменные, которые были установлены до редиректа — исчезнут. Это можно продемонстрировать следующим образом:

```
class ItemsController < ApplicationController

  def sign_in
    if params[:login] == "admin" &&
```

```
    params[:password] == "my_super_secure_password"
      @sign_in_time = Time.now
      redirect_to action: "hello"
    else
      render text: "Access denied", status: 403
    end
  end

  def hello
    @items_count = Item.all.count # переменная items_count - локальная
    @sign_in_time # => nil, т.е. переменная не установлена
    render "greeting"
  end

end
```

[1] Напоминаю, что ранее мы вызывали этот метод следующим образом: `render text: "Some text"`, что позволяло вывести в браузер текст, указанный в качестве значения для ключа `:text`.

[2] Обычно, использовать хэш `params` внутри шаблона — не слишком хорошая идея. Будьте осторожны и не делайте этого, если вы не уверены в себе. Почему это плохая идея, я дам возможность догадаться самостоятельно. Подсказка: поэкспериментируйте со значениями, которые передаются в переменных через url.

[3] Если вам пока что еще не совсем понятна разница между этими двумя методами — не волнуйтесь. Мы разберемся с этим в следующем разделе, посвященном RESTful контроллерам.

Ресурсы и RESTful контроллеры

В нашем приложении у нас есть несколько моделей. Пока что они никаким образом недоступны внешнему миру — то есть нельзя через браузер создать или просмотреть товары, которые есть в нашем магазине. Чтобы это сделать мы могли бы создать один большой контроллер, который мы бы назвали StoreController и сделать в нем много разных экшенов, таких как `#create_item`, `#destroy_item`, `#add_item_to_cart` и так далее. Несложно догадаться, что такой подход будет далек от оптимального и поэтому он неверный.

Вместо этого в Rails придумали использовать т.н. архитектуру REST - Representational State Transfer[1]. Контроллер представляет из себя набор заранее определенных этой архитектурой экшенов, а именно: `#new`, `#create`, `#edit`, `#update`, `#index`, `#show` и `#destroy`. Сам контроллер становится доступен по определенному url и превращается в способ доступа к коллекции *ресурсов* (по url `/items`) или к конкретному ресурсу в этой коллекции (по url `/items/1`, если речь идет о ресурсе с id 1). Управлять ресурсами можно через имеющиеся в контроллере экшены.

А в самих этих экшенах как раз и находится весь необходимый код, который манипулирует моделью. Таким образом, можно сказать, что *ресурс* является как бы фронтэндом для модели. Именно поэтому в Rails-приложении очень часто можно увидеть, что большинство моделей имеют одноименные ресурсы, представляющие из себя

контроллеры. Например, для модели `Item` мы уже создали контроллер `ItemsController`, только пока что он не является RESTful контроллером, то есть контроллером ресурса. Чтобы сделать его таковым, нам потребуется открыть файл `config/routes.rb` и внести в него изменения — удалить строку с методом `match`, которую мы добавили туда в предыдущем разделе, и вместо нее написать следующий код:

```
StoreApp::Application.routes.draw do do
  # Обратите внимание на множественное число
  # в названии метода (s на конце) - именно resourceS
  resources :items
end
```

Наш обычный контроллер `ItemsController` только что стал RESTful контроллером. Но чтобы он заработал, нам необходимо добавить в него все экшены, которые определяет архитектура REST (предварительно удалив все, что там было до этого):

```
class ItemsController < ApplicationController

  def index
  end

  def show
  end

  def new
  end

  def create
  end

  def edit
  end

  def update
  end

  def destroy
  end

end
```

Теперь наш контроллер — настоящий RESTful контроллер. Как видите, все экшены, которые требуются, в нем присутсвуют — но они пустые. То есть, фактически, на этом примере мы показали, что ресурсу необязательно как-то соотноситься с моделью. Мы можем создать произвольную коллекцию ресурсов с именем `items` и делать в экшенах все, что нам вздумается. Пользователь нашего веб приложения никогда не узнает, чем именно занимается контроллер — работает ли он с моделями ActiveRecord в своих экшенах или просто сохраняет все данные в текстовых файлах. И точно также, для любой модели в нашем приложении вовсе необзятально должен существовать RESTful контроллер — я

показывал это на примере в предыдущем разделе, когда говорил о модели `Cart`.

Мы, тем не менее, все-таки хотим научить наш контроллер манипулировать моделью `Item` и делать это таким образом, чтобы это было в соответствии с архитектурой REST. То есть чтобы когда пользователь посылал запрос в экшен `#show` — на экран ему выводились сведения о товаре, а когда он посылал запрос в экшен `#destroy` — соответствующая запись о товаре удалялась из БД. Поэтому мы последовательно рассмотрим каждые из вышеперечисленных экшенов RESTful контроллера: узнаем что должен делать каждый экшен и какой код нам следует там написать, чтобы наши ресурсы *на самом деле* заработали.

[1] Запоминать как расшифровывается REST — необязательно. Главное, чтобы вы поняли, что подразумевается под *ресурсом*.


[1] Representational state transfer, Wikipedia


[2] A Brief Introduction to REST


[3] Архитектура REST, Хабрахабр


[4] Книга "RESTful Web Services"

CRUD и доступ к экшенам в RESTful контроллере

В предыдущем разделе я сказал, что у любого RESTful контроллера есть набор определенных экшенов, которые призваны манипулировать ресурсами, обслуживаемыми данным контроллером. Если говорить в терминологии REST (забыв на секунду о Rails и контроллерах), то над ресурсами могут быть проведены следующие действия: создание (create), чтение (read), обновление (update) и удаление (delete). Эти действия получили аббревиатуру CRUD.

Ниже мы рассмотрим как CRUD соотносится с каждым из экшенов в контроллере. В качестве примера для url-ов используются ресурсы `items`. Если рядом с url-ом указан какой-либо запрос, кроме GET, то это, конечно же, означает, что мы не можем получить доступ к этому экшену просто набрав его адрес в строке адреса браузера — вместо этого нам сначала придется создать форму, которая будет отправлять соответствующий тип запроса на сервер. Это мы научимся делать в разделах, которые посвящены требующим не-GET запросов экшенам.

- Экшен `#new` — относится к "Create" экшенам. В Rails-приложении ожидается, что он должен выводить в браузер пользователю форму для создания нового ресурса.

Доступен с помощью GET запроса по адресу `/items/new`.

- Экшен `#create` — также относится к "Create" экшенам. В отличие от экшена `#new`, этот экшен принимает POST запрос на адрес `/items` (это не опечатка, просто `/items`) — запрос, обычно, приходит из формы, которая была выведена экшеном `#new`. Подразумевается, что внутри экшена мы напишем код, который будет обарабытывать поступившие из формы данные и создавать новый ресурс.
- Экшен `#show` — относится к "Read" экшенам и доступен по адресу `/items/1` для GET запроса (вместо единицы может быть другое число или даже строка, соответствующая id ресурса). Ожидается, что внутри этого экшена будет написан код, который выведет в браузер пользователю страницу, содержащую подробное описание ресурса; например в случае с товаром в магазине — это будет страница с подробной информацией о товаре.
- Экшен `#edit` — относится к "Update" экшенам и доступен по адресу `/items/1/edit`, GET запрос. Подразумевается, что экшен просто выводит нам форму редактирования ресурса — почти как экшен `#new`, только форма будет уже заполнена текущими данными, которые соответствуют ресурсу с указанным id.
- Экшен `#update` — разумеется относится к "Update" экшенам, он доступен по адресу `/items/1`, но отвечает на PUT запрос. Этот экшен должен, по замыслу, обновлять ресурс новыми данными, которые пришли в него из формы редактирования.
- Экшен `#index` — относится к "Read" экшенам, но в отличие от `#show` ожидается, что внутри него будет написан код, который позволит вывести в браузер пользователю все или какую-то часть ресурсов из коллекции — не только один ресурс. Это тот экшен, который будет отвечать за вывод каталога товаров в нашем магазине и он доступен через GET запрос по адресу `/items`.
- Экшен `#destroy` — единственный экшен относящийся к группе "Delete", доступен через DELETE запрос по адресу `/items/1`. В нем мы должны будем удалять ресурсы. Не перепутайте: экшен должен обязательно называться `#destroy`, DELETE — это имя http-глагола, который используется формой при отправке запроса на сервер.

Экшен #new

Основная задача экшена `#new` — отрендерить шаблон, который покажет пользователю форму, с помощью которой можно будет создать ту или иную модель. В нашем случае,

поскольку мы работаем над контроллером `ItemsController`, речь пойдет о форме для создания нового товара в магазине.

Прежде всего стоит вспомнить, что чтобы форма появилась на экране, нам достаточно создать соответствующий шаблон: `app/views/items/new.html.erb`. Сам экшен `#new` теоретически может оставаться пустым — как вы помните, шаблон все равно отрендерится, потому что метод `#render` выполнится автоматически. Мы, однако, все-таки потрудимся добавить в экшен одну строку кода, смысл которой я объясню немного позже. Вот как должен выглядеть наш метод `#new` в контроллере:

```
class ItemsController < ApplicationController

  ...

  def new
    @item = Item.new
  end

end
```

Здесь мы создали инстансную переменную и поместили в нее модель `Item`. Обратите внимание, что эта модель абсолютно новая: она не имеет соответствующей записи в базе данных и все ее поля — пустые.

Теперь перейдем к шаблону. Чтобы форма в нашем шаблоне работала правильно, у ее полей обязательно должны быть правильные имена и она должна отправляться на правильный адрес (адрес экшена `#create`). К счастью руками набирать весь html-код не только нет необходимости, но это, ко всему прочему, еще и неправильно с точки зрения Rails-приложения. Вместо этого, прямо внутри шаблона мы воспользуемся специальными методами, которые для нашего удобства были созданы авторами Rails. Эти методы, которые еще называют *хэлперами*, сгенерируют правильный html-код формы за нас. Нам нужно будет только передать им верные аргументы.

Для создания формы используется хэлпер `#form_for`, который в качестве аргумента принимает модель, поля для атрибутов которой содержатся в форме. И помимо аргумента этот хэлпер принимает блок, внутри которого можно вызвать другие хэлперы, которые помогут нам создать html input-ы (собственно, поля в форме). Вот как будет выглядеть наш шаблон с формой (которая пока что не содержит никаких полей):

```
<h1>Создание нового товара</h1>
<%= form_for @item do |f| %>
  <% # пока что здесь нет никаких полей %>
<% end %>
```

Мы передали методу `#form_for` модель `Item` (переменная `@item`, в которую эта модель была помещена в контроллере, доступна и в нашем шаблоне тоже). Теперь метод знает, на какой

адрес должна отправляться форма и какой html генерировать. Далее мы передали методу пустой блок, который, однако, определил один аргумент — `f`. Он понадобится нам буквально через минуту, когда мы начнем генерировать поля формы. А пока что, если посмотреть в браузер и заглянуть в исходный код страницы по адресу `http://localhost:3000/items/new`, то можно обнаружить там примерно такой html-код, сгенерированный хэлпером `#form_for`:

```
<form accept-charset="UTF-8" action="/items" class="new_item"
id="new_item" method="post">
  <div style="margin:0;padding:0;display:inline">
    <input name="utf8" type="hidden" value="✓"/>
    <input name="authenticity_token" type="hidden" value="556xmPaqSzfzLd
    .."/>
  </div>
</form>
```

Чтобы добавить поля в форму, необходимо воспользоваться специальным объектом `f` и вызвать на нем методы, соответствующие тому или иному типу поля. Например, чтобы создать текстовое поле для названия товара, мы напишем следующее:

```
<%= f.text_field :name %>
```

Как видите, метод принимает в качестве аргумента название атрибута в моделе (указывается в виде символа). Точно такие же текстовые поля можно было бы создать для всех остальных атрибутов в моделе `Item`. После этого нам останется добавить в самый низ формы кнопку "Создать" с помощью такого кода: `<%= f.submit "Создать товар" %>` и наша форма вскоре станет выглядеть вполне завершенной:

```
<h1>Создание нового товара</h1><%= form_for @item do |f| %>
  <%= f.text_field :name %>
  <%= f.text_field :description %>
  <%= f.text_field :price %>
  <%= f.text_field :weight %>
  <%= f.submit "Создать товар" %>
<% end %>
```

Обязательно посмотрите в исходный код страницы с формой в браузере — там должен быть примерно такой html, который был сгенерирован всеми использованными нами хэлперами:

```
<form accept-charset="UTF-8" action="/items" class="new_item"
id="new_item" method="post">
  <div style="margin:0;padding:0;display:inline">
    <input name="utf8" type="hidden" value="✓"/>
    <input name="authenticity_token" type="hidden" value="556xmPaqSzfzLd
    .."/>
  </div>
  <input id="item_name" name="item[name]" size="30" type="text"/>
  <input id="item_description" name="item[description]" size="30"
  type="text"/>
```

```
  <input id="item_price" name="item[price]" size="30" type="text"/>
  <input id="item_weight" name="item[weight]" size="30" type="text"/>
  <input name="commit" type="submit" value="Создать товар"/>
</form>
```

Наша форма готова, но, к сожалению, если сейчас попытаться нажать на кнопку "Создать товар" — вы получите ошибку "Template missing", потому что экшен `#create`, на который эта форма попытается себя отправить — пуст, и шаблона для него тоже нет. Мы исправим это в следующем разделе.

Стоит также упомянуть, что `f.text_field` — не единственный хэлпер, умеющий генерировать поля формы. Есть также ряд других хэлперов — например `f.hidden_field` и `f.check_box`. В нашем примере, у нас не было необходимости использовать какие-то другие типы полей формы, кроме обычных текстовых. Если вам понадобится использовать выпадающие списки, скрытые поля или что-либо еще — обратитесь к документации Rails [1]. Понять как работает каждый такой хэлпер не составляет труда: они также как и `f.text_field` принимают первым агрументом название атрибута в модели и затем могут принимать различные опциии, описанные в документации.

[1] *Список хэлперов для создания полей формы, Rails API*

Экшен #create

Когда администратор нашего магазина нажмет на кнопку "Создать товар" в форме, созданной нами в предыдущем разделе, все данные из этой формы будут отправлены в экшен `ItemsController#create`. Естественно, нам следует его предварительно написать, чем мы сейчас и займемся.

Внутри экшена `#create` нам нужно написать код, который будет делать следующие вещи:

1. Создавать новую модель и устанавливать все атрибуты этой модели в соответствии с теми значениями, которые пришли из полей формы.
2. Производить попытку сохранить эту модель в БД.
3. Если сохранение прошло успешно — редиректить браузер пользователя на какую-нибудь страницу (обычно редиректят на страницу, которая показывает информацию о только что созданной модели, т.е. на метод `#show`)
4. Если перед сохранением произошли ошибки валидации (и поэтому модель не была сохранена) — рендерить форму заново.

Кажется — придется подтрудиться, чтобы написать код, который делает все

вышеописанное. На самом деле, описание всех этих действий словами занимает больше места, чем код, который нам потребуется. Разберемся по шагам.

Шаг 1: создание новой модели из полей формы. Тут нам достаточно воспользоваться методом `.create`, который есть у всех моделей ActiveRecord. Этому методу мы можем "скормить" хэш `params[:item]`, в котором содержатся все данные из нашей формы. Каждый ключ в этом хэше соответствует одноименному атрибуту в нашей модели — именно так Rails знает, какое значение необходимо присвоить каждому атрибуту. Код будет выглядеть вот так:

```
class ItemsController < ApplicationContorller
  def create
    @item = Item.create(params[:item])
  end
end
```

У вас мог возникнуть справедливый вопрос — почему все данные из формы содержатся в хэше `params[:item]` и почему это вообще хэш? Чтобы ответить на него, вам нужно сначала посмотреть в исходный код страницы с формой, которую мы создали в предыдущем разделе. Вы увидите, что поля формы названы там необычным образом: сначала идет название модели, затем в квадратных скобках — название атрибута. Например, поле для атрибута `name` называется `item[name]`, и html-код для этого поля выглядит так:

```
<input id="item_name" name="item[name]" size="30" type="text"/>
```

С точки зрения языка html, фигурные скобки не означают ровным счетом ничего — они просто часть названия этого поля. Однако когда такое название приходит в контроллер Rails, перед тем как передать управление вашему экшену, Rails автоматически разбирает название поля и помещает значение этого поля во вложенный хэш `params[:item]`. Таким образом, доступ к этому значению можно получить обратившись к ключу `:name` этого вложенного хэша следующим образом:

```
params[:item][:name]
```

И в целом, наш вложенный хэш может выглядеть так:

```
params[:item]
# => { name: "Плюшевый крокодил",
#         description: "зеленый", price: 200.0 ... }
```

Передавая такой хэш методу `Item.create` мы создаем модель, устанавливаем ее атрибуты в значения из формы и, затем, сразу же пытаемся сохранить ее в БД, что автоматически покрывает и **шаг 2: сохранение модели в БД.**

Шаг 3: проверить есть ли ошибки валидации и сделать редирект. Здесь нам

понадобится узнать, появились ли в нашей модели какие-нибудь ошибки. Сделать это можно вызвав метод `#errors`, который, напомню, есть у каждой модели ActiveRecord и который возвращает специальный объект (очень похожий на массив), содержащий все ошибки валидаций. Поскольку этот возвращаемый методом `#errors` объект очень похож на массив, мы можем узнать, есть ли в нем какие-нибудь ошибки, с помощью метода `#empty?`. После добавления условного выражения с проверкой на наличие ошибок, наш метод будет выгядеть следующим образом:

```
class ItemsController < ApplicationContorller

  ...

  def create
    @item = Item.create(params[:item])
    if @item.errors.empty?
      redirect_to @item
    end
  end

end
```

Получается, что если ошибок нет — мы делаем редирект на конкретный товар. Упрощенная запись `redirect_to @item` — это тоже самое, что написать `redirect_to item_path(@item)`. Специальный метод-хэлпер `item_path` сгенерирует на основе имеющейся у нас модели `@item` путь `/items/[id]`, где [id] будет заменен на реальный id товара. Таким образом, в случае успешного добавления нового товара в БД, администратор сразу же попадет на страницу с описанием этого товара.

Шаг 4: если есть ошибки — снова показать страницу с формой. Здесь нам достаточно дополнить наше условное выражение ключевым словом `else` и на следующей строке — отрендерить шаблон с формой, т.е. шаблон `app/views/items/new.html.erb`. И код экшена `#create` примет свой окончательный вид:

```
class ItemsController < ApplicationContorller

  ...

  def create
    @item = Item.create(params[:item])
    if @item.errors.empty?
      redirect_to @item
    else
      render "new"
    end
  end

end
```

Теперь вы можете снова открыть в браузере форму с созданием товара и попробовать

ввести туда данные. Для начала введите неправильные данные — например оставьте поле `name` пустым (убедитесь, что у вас в моделе `Item` добавлена валидация `validates :name, presence: true`). После нажатия кнопки "Создать товар" вы должны увидеть перед собой ту же самую форму с теми же самыми полями — причем поля должны содержать ранее введенные значения. Это — нормально. Такое поведение означает, что в форме была допущена ошибка и вам нужно исправить ее[1].

Обращу внимание, что именно благодаря тому, что создаваемый объект мы записали в инстансную переменную `@item` все поля в нашей форме не очистились, а остались заполненными.

Если правильно заполнить форму и нажать кнопку "Создать товар" — вы снова должны увидеть сообщение "Template Missing". В этот раз оно появилось потому, что мы написали код, который должен редиректить нас на страницу с описанием товара, т.е. на экшен `ItemsController#show`. Шаблон для этого экшена пока отсутствует — впрочем как и сам экшен — и мы создадим его в следующем разделе.

[1] О том, как сообщить пользователю визуально о том, что была допущена ошибка при заполнении формы, читайте в главе Views, хэлперы и ассеты — вам понадобится использовать специальный метод `#flash`. Я также рекомендую использовать гем simple_form, позволяющий аккуратно выводить каждую ошибку возле каждого отдельно взятого поля в форме.

Экшен #show

Единственное назначение экшена `#show` — найти модель и вывести информацию об этой модели с помощью шаблона. То есть, говоря в терминологии REST — вывести ресурс. Код этого экшена в контроллере будет выглядеть следующим образом:

```
class ItemsController < ApplicationController

  ...

  def show
    @item = Item.find(params[:id])
  end

end
```

При обращении браузера по адресу `/items/10` наш экшен попробует найти в БД запись о товаре с id 10. Естественно, часть url после последнего слэша — число 10 — окажется в `params[:id]` автоматически. Если запись с таким id найдена не будет, ActiveRecord инициирует ошибку `ActiveRecord::RecordNotFound`, контроллер в свою очередь автоматически эту ошибку перехватит и вместо того, чтобы прекратить выполнение программы, просто покажет пользователю страницу 404. То есть, всего лишь одна строка, которую мы написали в нашем экшене `#show`, сделала за нас довольно большую работу.

Нам остается лишь добавить шаблон, который будет отвечать за вывод в браузер информации о товаре и который будет автоматически рендериться нашим экшеном. Для этого, точно также как мы сделали при создании экшена `#new`, в папку `views/items/` мы добавим файл `show.html.erb`. Внутри этого шаблона, используя переменную `@item`, мы выведем информацию о товаре отформатировав ее с помощью html-тэгов:

```
<h1><%= @item.title %></h1>
<p>Описание: <%= @item.description %></p>
<p>Цена: <b><%= @item.price %></b></p>
```

Попробуйте создать товар с помощью формы и, после того, как вы отправите эту форму, браузер должен будет редиректнуть вас на страницу с тем товаром, который вы только что создали. Благодаря написанному нами экшену `#show` и шаблону `show.html.erb` вы и увидите страницу с информацией о товаре, а надпись "Template Missing" больше не появится.

Экшен #edit

Задача этого экшена — вывести форму для редактирования товара. Вопреки впечатлению, которое у вас могло бы появиться от названия этого экшена, в его задачи не должно входить обновление записи в БД, которая соответствует конкретному товару. Это относится к обязанностям экшена `#update`. Этот же экшен очень похож на предыдущий экшен `#show`. Даже код внутри этого экшена в контроллере будет таким же:

```
class ItemsController < ApplicationController

  ...

  def edit
    @item = Item.find(params[:id])
  end

end
```

Единственное отличие от экшена `#show` будет заключаться в том, что тут у нас, в соответствии с соглашениями контроллеров Rails, будет автоматически рендериться шаблон `edit.html.erb`. Его нам и нужно создать в папке `views/items/`.

Сам шаблон должен содержать ту же самую форму, что и шаблон `views/items/new.html.erb` — это логично, т.к. все атрибуты у нас те же самые — мы же редактируем ту же самую модель! И тут у нас возникает соблазн взять и скопировать шаблон `views/items/new.html.erb`, чего я настоятельно не рекомендую делать. Вместо этого, мы воспользуемся т.н. *паршиалом* (partial) — специальным шаблоном, который можно

вставить в другой шаблон. Тогда, если наша форма будет содержаться в отдельном паршиале, мы сможем вставить ее и в шаблон `new.html.erb`, и в шаблон `edit.html.erb` избежав при этом дублирования кода.

Имена файлов всех шаблонов-паршиалов должны начинаться с нижнего подчеркивания. Поэтому мы создадим паршиал `views/items/_form.html.erb` и перенесем в него часть формы из шаблона `new.html.erb`:

```
<%= f.text_field :name %>
<%= f.text_field :description %>
<%= f.text_field :price %>
<%= f.text_field :weight %>
```

Если вы внимательно присмотритесь, то увидите, что я перенес в паршиал `_form.html.erb` все поля формы, кроме последней кнопки "Создать товар". Также я оставил в шаблоне `new.html.erb` строку с вызовом метода `#form_for`. Через секунду я поясню, почему я сделал именно так, а пока что посмотрим на измененный код шаблона `new.html.erb`, в который я добавил строку для загрузки созданного нами паршиала:

```
<h1>Создание нового товара</h1>
<%= form_for @item do |f| %>
  <%= render partial: "form", locals: { f: f } %>
  <%= f.submit "Создать товар" %>
<% end %>
```

Могу поспорить, вас смутила опция `locals: { f: f }` к вызову метода `#render` [1]. Все очень просто: с помощью нее можно установить локальные переменные в вызываемом шаблоне. Так как для построения нашей формы используется специальный объект, содержащийся в переменной `f`, нам нужно, чтобы эта переменная была доступна и в паршиале. Первая `f` — ключ, это название переменной в паршиале. Мы могли бы назвать ее не `f`, а, например, `form`, но тогда и все упоминания переменной `f` в шаблоне `_form.html.erb` нужно было бы переделать в `form` (не будем все усложнять!). Вторая `f` — значение, которое будет записано в локальную переменную. Очень важно туда передать именно специальный объект для конструирования формы, который как раз и содержится в переменной `f`.

Возвращаясь к причине, по которой я не перенес весь код в шаблон `_form.html.erb` следует объяснить, что нам было важно, чтобы кнопка "Создать товар" осталась исключительно в шаблоне с формой *для создания товара*, в то время как в шаблоне для редактирования товара эта кнопка называлась бы иначе. Вот как будет выглядеть шаблон `edit.html.erb`:

```
<h1>Редактирование товара</h1>
<%= form_for @item do |f| %>
  <%= render partial: "form", locals: { f: f } %>
```

```
  <%= f.submit "Сохранить товар" %>
<% end %>
```

Теперь форма редактирования всегда доступна для каждого товара по адресу `/items/[id]/edit`

Тут можно заметить, что строка, где вызывается метод `form_for` в обоих шаблонах осталась все-таки одинаковой. Получается, что форма будет отправляться на один и то же адрес? Вовсе нет. Дело в том, что хэлпер `form_for` проверяет передаваемую ему модель (в наших примерах это `@item`) и если модель еще не была сохранена в БД — как в случае с моделью в экшене `#new` — он сделает так, чтобы форма отправлялась на url, соответствующий экшену `#create`. В случае если запись о данной моделе уже есть в БД, форма будет указывать на адрес, соответствующий экшену `#update`, речь о котором пойдет как раз в следующем разделе. А убедиться, что формы будут отправляться на разные адреса можно, если посмотреть исходный код страниц редактирования и создания товара — обратите внимание на атрибут `action` у тэга `<form>` — он будет отличаться.

> *[1] Подробнее о рендеринге паршиалов читайте в Rails API*

Экшен #update

Если вы все сделали правильно в предыдущем разделе, то при нажатии кнопки "Сохранить товар" ваша форма редактирования товара должна отправиться в экшен `#update`, то есть в нашем примере на url `/items/[id]` с использованием запроса PUT[1].

Сам экшен `#update` по своим задачам очень похож на `#create`: ему необходимо обновить запись в базе данных новыми данными пришедшими из формы, если есть ошибки валидации — отрендерить форму редактирования заново, если ошибок нет — сделать редирект на экшен `#show`. Одно из важных отличий состоит в том, что перед всеми этими действиями экшену `#update` необходимо сначала отыскать в БД запись о ресурсе, который он редактирует.

Не стану подробно описывать каждое действие — мы сделали это в разделе посвященном экшену `#create` — а лучше покажу код:

```
class ItemsController < ApplicationController

  ...

  def update
    @item = Item.find(params[:id])
    @item.update_attributes(params[:item])
    if @item.errors.empty?
      redirect_to @item
```

```
    else
      render "edit"
    end
  end

end
```

Как видите, сначала мы нашли товар, также как мы сделали это внутри экшенов `#show` и `#edit`, а затем, используя хэш с полями формы `params[:item]`, попытались обновить запись в БД. Только в случае, если валидации в моделе вызовут ошибки, рендрить мы будем не шаблон `new.html.erb`, как в экшене `#create`, а шаблон `edit.html.erb`.

Если по какой-то причине вы не понимаете код, приведенный в этом разделе — перечитайте раздел про экшен `#create`. А мы тем временем приступим к следующему CRUD-экшену.

[1] Как я отмечал ранее, браузеры не умеют отсылать PUT запросы. В Rails придумали хитрость: в форму редактирования товара автоматически добавляется скрытое поле с именем "_method" (его можно увидеть, заглянув в исходный код страницы). Когда запрос приходит в Rails контроллер, он замечает присутствие этого поля и, если его значение равно "PUT", запрос направляется в соответствующий экшен, т.е. в экшен `#update`.

Экшен #index

Все экшены, которые мы рассматривали до сих пор имели дело с каким-то одним ресурсом. В задачи экшена `#index` входит поиск всех ресурсов и вывод их на экран. В самом простом случае, код экшена будет состоять из, всего лишь, одной строки:

```
class ItemsController < ApplicationController

  ...

  def index
    @items = Item.all
  end

end
```

Обратите внимание, что мы назвали инстансную переменную *во множественном числе*. Это никак не влияет на поведение программы, но программисты читающие код, сразу смогут понять по имени переменной, что в ней содержится массив из найденных объектов класса `Item`, а не один такой объект.

Erb-шаблон для этого экшена будет представлять из себя следующий код:

```
<h1>Все товары</h1>
<ul>
  <% @items.each do |i| %>
    <li><%= link_to i.title, i %></li>
  <% end %>
```

```
</ul>
```

Здесь при помощи метода `Array#each` мы перебираем массив и выводим каждый его элемент внутри тэга `li`. Причем мы выводим не всю информацию о товаре, а только его название, которое превращаем в ссылку с помощью хэлпера `#link_to`.

Но задумаемся на секунду, а что произойдет если в нашей БД будет 1000 товаров? Все они окажутся на одной странице, что сделает загрузку страницы относительно долгой, а саму страницу не очень удобной к просмотру. Было бы неплохо, если бы товары выводились постранично. К счастью, для этих целей существует специальный гем, который называется will_paginate. Добавьте строку `gem 'will_paginate'` в ваш `Gemfile` и запустите из терминала команду `bundle install`. После этого, у вас появится возможность изменить код экшена `#index` следующим образом:

```
class ItemsController < ApplicationController

  ...

  def index
    @items = Item.paginate(per_page: 10, page: params[:page])
  end

end
```

Несложно догадаться, что изменяя значение переменной `page` в адресной строке браузера, можно попасть на разные страницы с товарами. Например, адрес `/items?page=2` приведет вас на вторую страницу. Хотелось бы, однако, чтобы переключаться между страницами было возможно с помощью ссылок, а не изменяя переменные в адресной строке вручную. С гемом `will_paginate` не может быть ничего проще. Просто добавьте следующий код в самый низ шаблона `index.html.erb`:

```
<%= will_paginate @items %>
```

и внизу страницы у вас автоматически появятся ссылки навигации, позволяющие переключаться между страницами с товарами.

Наконец, вам может понадобиться не просто выводить все товары, но также сортировать их в определенном порядке или делать так, чтобы выводимые товары соответствовали определенным условиям. В этом случае, ничто не мешает вам добавить в экшен `#index` выражение `.where(...)` или `.order(...)`:

```
class ItemsController < ApplicationController

  ...

  def index
    @items = Item
```

```
    # Выводим только активные товары
    @items = @items.where(active: true)

    # Устанавливаем сортировку
    order = params[:order] == "asc" ? "asc" : "desc"
    @items = @items.order("created_at #{order}")

    @items = @items.paginate(per_page: 10, page: params[:page])
  end

end
```

Рассмотрим каждую строку кода. Для начала мы решили выводить только те товары, которые на данный момент являются активными[1], что, в общем логично — зачем показывать товары, которые по той или иной причине недоступны в продаже, но тем не менее есть в нашей базе?

Далее следует странная строка `order = params[:order] ? "desc" : "asc"`. Это пример использования *тернарного оператора*[2] (если вы не знаете, что это такое — убедитесь, что разобрались в этом, прежде чем продолжить чтение книги). В params[:order] у нас будет содержаться порядок, в котором должны быть отсортированы результаты: если у пользователя в адресной строке браузера будет стоять `/items?order=asc`, то на самом верху списка на первой странице будут находиться самые старые товары. Если же переменная `order` в адресной строке будет отсутствовать, либо же у нее будет какое-либо другое значение, то наверху списка окажутся самые новые товары. За выполнение этой логики как раз и отвечает следующая строка `@items = @items.order("created_at #{order}")`. Причина, по которой нам пришлось определять локальную переменную `order`, заключается в том, что мы хотели избежать sql-injections. Если бы наш код выглядел так:

```
@items = @items.order("created_at #{params[:order]}")
```

злоумышленник мог бы с легкостью вставить в params[:order] вредоносный код.

Напомню, что т.к. методы `.where` и `.order` возвращают не результаты запроса, а сами сконструированные запросы в виде объекта ActiveRecord::Relation[3], мы можем "наращивать" условия запроса постепенно, используя все ту же переменную `@items`. Главное не забыть, что если бы мы написали `@items.order("created_at #{order}")` вместо `@items = @items.order("created_at #{order}")`, то это условие "пропало" бы, т.к. новый объект `ActiveRecord::Relation` не был бы перезаписан в переменную `@items`.

Теперь покажем на странице со списком товаров специальные ссылки, которые позволят пользователю отсортировать товары. Изменим наш шаблон `index.html.erb` следующим образом:

```
<h1>Все товары</h1>
```

```
<p>
  Сортировка:
  <%= link_to "Новые сверху",  items_path(order: "desc") %> |
  <%= link_to "Старые сверху", items_path(order: "asc")  %>
</p>
<ul>
  <% @items.each do |i| %>
    <li><%= link_to i.title, i %></li>
  <% end %>
</ul>
<%= will_paginate @items %>
```

Несложно себе представить, что похожим образом можно добавлять различные условия к поиску ресурсов в любом контроллере. Главное быть предельно аккуратным с вставкой данных пришедших от пользователя в запрос и всегда помнить об sql-injections.

[1] При создании модели `Item` мы не добавляли атрибут `#active` (и не прописывали создание соответствующего поля в миграциях), но ради данного примера представим, что мы это сделали и что он может принимать два значения — true или false.

[2] Тернарная условная операция — удобный способ записи оператора if-else. В данном случае она означает следующее: если `params[:order]` равен "asc", тогда в переменную `order` нужно записать значение "asc", во всех других случаях в переменной `order` должно быть значение "desc". Подробнее о тернарной условной операции можно прочитать в википедии. [1]

[3] Подробнее об объектах `ActiveRecord::Relation` и "наращивании" запросов читайте в разделе Условия поиска и метод where главы Модели, ActiveRecord и взаимодействие с БД

[1] Тернарная условная операция, Wikipedia (русский)

Экшен #destroy

Основная и единственная задача, которая, как правило, стоит перед экшеном `#destroy` — найти и затем удалить ресурс. Экшен доступен по тому же адресу, что и `#update`, но отвечает на другой http-глагол — не PUT, а DELETE.

Обычно, при удалении конкретного экземпляра модели у нас не возникает никаких критических ошибок и уж точно у нас нет никаких валидаций, которые должны пройти перед удалением (в Rails невозможно сделать валидации перед удалением объекта, во всяком случае стандартным способом). Поэтому тело нашего экшена будет состоять всего лишь из трех строк:

```
class ItemsController < ApplicationController

  ...

  def destroy
    @item = Item.find(params[:id])
    @item.destroy
    redirect_to items_path
  end
```

```
end
```

Последняя строка внутри экшена перенаправит браузер пользователя на список всех товаров (то есть на экшен `#index`, о котором мы поговорим в следующем разделе).

Как видите, сам код экшена не представляет из себя ничего сложного. Однако, пока что вам наверняка не совсем понятно, каким образом создать ссылку или кнопку, которая позволит нам послать запрос к этому экшену со страницы редактирования товара (согласитесь, это логично — дать возможность удалить товар на странице редактирования).

Для начала нужно понять, на какой адрес должна указывать ссылка. Чтобы это выяснить, находясь в папке проекта выполним в терминале команду `rake routes`. Вот что она должна показать нам:

```
    items GET    /items(.:format)          items#index
          POST   /items(.:format)          items#create
 new_item GET    /items/new(.:format)      items#new
edit_item GET    /items/:id/edit(.:format) items#edit
     item GET    /items/:id(.:format)      items#show
          PUT    /items/:id(.:format)      items#update
          DELETE /items/:id(.:format)      items#destroy
```

Интересующий нас адрес находится в самом низу. И сразу становится понятно, что хэлпер, который сгенерирует нам нужный адрес называется `#item_path` (т.к. адрес точно такой же, как и для экшена `#show`). Но проблема для нас заключается в том, что мы не можем просто создать ссылку на этот экшен обычным способом, ведь все ссылки в браузере, по которым щелкает пользователь, заставят браузер выполнить GET запрос, независимо от того адреса, на который указывает ссылка. Поэтому, если мы добавим в наш шаблон следующий код:

```
link_to "Удалить товар", item_path(@item)
```

то при клике по ссылке, товар вовсе не будет удаляться, как можно было бы подумать. Вместо этого, пользователь просто увидит перед собой страницу с описанием товара (потому что браузер загрузит то, что получилось после рендеринга шаблона `show.html.erb`).

К счастью, в Rails есть простой способ сгенерировать правильную ссылку для удаления ресурса. Для этого вам нужно всего лишь добавить пару дополнительных опций к хэлперу `#link_to`:

```
link_to "Удалить товар", item_path(@item),
        method: :delete, data: { confirm: "Уверены?" }
```

В этом случае, хэлпер `#link_to` сгенерирует дополнительный javascript, понятный браузеру, который позволит а) вывести диалоговое окно с запросом подтверждения на удаление товара и б) послать DELETE запрос на соответствующий адрес[1]. Естественно, такую ссылку

можно добавить в любой шаблон, не только в `edit.html.erb` — все зависит от того, где вы считаете уместным присутствие этой ссылки. Главное, чтобы хэлпер `item_path` получил в качестве аргумента сохраненный объект класса `Item` (то есть не новый, а такой, у которого есть id).

[1] Чтобы этот код заработал, убедитесь, что в вашем `Gemfile` присутствует строка `gem 'jquery-rails'`

Дополнительные RESTful экшены

Иногда стандартных CRUD-экшенов не хватает для того, чтобы реализовать весь необходимый функционал вашего контроллера. Скажем, в соответствии с вашей задумкой, с ресурсом можно производить и другие действия — помимо создания, чтения, обновления и удаления. Предположим, что мы хотели бы создать отдельный экшен для активации товара. В контроллере это выглядело бы так:

```
class ItemsController < ApplicationController

  ...

  def activate
    @item = Item.find(params[:id])
    @item.active = true
    @item.save
  end

end
```

Однако пока что этот экшен недоступен внешнему миру и на него никак нельзя попасть из браузера.

Чтобы он стал полноправным участником нашего RESTful контроллера, нам необходимо будет открыть файл `config/routes.rb` и немного подредактировать объявление ресурса items:

```
resources :items do
  member do
    post :activate
  end
end
```

Разберем это выражение. Первый блок, который передан методу `#resources` по сути является "оберткой" для опций этого ресурса — просто примите такой синтаксис как данное. Метод `#member`, которому передается второй блок, говорит Rails о том, что сейчас мы будем добавлять к ресурсу экшены, которые будут действовать для каждого конкретного ресурса в отдельности, а не для всей коллекции. Это означает, что наш экшен `#activate`

станет автоматически доступен по адресу `/items/[id]/activate`. Наконец метод `#post` говорит Rails о том, что для экшена `#activate` нужно передавать только POST-запросы, пришедшие на вышеуказанный адрес `/items/[id]/activate`.

Остается добавить что-нибудь в уже созданный нами шаблон `app/views/items/edit.html.erb`, чтобы мы могли отправить POST запрос на адрес `/items/[id]/activate`. В идеальном случае это была бы кнопка, но чтобы сэкономить нам немного времени и объяснения того, как работает Javascript, поступим также, как мы поступили в разделе Экшен #destroy и создадим ссылку с подтверждением:

```
link_to "Активировать товар", activate_item_path(@item),
        method: :post, data: { confirm: "Точно активировать товар?" }
```

Обратите внимание, как мы использовали хэлпер `#activate_item_path` — он стал нам доступен именно благодаря тому, что мы соответствующим образом отредактировали определение нашего ресурса в `config/routes.rb`.

Добавлять новые экшены можно не только к отдельно взятым ресурсам, но и ко всей коллекции. Предположим, что нам понадобился отдельный экшен, чтобы выводить только самые популярные товары (и пусть популярность товара определяется специальным полем `Item#rating`). Тогда мы могли бы написать такой экшен:

```
class ItemsController < ApplicationController

  ...

  def top
    @items = Item.where(active: true).order('rating DESC').limit(10)
  end

end
```

Этот код найдет 10 самых популярных товаров. Осталось объяснить Rails, что мы хотели бы сделать этот экшен доступным по адресу `/items/top`, причем чтобы он отвечал на GET запрос. Снова отредактируем файл `config/routes.rb`:

```
resources :items do
  member do
    post :activate
  end
  collection do
    get :top
  end
end
```

Метод `#collection` как раз и объясняет Rails, что экшен `#top` имеет отношение ко всей коллекции, а не к конкретном ресурсу.

Хотелось бы особо отметить, что мы выбрали http-глагол POST для экшена `#activate`,

потому что внутри этого экшена мы производим обновление записи в БД. В то же время, внутри экшена `#top`, обновление БД не происходит, мы только читаем данные из БД — отсюда и выбор http-глагола GET.

Теоретически в контроллер можно добавлять сколько угодно дополнительных RESTful экшенов, но если вы заметили, что ваш контроллер начал разрастаться, то, возможно, вы что-то делаете неправильно — может быть вместо одного навороченнего дополнительными экшенами контроллера, у вас должно быть два обычных RESTful контроллера. Всегда помните, что излишнее усложненение может быть свидетельством того, что структуру вашей программы нужно разбить на части и, таким образом, упростить.

Приватные методы

В языке Ruby приватные методы объекта — это такие методы, которые недоступны извне. То есть если я создам приватный метод `#do_something` внутри класса `Item`, то инициализировав объект этого класса, я не смогу вызвать на нем этот метод. Например:

```
class Item

  private
    def do_something
      ...
    end

end

item1 = Item.new
item1.do_something
# => NoMethodError: private method
# `do_something' called for &#lt;Item:0x0000000124cda8>
```

То есть мы получим ошибку, которая говорит, что данный метод нельзя вызвать "снаружи", потому что он приватный.

В Rails классы контроллеров используются несколько иначе — нам не приходится вручную создавать объекты класса `ItemsController`, но в наши задачи, как Rails-разработчика, входит написание всех внутренностей контроллера. Естественно, в Rails-контроллерах, как и в любых других классах Ruby, мы можем определить приватные методы, но они будут иметь несколько другое смысловое значение. С точки зрения Rails, все публичные методы внутри контроллера — это экшены, которые должны быть доступны посетителю сайта через то или иное сочетание url + http-глагола. Приватные же методы экшенами не являются, но могут быть вызваны изнутри любого экшена в контроллере.

Первый вопрос, который сразу же возникает: зачем нам могло бы понадобиться создавать

приватные методы и переносить код туда? Неужели нельзя было бы просто поместить нужный код внутрь экшена? Давайте представим себе ситуацию, когда нам нужно предотвратить возможность создания или редактирования товара посетителями сайта (т.е. покупателями) и дать эту возможность только администраторам. Предположим, что у нас в приложении имеется модель `User` со специальным полем `#admin`, которое может принимать значения `true` или `false`. И также предположим, что получить объект содержащий текущего пользователя, у нас волшебным образом можно через специальный метод `#current_user`[1], который доступен во всех контроллерах. Тогда мы могли бы попробовать ограничить доступ к экшену `#create` не-администраторам сайта следующим образом:

```
class ItemsController < ApplicationController

  ...

  def create
    unless current_user
      redirect_to "/users/sign_in" and return
    end

    unless current_user.admin
      render(file: "public/403.html", status: "403.html") and return
    end

    @item = Item.create(params[:item])
    if @item.errors.empty?
      redirect_to @item
    else
      render "new"
    end
  end

end
```

Такой код действительно будет работать именно так, как мы и ожидаем: 1) в случае если пользователь вообще никак не идентифицирован, мы редиректим его на страницу "Входа" 2) если же пользователь не является администратором, то мы покажем ему страницу 403 ("Доступ запрещен").

Однако же стоит вспомнить, что `#create` не единственный экшен, который должен быть доступен только администраторам. Также нам нельзя позволять обычным пользователям редактировать или удалять товар. А это значит, нам придется написать те же самые две строки в начале методов `#new`, `#edit`, `#update` и `#destroy`. Это будет противоречить упоминавшемуся в начале книги принципу DRY (Don't Repeat Yourself) и будет означать, что в нашем классе есть большое количество повторяющихся строк кода.

Поэтому самый простой выход из ситуации — вынести эти строки кода в отдельный приватный метод, который потом станет доступным внутри любого экшена в контроллере. Попробуем написать так:

```
class ItemsController < ApplicationController

  ...

  def create
    return unless has_admin_privileges?
    @item = Item.create(params[:item])
    if @item.errors.empty?
      redirect_to @item
    else
      render "new"
    end
  end


  private

    def has_admin_privileges?

      unless current_user
        redirect_to "/users/sign_in" and return
      end

      unless current_user.admin
        render(file: "public/403.html", status: "403.html") and return
      end

      true
    end

end
```

Здесь мы немного видоизменили сам код, ответственный за проверку пользователей. Обратите внимание как мы написали `return false` вместо просто `return`. Это нужно затем, чтобы в самой первой строке тела экшена `#create` мы смогли бы проверить возвращаемое значение и прекратить выполнение метода ключевым словом `return`. Если этого не сделать, метод продолжит выполняться и Rails выдаст вам ошибку о том, что `#render` или `#redirect_to` был вызван несколько раз (а этого делать, как вы помните, нельзя). Наконец, в самом низу метода `#has_admin_privileges?` мы возвращаем значение `true` — оно вернется только в том случае, если текущий пользователь действительно является администратором — если это не так, то сработает одно из перечисленных выше условий и метод вернет `false` и прекратит выполнение до того момента, как доберется до `true`.

Теперь мы можем добавить строку `return unless has_admin_privileges?` в начало всех экшенов, где требуется проверка на администратора. Не идеально — но уже гораздо больше соответствует принципам DRY. В следующем разделе мы посмотрим, как выполнить эту задачу еще элегантнее, с помощью т.н. фильтров.

[1] На самом деле, "прикрутить" аутентификацию пользователей к вашему Rails-приложению можно с помощью специального гема Devise. После подключения этого гема, во всех контроллерах вам станет доступен метод `#current_user`. Также можно посмотреть скринкаст о подключении гема Devise к Rails (на русском языке).

Фильтры

Фильтры в контроллерах — это специальные методы, которые выполняются до, после или во время определенного экшена. Их можно сравнить с коллбэками в моделях. Отличие заключается в том, что в моделях коллбэки выполняются в ответ на какое-то событие. В контроллере же фильтры связаны с конкретными экшенами, причем один и тот же фильтр можно назначить нескольким экшенам. Обратимся к примерам.

В предыдущем разделе мы создали приватный метод `#has_admin_privileges?`, который вызывался внутри каждого экшена контроллера, где нам нужно было проверить, является ли пользователь администратором. Обратите особое внимание, я говорю "мы вызывали этот метод внутри экшенов", т.е. мы вручную прописывали название метода `#has_admin_privileges?` в том месте, где мы хотели его вызвать. Но мы можем избежать даже повторения этой одной, пусть и несложной строки. Назначим этот метод фильтром — для этого в контексте класса необходимо вызвать специальный метод `before_filter` и в качестве аргумента передать ему название нашего метода (в виде символа, разумеется), вот так:

```
class ItemsController < ApplicationController
  before_filter :has_admin_privileges?, only: [:new, :create,
  :edit, :update, :destroy]
  ...
end
```

В опции `:on` указан массив из названий тех экшенов, перед которыми выполнится метод `has_admin_privileges?`, или говоря несколько более грамотно — те экшены, к которым будет применен фильтр. В данном случае тех экшенов, к которым фильтр применяется в нашем контроллере больше, чем тех экшенов, к которым фильтр НЕ применяется, поэтому логичнее было бы использовать вместо опции `:on` опцию `:except`:

```
 class ItemsController < ApplicationController
  before_filter :has_admin_privileges?, except: [:index, :show]
  ...
end
```

Осталось сделать две вещи: 1) убрать из всех экшенов ручной вызов нашего метода, т.е. строку `return unless has_admin_privileges?` 2) немного изменить сам фильтр следующим образом:

```
def has_admin_privileges?
  redirect_to "/users/sign_in" unless current_user
  render(file: "public/403.html",
         status: "403.html") unless current_user.admin
end
```

Как вы могли заметить, мы убрали из метода все выражения `return`, т.к. нам больше не требуется возвращать что-либо из этого метода. Все дело в том, что Rails в данном случае не будет жаловаться на повторный render или redirect — если render или redirect происходят в фильтре, то основной экшен выполнен не будет и, значит, до всех render-ов или redirect-ов, которые в нем есть, дело тоже не дойдет.

Приведем еще один пример использования приватных методов в качестве фильтров. Наверное вы заметили, читая код в предыдущих разделах, что в `ItemsController`-е в экшенах `#show`, `#edit`, `#update` и `#destroy` есть одинаковая строка, а именно `@item = Item.find(params[:id])` и она находится в самом начале каждого из экшенов. Ну не прекрасный ли кандидат на фильтр?!

```
class ItemsController < ApplicationController

  before_filter :has_admin_privileges?, except: [:index, :show]
  before_filter :find_item,             only:   [:show, :edit,
                                                 :update, :destroy]

  ...

  private

    def find_item
      @item = Item.find(params[:id])
    end

end
```

Остается удалить строку `@item = Item.find(params[:id])` из вышеупомянутых экшенов и все будет работать точно также — только меньше кода и больше соблюдения принципа DRY. Обратите внимание на следующую особенность: мы использовали внутри метода `#find_item` *инстансную* переменную `@item`. Именно потому, что это инстансная переменная, она потом становится доступна внутри самих экшенов, к которым этот фильтр назначен.

Есть еще два типа фильтров, которые доступны в Rails-контроллерах: after_ и around_ фильтры, но т.к. они используются гораздо реже before_ фильтров, разговор о них я решил оставить за рамками этой книги [1].

[1] *After и Around фильтры в контроллерах Rails, Rails Guides*

Как правильно создать и назвать контроллер

По ходу нашего разговора и описания различных частей контроллера у читателя могло сложиться неполное впечатление о том, как же собственно правильно создавать и называть контроллер. Ниже я предлагаю небольшой чеклист, который позволит вам правильно и последовательно выполнить все действия, связанные с созданием контроллера.

1. **Придумайте название для контроллера.** Если это RESTful контроллер, т.е. контроллер, который представляет из себя коллекцию ресурсов, то следует назвать контроллер по имени ресурса во множественном числе. Например, в случае ресурса `item`, мы назовем контроллер `ItemsController`, а в случае ресурса `order` — `OrdersController`. Как уже упоминалось ранее, ресурсы в Rails приложении часто соответствуют конкретным моделям, поэтому когда вы создаете RESTful контроллер, который будет взаимодействовать с этой моделью, очень часто его название — это название модели во множественном числе + слово `Controller`. Иногда, нам потребуется создать контроллер для одиночного ресурса[1] — в этом случае название контроллера будет идти в единственном числе, например `CartController`.
2. **Запустите `rails g controller [resource_name]`** — команда, которую вам нужно запустить из терминала находясь в корневой папке приложения. [resource_name] — это имя контроллера без слова `Controller` в конце и без скобок, например `rails g controller items` (если в имени контроллера несколько слов — они все набираются в нижнем регистре и разделяются символом нижнего подчеркивания). Запущенный таким образом скрипт сгенерирует вам несколько файлов. Внимательно посмотрите какие файлы были сгенерированы. В большинстве случаев, единственными файлами, которые вам понадобятся, будут сам файл контроллера и файл с тестами для него. Например, в случае с контроллером `ItemsController` этими файлами будут `app/controllers/items_controller.rb` и `spec/controllers/items_controller_spec.rb`. Файлы со стилями и хэлперами скорее всего можно удалить — если только вы не уверены, что они вам абсолютно точно понадобятся.
3. **Добавьте route в файл `config/routes.rb`**. Если речь идет о контроллере ресурса (в современном Rails приложении это будет так в 99% случаев), скорее всего вам нужно как-минимум добавить строку `resources :resources_name_plural`, где

`:resources_names_plural` — имя ресурса во множественном числе, например: `resources :articles`

4. **Убедитесь, что вы добавили в ваш контроллер все необходимые CRUD-экшены**: `#new`, `#create`, `#edit`, `#update`, `#show`, `#index`, `#destroy`. Вам необязательно иметь все эти экшены в контролере. Например, если у вас есть контроллер, который называется `VotesController`, предназначенный для голосования, то вам нужен будет только один экшен — `#create`. Все остальные экшены скорее всего окажутся довольно бесполезными, т.к. вам вряд ли понадобится "обновлять" или "редактировать" голос (хотя это может понадобиться, если вы хотите дать пользователям возможность "передумать"). Так или иначе, ресурс не обязан иметь все CRUD-экшены, но будет неплохо помнить название и назначение каждого, перед тем как решать, нужен ли он вашему ресурсу или нет. И не забудьте, название экшена для удаления ресурса должно быть `#destroy`, а не `#delete`.

[1] В рамках этой главы мы не успели рассмотреть тему одиночных ресурсов, но в этом нет ничего сложного. С учетом полученных в этой главе знаний вы сможете самостоятельно разобраться в этой теме обратившись к документации [1]. Примером типичного одиночного ерсурса можно считать Корзину в нашем магазине (Cart). Т.к. пользователи ни при каких обстоятельствах не должны иметь возможность просматривать корзины других пользователей на сайте, и у одного пользователя может быть только одна корзина — логично сделать этот ресурс одиночным.

[1] Одиночные ресурсы, Rails Guides

Вложенные ресурсы

У вас может возникнуть желание (но в идеальном случае — необходимость) сделать в Rails приложении один ресурс логически вложенным в другой. Представим, что в нашем магазине помимо ресурса "Товары" есть также ресурс "Категории" и вы бы хотели, чтобы url всех товаров был вида `/categories/:section_id/items/:id`, то есть чтобы из url было понятно, в какой категории находится товар.

В этом случае, нам необходимо будет внести изменения в наш файл `config/routes.rb`:

```
resources :categories do

  resources :items do
    member do
      post :activate
    end
    collection do
      get :top
    end
  end
```

```
end
```

Теперь всё определение ресурса "Товары" находится внутри блока, переданного `resources :categories`. Обратившись по url `/categories/1/items/2` мы должны будем получить страницу с описанием товара с id = 2. В самом контроллере `ItemsController` мы теперь сможем узнать id категории и быстро отыскать запись о ней в БД[1]:

```
class ItemsController < ApplicationController
  ...

  def show
    @item     = Item.find(params[:id])
    @category = Category.find(params[:category_id])
  end

  ...
end
```

В данном случае, id категории (которое присутствует в нашем url) будет содержаться в `params[:category_id]` и используя его мы сможем найти категорию с помощью `Category.find`. Проблема заключается в том, что даже если данный товар не принадлежит к категории, чей id указан в url, страница с информацией о товаре все равно будет показана. Если предположить, что модель `Category` связана ассоциативной связью с моделью `Item`, то мы могли бы сделать следующую проверку в контроллере:

```
class ItemsController < ApplicationController
  ...

  def show
    @item = Item.find(params[:id])
    render_404 and return unless @item.category_id == params[:category_id]

  end

  ...
end
```

В этом случае, мы увидим страницу 404 (страница не найдена), когда произойдет попытка обратиться к странице товара по неправильному url-у, содержащему id той категории, к которой товар не принадлежит.

Следует отметить, что сам id товара не имеет отношения к категориям — в БД не может быть двух товаров с одинаковым id, даже если они принадлежат к разным категориям. Вложенность ресурсов — это понятие касающееся исключительно контроллеров и ресурсов, но никак не моделей. Именно поэтому вложенность ресурсов не всегда бывает оправданным решением. Это красиво с точки зрения url и логики, но добавляет работы программистам — заметьте, что нам пришлось писать дополнительную проверку на

принадлежность товара конкретной категории. Если вы хотите вывести информацию о категории на странице товара, вам вовсе необязательно делать вложенные ресурсы — альтернативным решением в данном случае мог бы стать вызов метода `@item.category` (этот метод доступен в объектах класса `Item`, т.к мы договорились, что они связаны с объектами класса `Category` ассоциативной связью).

[1] Данный пример подразумевает, что в нашем приложении также есть модель `Category`.

[1] Вложенные ресурсы, Rails Guides

Apllication Controller

В предыдущих примерах вы заметили, что наш контроллер `ItemsController` наследуется от класса `ApplicationController`. Вы также могли заметить, что файл `application_controller.rb` лежит в той же самой директории. А раз это так, то это значит, что разработчики Rails явно хотели, чтобы мы могли его редактировать и дополнять. Остается самый главный вопрос — зачем?

Несложно догадаться, что т.к. все другие контроллеры будут наследоваться именно от `ApplicationController`, то все методы, объявленные в нем, будут, по законам ООП, доступны и всем остальным контроллерам. И именно для этих целей нам нужен `ApplicationController` — определять методы, которые могут нам понадобиться во всех (или хотя бы в нескольких других) контроллерах.

Например, метод `has_admin_privileges?`, который мы создали в разделе Фильтры, можно было бы перенести из `ItemsController` в `ApplicationController`. При этом поведение самого `ItemsController` никак бы не изменилось — метод `has_admin_privileges?` можно было бы по прежнему использовать как фильтр. Отличие заключалось бы в том, что теперь этот метод мы смогли бы использовать и в других контроллерах — например в `OrdersController`, как фильтр к экшену `#update` — потому что мы хотим дать возможность редактировать и изменять статус заказа только администраторам, которые обрабатывают заказы. Вот как будет выглядеть `ApplicationController` после добавления этого метода:

```
class ApplicationController < ActionController::Base

  ...

  def has_admin_privileges?
    redirect_to "/users/sign_in" unless current_user
    render(file: "public/403.html",
```

```
          status: "403.html") unless current_user.admin
  end

end
```

Естественно, одноименный метод в контроллере `ItemsController` теперь можно удалить — он там больше не нужен.

Еще один пример метода, который можно было бы поместить внутрь `ApplicationController` — это метод, который отвечает за вывод страницы с ошибкой. В обычном случае, если вы хотите вывести пользователю страницу с ошибкой 403, вам придется добавить такой код:

```
render(file: "public/403.html", status: "403.html")
```

Чтобы не писать эту длинную строку каждый раз, когда мне нужно вывести ошибку 403, я предпочитаю сделать короткий метод, поместить его в `ApplicationController` и затем вызывать его каждый раз, когда он мне понадобится в других контроллерах:

```
class ApplicationController

  ...

  def render_403
    render(file: "public/403.html", status: "403.html")
  end

end
```

Этот метод можно оставить публичным — т.к. `ApplicationController` сам по себе никакие запросы не обрабатывает, то его методы по определению не являются экшенами.

Какой код должен быть в моделе, а какой в контроллере?

Новички в Rails часто совершают следующую ошибку: код, который должен быть в моделе, они помещают в контроллер. И иногда наоборот. В этом разделе я хотел бы привести несколько самых рапространенных примеров и дать общие абстрактные указания, касающиеся решения того, куда поместить тот или иной код — в модель или в контроллер.

Начнем с абстрактных правил:

- В контроллере должно быть как можно меньше кода. Если контроллер начинает расти — вам стоит либо вынести часть функционала в другой контроллер, либо (что более вероятно) — перенесети часть функционала в модель.

- Контроллер не должен влиять на логику программы. Вы должны иметь возможность полностью управлять вашей программой из Rails-консоли (`rails c`). Если вдруг вы обнаруживаете, что вы не можете выполнить какое-то действие с вашими моделями через `rails c`, но при этом то же самое действие возможно выполнить через браузер — скорее всего, ваш контроллер взял на себя слишком много обязанностей.
- Когда ваш контроллер создает или обновляет большое количество разных моделей — это не очень правильно. Модели должны быть связаны между собой ассоциативными связями. Если обновление одной модели требует обновления связанной с ней модели — это забота первой модели, но никак не контроллера.
- В целом, контроллер желательно свести к двум следующим задачам: 1) получить данные от пользователя и отправить их в модель; 2) получить ответ от модели и решить, что показать пользователю в ответ на его запрос (render или redirect).

Наконец несколько примеров того, что должно, а что НЕ должно быть заботой контроллера:

- Обычно именно в контроллере проверяют правщ того или иного пользователя совершать то или иное действие. Проверяющий код помещают в метод-фильтр и затем назначают этот фильтр различным экшенам. Именно так мы поступили с методом `#has_admin_privileges?` в одном из предыдущих разделов.
- Контроллер НЕ должен заниматься проверкой приходящих от пользователя данных (т.е. валидациями). Если вы начинаете писать условные выражения и проверять, что содержится в массиве `params` — вы поступаете неправильно. Проверкой данных занимаются модели и для этого в Rails существует набор стандартных валидаций.
- Контроллер также НЕ должен заниматься модификацией данных, пришедших от пользователя. Предположим в вашей модели `User` есть атрибут `#homepage` и вы хотите автоматически удалять из этого поля префикс `http://`, если он там есть. Не стоит делать это в контроллере модифицируя хэш `params`. Оставьте эту работу для модели — напишите свой собственный сеттер `#homepage=` и внутрь этого метода поместите код, который будет убирать ненужную часть адреса домашней страницы. Метод мог бы выглядеть так:

```
class Item < ActiveRecord::Base

  ...
```

```
  def homepage=(url)
    write_attribute :homepage, url.sub('http://', '')
  end

end
```

Понимание того, какой код следует помещать в модель, а какой — в контроллер приходит со временем и опытом. Правила, приведенные в этом разделе, помогут вам не сделать самых распространенных ошибок, но вы обязательно сделаете свои собственные, прежде чем научитесь писать код правильно.

Пример RESTful контроллеров

Ниже приведен относительно сложный пример нескольких контроллеров (пример взят НЕ из нашего воображаемого магазина, а из реального Rails-приложения). Убедитесь, что вы понимаете весь код — в нем нет почти ничего, чего бы мы не разобрали в этой главе. Исключение составляют методы `#flash` и `#t`, которые вызываются внутри экшенов. Метод `#flash` — это способ вывести сообщение пользователю после редиректа (его мы рассмотрим в главе Views, хэлперы и ассеты), а метод `#t` — это специальный метод класса `I18n`, с помощью которого в Rails можно добавить интернационализацию (поддержку нескольких языков).

Контроллер NewsMessagesController

```
class NewsMessagesController < ApplicationController

  before_filter :allow_only_admins
  before_filter :find_news_message, only: [:edit, :update, :destroy]

  def index
    @news_messages = NewsMessage.paginate(page: params[:page],
    per_page: 30)
  end

  def new
    @news_message = NewsMessage.new
  end

  def create
    @news_message = NewsMessage.create(params[:news_message])
    if @news_message.errors.empty?
      redirect_to action: "index"
    else
      render "new"
    end
  end

  def edit
  end
```

```
  def show
    redirect_to action: "edit", id: params[:id]
  end

  def update
    @news_message.update_attributes(params[:news_message])
    if @news_message.errors.empty?
      redirect_to action: "index"
    else
      render "edit"
    end
  end

  def destroy
    @news_message.destroy
    redirect_to action: "index"
  end


  private

    def find_news_message
      @news_message = NewsMessage.find(params[:id])
    end

end
```

контроллер ScreencastsController

```
class ScreencastsController < ApplicationController

  before_filter :find_section_by_subdomain
  before_filter :allow_only_section_owners_and_admins, :except
  => [:show, :index]
  before_filter :find_screencast, :only => [:show, :edit, :update,
  :destroy]

  def index
    order = (params[:desc] ? "DESC" : "ASC")
    @screencasts = Screencast.where(section_id: @section.id, published:
    true).order("created_at #{order}").paginate(per_page: 10,
    page: params[:page])


    if params[:format] == "rss"
      @pro_screencasts = StoreItem.where(section_id: @section
      id, active: true, item_type: "Screencast").order("created_at
      #{order}").limit(5)
      @screencasts += @pro_screencasts
      if order == "DESC"
        @screencasts.sort! { |x,y| y.created_at => x.created_at }
      else
        @screencasts.sort! { |x,y| x.created_at => y.created_at }
      end
      @screencasts = @screencasts[0..9]
    end

  end
```

```
def show
  if @screencast.published || @section.user == current_user
  || current_user.is_admin?
    @latest_questions = Question.where(screencast_id: @screencast
    id).includes(:user).order("created_at DESC").limit(10)
  else
    render_404
  end
end

def new
  @screencast = Screencast.new
  @screencast.number = Screencast.where(section_id: @section
  id).order("created_at DESC").first.try(:number)
  @screencast.number = @screencast.number ? @screencast.number
  + 1 : 1
end

def create
  @screencast = Screencast.create(params[:screencast].merge({section:
  @section, user: current_user}))
  if @screencast.errors.empty?
    flash[:success] = t("controllers.screencasts.success.create")
    redirect_to @screencast
  else
    flash[:error] = t("controllers.screencasts.errors.create")
    render "new"
  end
end

def edit
end

def update
  @screencast.update_attributes(params[:screencast])
  if @screencast.errors.empty?
    flash[:success] = t("controllers.screencasts.success.update")
    redirect_to @screencast
  else
    flash[:error] = t("controllers.screencasts.errors.update")
    render "edit"
  end
end

def destroy
  @screencast.destroy
  flash[:success] = t("controllers.screencasts.success.destroy")
  redirect_to screencasts_path
end

private

  def find_screencast
    @screencast = Screencast.where(section_id: @section.id,
    number: params[:id]).first
    render_404 unless @screencast
  end

  def find_section_by_subdomain
```

```
      render_404 unless super
    end

end
```

Глава 7. Views, хэлперы и ассеты

В главе Контроллеры и ресурсы мы уже начали немного работать в шаблонами, но в этой главе мы основательно разберемся с тем, что же это такое и как устроены шаблоны.

Что такое шаблон и из чего он состоит?

Шаблоны — это файлы, которые находятся в Rails-приложении в папке `app/views`. И, конечно, шаблоны — это буква V (views, представление) в аббревиатуре MVC. Они отвечают за вывод информации на экран пользователю. Каждый шаблон — это файл, имя которого заканчивается на название того или иного *шаблонизатора*. Например, имя шаблона может заканчиваться на `.erb` или `.haml`.

Шаблонизатор — это составляющая часть фреймворка Ruby On Rails. По сути шаблонизатором называют парсер, который читает файлы шаблонов, написанные по правилам, понятным именно этому шаблонизатору, и генерирует из них тот или иной результат. Например, если вы используете шаблонизатор Erb (шаблонизатор по умолчанию), то он будет читать все файлы с расширением `.erb`, которые лежат в папке `app/views`, и выдавать тот или иной результат. Результатом является формат, понятный определенной программе, например браузеру.

Рассмотрим шаблон, выводящий информацию о товаре, из главы Контроллеры и ресурсы, файл `app/views/item/show.html.erb`:

```
<h1><%= @item.title %></h1>
<p>Описание: <%= @item.description %></p>
<p>Цена: <b><%= @item.price %></b></p>
```

Во-первых обратим внимание на название файла — у него двойное расширение `.html.erb`. Это нужно, чтобы объяснить Rails, что этот файл должен рендериться только в ответ на обычные запросы браузера, которые назад ожидают получить html-страницу[1]. Когда вы создаете шаблоны, называйте их именно двойным расширением — не забывайте добавлять `.html` перед `.erb`

Теперь, если мы посмотрим на, непосредственно, содержимое шаблона, мы заметим, что у нас там явно присутствуют html-тэги вперемешку с не совсем понятными и странно выглядящими тэгами <% %> и <%= %>. Это так называемые erb-тэги, внутри которых можно поместить любой Ruby-код. Разница между двумя вариантами записи тэга — со

знаком равенства и без — заключается в том, что в первом случае результат выполнения кода будет подставлен в шаблон. То есть, вышеприведенный код скомпилируется в следующую html-страницу (подразумевается, что в переменной `@item` содержится объект класса `Item`):

```
<h1>Плюшевый крокодил</h1>
<p>Описание: зеленый, мягкий, большой</p>
<p>Цена: <b>1500</b></p>
```

В случае, если в первой строке мы заменим <%= %> на <% %>, и код в ней будет выглядеть так

```
<h1><% @item.title %></h1>
```

то в результирующем html содержимое заголовка останется пустым:

```
<h1></h1>
<p>Описание: зеленый, мягкий, большой</p>
<p>Цена: <b>1500</b></p>
```

Возникает вопрос: зачем же нужны erb-тэги, в которых результат выполнения Ruby-кода не подставляется в шаблон? Обратимся к еще одному примеру из главы Контроллеры и ресурсы, где мы выводили список всех товаров в шаблоне `app/views/items/index.html.erb`:

```
<ul>
  <% @items.each do |i| %>
    <li><%= link_to i.title, i %></li>
  <% end %>
</ul>
```

Как видно из этого примера, erb тэг без знака равенства используется здесь в строке, где мы вызываем метод `#each`, а также в строке, где мы закрываем блок, переданный методу `#each`. Это делается потому, что в этой строке метод `#each` сам по себе не возвращает никакой полезной информации, которую мы могли бы вывести в шаблон. Полезная информация — каждый следующий найденный товар — оказывается в переменной `i` внутри блока. И именно внутри блока мы используем erb-тэг со знаком равенства, чтобы вывести информацию о товаре.

Erb-шаблон не обязательно должен содержать html. Например для отправки почты из приложения мы можем использовать текстовые шаблоны. Представим, что нам нужно отправить подтверждение о том, что заказ принят, пользователю на email[2]. Тогда шаблон мог бы называться `order_created.text.erb` и его содержимое могло бы выглядеть так:

```
Уважаемый <%= @order.user.name %>,
```

```
Ваш заказ №<%= @order.id %> принят в обработку. Спасибо!
```

Таким образом, любой шаблон в Rails-приложении состоит из следующих частей: содержимого в требуемом клиентом формате (обычно — html) и Ruby-кода. При этом желаемые части Ruby-кода могут быть заменены на то, во что этот код вычисляется, и подставлены в шаблон.

Стоит также отметить, что в современных Rails-приложениях разработчики по большей части используют шаблонизатор haml. В образовательных целях, в этой книге мы будем использовать erb, т.к. его в любом случае нужно уметь использовать и он нагляднее демонстрирует html-код. Тем не менее, если вам интересно научиться работать с haml, я рекомендую вам прочитать раздел Почему haml лучше чем erb?.

[1] Бывают еще и асинхронные запросы, которые браузер можно заставить сделать с помощью Javascript-кода. В таких запросах вы, как правило, хотите получить назад json, а не html.

[2] В этой книге мы не рассматриваем вопрос отправки email-сообщений из Rails приложений. Для того, чтобы научиться это делать, вам необходимо разобраться как работать с `ActionMailer`-ом [1].

[1] *ActionMailer basics, Rails Guides (на английском)*

Хэлперы и почему в шаблонах не должно быть слишком много ruby-кода

Создание хэлперов

Иногда нам требуется добавить относительно сложную логику в шаблон. Рассмотрим случай, когда мне необходимо в шаблоне `items/show.html.erb` показывать ссылку на страницу редактирования товара. Но делать это нужно только в том случае, если у пользователя есть права на редактирование товара. Предположим также, что редактировать товар могут не только администраторы сайта, но также администраторы конкретного раздела (модель `Category`) магазина. Код во шаблоне мог бы выглядеть следующим образом:

шаблон items/show.html.erb

```
<h1><%= @item.title %></h1>
<p>Описание: <%= @item.description %></p>
<p>Цена: <b><%= @item.price %></b></p>
<% if current_user  (
      current_user.admin ||
      @item.category.administrator == current_user
      ) %>
  <%= link_to "Редактировать", edit_item_path(@item) %>
```

```
<% end %>
```

Из-за одной ссылки мы сделали наш шаблон гораздо менее читабельным и добавили в него большое количество Ruby-кода! Именно в таких случаях нам следовало бы вынести код в шаблоне в специальный метод, который бы возвращал нам `true` или `false`. Разве не удобне было бы написать в шаблоне нечто подобное:

```
<h1><%= @item.title %></h1>
<p>Описание: <%= @item.description %></p>
<p>Цена: <b><%= @item.price %></b></p>
<%= link_to "Редактировать", edit_item_path(@item) if can_edit_item?(@item)
%>
```

Теперь остается понять, откуда должен взяться этот самый метод `#can_edit_item?`. Такие методы, которые мы вызываем в шаблоне, называют *хэлперами*. Чтобы создать хэлпер, вам необходимо зайти в папку `app/helpers` и открыть файл, соответствующий названию того контроллера, в шаблонах которого вы будете вызывать хэлпер (или открыть файл `application_helper.rb`, если вы хотите, чтобы хэлпер был доступен в любом шаблоне вашего приложения). Например, в нашем случае, мы будем вызывать хэлпер `#can_edit_item?` в шаблоне `items/show.html.erb`, рендерингом которого занимается контроллер `ItemsController`. Следовательно, нужно открыть файл `app/helpers/items_helper.rb` (или создать его командой `rails g helper items`). Внутри вы увидите следующий код:

```
module ItemsHelper
end
```

Прямо внутри этого модуля и следует создать метод `#can_edit_item?`:

```
module ItemsHelper

  def can_edit_item?(item)
    current_user  (
    current_user.admin ||
    item.category.administrator == current_user)
  end

end
```

Поскольку внутри метода мы использовали логические операторы `&&` и `||`, то выражение (а следовательно и сам метод) вернет нам либо значение `true`, либо значение `false`. Это именно то, чего мы добивались, и теперь наш код в шаблоне заработает. Более того, теперь мы можем использовать этот метод в других местах шаблона или даже в других шаблонах внутри папки `app/views/items/`. Например, в случае, если нам нужно узнать, сколько было сделано заказов с этим товаром (и показывать эту информацию только администраторам

раздела или всего магазина):

```
<h1><%= @item.title %></h1>
<p>Описание: <%= @item.description %></p>
<p>Цена: <b><%= @item.price %></b></p>
<%= link_to "Редактировать", edit_item_path(@item) if can_edit_item?(@item)
%>
<%= "Куплено товаров: #{@item.orders.count}"       if can_edit_item?(@item)
%>
```

Надо также сказать, что внутри хэлперов доступны все те же объекты и методы, которые доступны и в самом шаблоне (об этом подробнее читайте в разделе Как views связаны с контроллером?)

Предопределенные хэлперы

Во всех шаблонах вам доступны уже заранее заготовленные хэлперы, написанные для вас авторами фреймворка Ruby On Rails. Здесь мы поговорим о самых распространеных двух из них.

link_to генерирует за вас html-ссылку, т.е. тэг `<a>` с атрибутами и содержанием. Вы, безусловно, могли бы создать ссылку вручную написав необходимый html, но делать это не рекомендуется. Дело в том, что хэлпер `#link_to` берет на себя заботу по генерированию правильного url-а для каждой ссылки, а вы не всегда можете его помнить; или же динамическую часть url-а может быть неудобно вставлять в тэг `<a>`. Рассмотрим два варианта создания ссылки на страницу с товаром — сначала вручную:

```
<a href="/items/#{@item.id}"><%= @item.title %></a>
```

а затем с помощью хэлпера `#link_to`

```
<%= link_to @item.title, @item %>
```

Согласитесь, что вариант номер 2 — несколько более лаконичен. И, как можно догадаться, метод `#link_to` принимает 2 обязательных аргумента: первый аргумент является текстом ссылки, а второй аргумент — это путь, который окажется в атрибуте `href`. В нашем случае, в качестве второго аргумента мы передали не путь, а объект класса `Item`, но умный хэлпер автоматически понял, что мы от него хотим, и превратил второй аргумент в `item_path(@item)`; то есть приведенный выше код эквивалентен следующему коду:

```
<%= link_to @item.title, item_path(@item) %>
```

Хэлпер image_tag генерирует для вас тэг `img` с правильным путем к изображению в атрибуте тэга `src`. При этом, само изображение должно находиться в папке

`app/assets/images`. На первый взгляд, могло бы показаться, что несложно будет добавить изображение, вручную прописав тэг, например так:

```
<img src="/assets/logo.png"/>
```

Однако попробуем сделать это с помощью хэлпера `image_tag`:

```
<%= image_tag "logo.png" %>
```

Если бы вы запускали приложение в production-среде на сервере, вы бы увидели следующий html-код в том месте, где должна быть картинка:

```
<img src="/assets/logo-87ee0c5671ea23d5c33c8ae207c418a3.png"/>
```

Как вы видите, к названию файла добавилось длинное сочетание букв и цифер. Это произошло потому, что Rails генерирует уникальные имена файлов каждый раз, когда содержимое файлов меняется. Уникальные имена нужны затем, чтобы если вы в какой-то момент решите обновить логотип, браузер не продолжал показывать посетителям вашего сайта старую закэшированную версию. И хэлпер `#image_tag` автоматически подставляет правильное имя файла. Если бы вы написали тэг `<img<` вручную, браузер показал бы пустое место, а не изображение, т.к. на самом деле файла с таким именем на сервере не существовало бы.[1]

Хэлперы, доступные в контроллере

Иногда бывает полезно сделать хэлперы доступными и в контроллерах, и в шаблонах. В этом случае, нам нужно перенести сам метод в контроллер и затем объяснить контроллеру, что этот метод нужно также рассматривать как хэлпер (и, следовательно, сделать его доступным в шаблонах).

Например, если вы помните, в нашем контроллере `ItemsController` был такой код:

```
class ItemsController < ApplicationController

  ...

  private

    def has_admin_privileges?

      unless current_user
        redirect_to "/users/sign_in" and return
      end

      unless current_user.admin
        render(file: "public/403.html", status: "403.html") and return
```

```
      end

      true
    end

end
```

Метод `#has_admin_privileges?` являлся фильтром, который рендерил страницу 403, в случае если пользователь пытался отредактировать товар, не будучи при этом администратором. Предлагаю немного переименовать этот метод, чтобы не спутать с нашим хэлпером:

```
class ItemsController < ApplicationController

  ...

  private

    def disallow_access_if_no_editing_rights

      unless current_user
        redirect_to "/users/sign_in"
      end

      unless current_user.admin
        render(file: "public/403.html", status: "403.html")
      end

    end

end
```

и затем перенести наш хэлпер `#can_edit_item?` в контроллер:

```
class ItemsController < ApplicationController

  ...

  private

    def can_edit_item?(item)
      current_user && (
      current_user.admin ||
      item.category.administrator == current_user)
    end

    def disallow_access_if_no_editing_rights
      unless current_user
        redirect_to "/users/sign_in"
      end

      unless can_edit_item?(@item)
        render(file: "public/403.html", status: "403.html")
      end
    end
```

```
end
```

Обратите внимание, как вместо `unless current_user.admin` мы написали `unless can_edit_item?(@item)`

Теперь остается сделать две вещи: 1) убедиться, что метод `#find_item` вызывается в контроллере ДО метода `#disallow_access_if_no_editing_rights` (иначе переменная `@item` будет пустая) и 2) объявить метод `#can_edit_item?` хэлпером. Вот как будет выглядеть наш контроллер после выполнения этих двух шагов:

```
class ItemsController < ApplicationController

  # ------------------------------------------------------------
  # Сначала find_item, затем disallow_access_if_no_editing_rights
  # ------------------------------------------------------------
  before_filter :find_item,
    on: [:show, :edit, :update, :destroy]
  before_filter :disallow_access_if_no_editing_rights,
    except: [:index, :show]


  # ------------------------------------------------
  # Делаем доступным метод can_edit_item? в шаблонах
  # ------------------------------------------------
  helper_method :can_edit_item?

  ...

  private

    def can_edit_item?(item)
      current_user && (
      current_user.admin ||
      item.category.administrator == current_user)
    end

    def disallow_access_if_no_editing_rights
      unless current_user
        redirect_to "/users/sign_in"
      end

      unless can_edit_item?(@item)
        render(file: "public/403.html", status: "403.html")
      end
    end

end
```

[1] Подробнее об этой особенности вывода изображений в Rails — Asset Pipeline — можно почитать в разделе Assets

[1] *Документация по хэлперу #link_to, Rails API*

[2] *Документация по хэлперу #image_tag, Rails API*

Как views связаны с контроллером?

В главе Контроллеры и ресурсы мы уже немного коснулись того, как работают views. Мы, например, выяснили, что шаблоны из папки `app/views` рендерятся, если мы явно вызываем метод `#render`, либо если экшен завершает свое выполнение и в соответвующей контроллеру папке внутри папки `app/views` есть файл с соответствующим экшену названием. Мы также выяснили, что инстансные переменные (переменные начинающиеся на символ @), которые были определены в контроллере — доступны и шаблонах, и их можно использовать для вывода тех или иных данных. В этом разделе мы рассмотрим все эти правила несколько подробней, а также познакомимся с некоторыми другим правилами.

Но начать следует с более абстрактного вопроса — как же именно views связаны с контроллером? Вам следует помнить две вещи: контроллер решает какой шаблон должен быть отрендерен, а шаблон, в свою очередь, решает, к какому контроллеру послать запрос (вернее сказать, это решает содержимое шаблона — ссылки и формы). Контроллер не имеет практически никакого влияния на то, какие данные будут отображены во views. Даже несмотря на то, что шаблонам доступны инстансные переменные из контроллера, в которых обычно содержатся интересующие шаблон модели, никто не мешает вам сделать запрос к базе данных и получить массив из моделей прямо в шаблоне, например:

```
<% Item.all.each do |item| %>
  <p><%= item.title %></p>
<% end %>
```

Этот код выведет нам заголовки всех товаров на одной странице. Тем не менее, поступать таким образом не стоит. Этот пример был призван показать, что шаблон относительно независим от контроллера, но большая степень независимости может нарушить MVC-структуру приложения.

Метод #render

Когда контроллер решает, какой шаблон отрендерить, он прежде всего полагается на явно вызванный внутри экшена метод `#render` — мы делали так несколько раз в главе Контроллеры и ресурсы. Метод `#render` — это довольно умный метод, который может обойтись без дополнительной информации совсем: если не передать ему ни одного аргумента, он автоматически отрендерит шаблон, который соответствует данному контроллеру и экшену (т.е. шаблон `app/views/[controller_name]/[action_name]`).

Мы также можем передать методу `#render` неполный путь к шаблону, например `render "new"`. Если этот код был вызван где-либо в контроллере `ItemsController`, то метод

`#render` автоматически поймет, что ему требуется отрендерить шаблон `new` из папки `app/views/items/`. То есть, мы с таким же успехом могли бы написать `render "items/new"` и в работе нашей программы ничего бы не изменилось.

Единственный случай, когда нам может понадобиться указать полный путь к шаблону — если мы хотим отрендерить шаблон, соответствующий одному контроллеру из другого контроллера. Это бывает крайне редко и, по большей части, относится к вызову метода `#render` не из контроллера, а из другого шаблона (например, когда мы хотим отрендерить паршиал[1]).

Следует также упомянуть, что методу `#render` можно указать не шаблон, а экшен. Например:

```
render action: "new"
```

В этом случае важно понимать, что экшен `#new` не будет выполнен. Мы всего лишь объясняем методу `#render`, что необходимо отрендерить шаблон, по умолчанию ассоциированный с экшеном `#new`. Метод `#render` не имеет суперсилы самостоятельно вызывать другие экшены в контроллере.

Какие данные доступны внутри шаблона?

Внутри каждого шаблона у программиста есть доступ к разного рода объектам. Рассмотрим их:

- Все инстансные переменные, которые были объявлены в контроллере в процессе выполнения текущего экшена доступны и в шаблоне. При этом важно отметить, что переменные из других экшенов будут недоступны. Например, если у нас есть запрос к экшену `#show`, который создал переменную `@item` с товаром, то она будет доступна в том шаблоне, который будет отрендерен этим экшеном. Однако, переменная `@items` (множественное число!) из экшена `#index`, содержащая массив товаров, доступна не будет — просто по той причине, что экшен `#index` не был выполнен и переменная определена не была.
- Хэш `params` также доступен внутри шаблонов. Однако, будьте осторожны! Если вы напрямую выводите содержимое хэша `params` в какой либо html-элемент, вы рискуете дефейсом страницы. Т.к. пользователь может передать в этот хэш что угодно (с помощью переменных адресной строки), полученная им страница может выглядеть не так, как вы предвидели. Это вряд ли представляет угрозу безопасности для вашего приложения, но может быть неприятным сюрпризом для пользователя, который по ошибке введет в форму не те данные, которые вы

ожидали от него получить.

- Динамические хэлперы для генерирования пути к контроллерам Rails доступны как в самих контроллерах, так и во views. Именно благодаря одному из таких хэлперов мы можем создать ссылку на конкретный товар, которая отправляет запрос в экшен `ItemsController#show`:

  ```
  <%= link_to @item.title, item_path(@item) %>
  ```

  В этом примере хэлпер `item_path` создал путь `/items/1` (мы подразумеваем, что id товара в данном случае — 1)

- Стандартные хэлперы Rails, самые распространенные из которых — это:
  - **link_to** — генерирует html-код ссылки
  - **image_tag** — генерирует html-код для вставки изображения в страницу
  - **form_for** — генерирует html-код формы

- В шаблонах также доступны все классы, которые есть в нашем Rails приложении. Это означает, что мы можем без проблем написать в нашем шаблоне, например, такой код:

  ```
  <%= Item.last.title %>
  ```

  который выведет заголовок последнего найденного в БД товара.

[1] Подробнее о том, что такое *паршиал* и как его рендерить читайте в разделе Паршиалы

Лэйауты

Помимо непосредственно информации связанной с конкретным ресурсом (например с товаром в магазине), сайты обычно демонстрируют на странице и другие элементы. Как правило, часть из этих элементов является одинаковой для всех страниц сайта — например шапка и футер обычно остаются одинаковыми на всех страницах сайта. Именно для этих целей — вывода одинаковых частей страницы — в Rails приложении существуют т.н. *лэйауты* (layouts). Лэйаутами называют специальные шаблоны, которые находятся в папке `app/views/layouts`, и по умолчанию в любом Rails приложении есть хотя бы один лэйаут, находящийся в файле `app/views/layouts/application.html.erb`. Откройте этот файл и вы увидите примерно следующее:

```
<!DOCTYPE html>
<html>
  <head>
    <title>StoreApp</title>
    <%= stylesheet_link_tag "application", :media => "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
  </head>
  <body><%= yield %></body>
</html>
```

Обратите внимание на строку, в которой вызывается метод `#yield`[1] -это именно то место, куда будет вставляться содержимое любого шаблона, который рендерится вашим приложением. То есть получается, что содержимое любого шаблона будет окружено html-тэгами `<body></body>`

Мы можем немного исправить наш лэйаут, чтобы на каждой странице сайта выводилась шапка с информацией о текущем пользователе или предложением залогиниться. Это можно было бы сделать следующим образом[2]:

```
<!DOCTYPE html>
<html>
  <head>
    <title>StoreApp</title>
    <%= stylesheet_link_tag "application", :media => "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <header>
      <% if current_user %>
        Вы залогинены как <b><%= current_user.login %></b>.
      <% else %>
        Попробуйте <%= link_to "войти", "/sign_in" %> или
        <%= link_to "зарегистрироваться", "/sign_up" %>.
      <% end %>
    </header>
    <%= yield %>
  </body>
</html>
```

Теперь сверху на каждой странице будет выводиться информация о текущем пользователе. Несложно догадаться, что таким же способом можно добавить в лэйаут что угодно.

Еще один интересный момент заключается в том, что в лэйауте доступны все те же самые объекты и методы, которые доступны в вашем шаблоне. Мы можем использовать это, чтобы изменить заголовок (title) каждой страницы. Например сейчас, если вы посмотртите на содержимое тэгов `<title></title>` вы увидите там только текст "StoreApp", который будет одинаковым для всех страниц. Это не очень хорошо (особенно с точки зрения поисковых систем и SEO) и было бы неплохо, чтобы заголовок был уникальным для каждой страницы сайта. Мы можем добиться этого, используя инстансную переменную, например так:

```
<title><%= @page_title %> | StoreApp</title>
```

и затем присвоив значение этой инстансной переменной в каждом отдельном шаблоне, например в шаблоне `app/items/show.html.erb`:

```
<h1><%= @page_title = @item.title %></h1>
<p>Описание: <%= @item.description %></p>
<p>Цена: <b><%= @item.price %></b></p>
```

Здесь мы убили двух зайцев сразу: вывели название товара между тэгами `<h1></h1>` и также записали название товара в переменную `@page_title`, которую мы затем используем в лэйауте. Если название нашего товара — "Плюшевый крокодил", то между тэгами `<title></title>` мы получим следующий результат:

```
<title>Плюшевый крокодил | StoreApp</title>
```

Присвоить значение инстансной переменной `@page_title` мы, естественно, можем не только в шаблоне. Мы могли бы вместо этого установить эту переменную в экшене `ItemsController#show` — переменная все равно была бы доступна в лэйауте.

Использование альтернативных лэйаутов

В каких-то случаях вам может понадобиться использовать различные лэйауты для разных частей сайта. Например, администраторам вы хотели бы показывать одну шапку, простым пользователям — другую. Нет ничего проще. Создадим файл `app/views/layouts/admin.html.erb` и затем укажем методу `#render`, что нужно использовать именно этот лэйаут:

```
class ItemsController < ApplicationController

  ...

  # Этот экшен доступен только администраторам,
  # значит рендерим его в администраторском лэйауте.
  def edit
    render layout: "admin"
  end

end
```

Иногда вам может понадобиться, чтобы все экшены в контроллере рендерили шаблоны с другим лэйаутом. В этом случае не очень удобно указывать метод `#render` явно внутри каждого экшена. Предположим, что у нас есть контроллер `AdminsController`, доступный только администраторам. В этом случае, чтобы все экшены рендерили шаблоны в лэйауте

"admin", достаточно вызвать метод класса `.layout`:

```
class AdminsController < ApplicationController

  layout "admin"
  ...

end
```

[1] Если вы не знаете, в каких случаях в языке Ruby используется метод `#yield`, вам стоит основательно разобраться в том, что такое *блоки в языке Ruby*.

[2] Метод `#current_user`, который мы использовали в этом примере, будет доступен вам, только если вы установите гем `Devise` для модели `User`.

Паршиалы

Паршиалы — это специальные шаблоны, имена файлов которых начинаются на символ нижнего подчеркивания. В главе Контроллеры и ресурсы мы уже создавали паршиал `items/_form.html.erb`. Как правило паршиалы необходимы для того, чтобы можно было вставлять их в несколько разных других шаблонов, соблюдая таким образом принцип DRY и не повторяя код в обоих шаблонах. Так, паршиал с формой мы вставляли в шаблоны `new.html.erb` и `edit.html.erb`. В этом разделе я приведу еще один небольшой пример использования паршиалов для закрепления пройденного.

В нашем магазине мы сделали так, что список всех товаров выводится экшеном `ItemsController#index` с использованием шаблона `items/index.html.erb`. Возможно вам также хотелось бы выводить товары на главной странице магазина — но не все, а всего-лишь три самых последних добавленных товара (новинки). Быстро создадим для этой цели контроллер `PagesController` и добавим туда экшен `#frontpage`, который и будет экшеном, ответственным за главную страницу сайта:

```
class PagesController

  def frontpage
    @latest_items = Item.order('created_at DECS').limit(3)
  end

end
```

Затем добавим в `config/routes.rb` строку, которая объяснит Rails, что за главную страницу сайта отвечает только что созданный нами экшен `PagesController#frontpage`:

```
StoreApp::Application.routes.draw do
  ...
  root :to => "pages#frontpage"
```

```
end
```

И, наконец, создадим соответствующий шаблон для главной страницы `app/views/pages/frontpage.html.erb`:

```
<h1>Добро пожаловать в наш магазин!</h1>
Новинки:

<ul>
  <% @latest_items.each do |i| %>
    <li><%= link_to i.title, i %></li>
  <% end %>
</ul>
```

Приведенный выше код, безусловно, будет работать, и вы увидите главную страницу со списком из последних трех добавленных в магазин товаров. Однако, сложно не заметить, что часть шаблона главной страницы, где методом `#each` выводятся товары, очень напоминает такую же часть из шаблона `items/index.html.erb`. И не просто напоминает — единственная разница между ними — это название переменной. Взгляните на шаблон `items/index.html.erb`:

```
<h1>Все товары</h1>
<p>
  Сортировка:
  <%= link_to "Новые сверху",  items_path(order: "desc") %> |
  <%= link_to "Старые сверху", items_path(order: "asc")  %>
</p>
<ul>
  <% @items.each do |i| %>
    <li><%= link_to i.title, i %></li>
  <% end %>
</ul>
<%= will_paginate @items %>
```

Это как раз тот случай, когда нам пригодится паршиал. Вынесем общую часть в паршиал `items/_items_list.html.erb`:

```
<ul>
  <% items.each do |i| %>
    <li><%= link_to i.title, i %></li>
  <% end %>
</ul>
```

Обратите внимание, как я сделал переменную `items` локальной — это важно! Теперь нам остается вставить этот паршиал в оба наших шаблона. В `pages/frontpage.html.erb`:

```
<h1>Добро пожаловать в наш магазин!</h1>
Новинки:
<%= render partial: "items/items_list", locals: { items: @latest_items
} %>
```

и в `items/index.html.erb`:

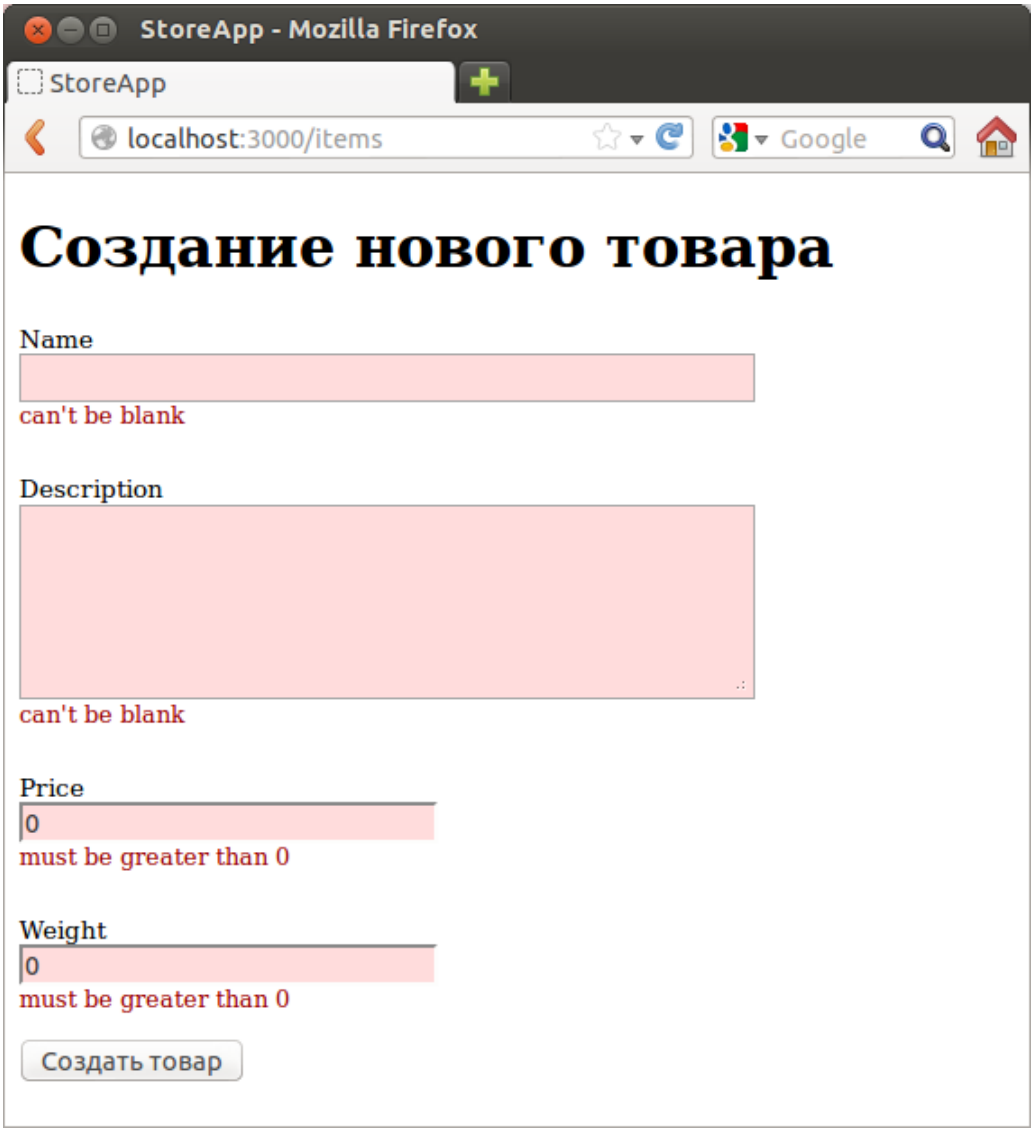
```
<h1>Все товары</h1>
<p>
  Сортировка:
  <%= link_to "Новые сверху",  items_path(order: "desc") %> |
  <%= link_to "Старые сверху", items_path(order: "asc")  %>
</p>
<%= render partial: "items/items_list", locals: { items: @items
} %>
<%= will_paginate @items %>
```

Методу `#render` передаются две опции. Первая — `:partial` — устанавливает путь к паршиалу, причем само название паршиала указывается без символа нижнего подчеркивания. И опция `:locals` устанавливает локальные переменные в паршиале. Т.к. внутри паршиала мы используем локальную переменную `items`, нам нужно было установить ее с помощью переданного этой опции хэша. При этом, мы не могли просто использовать в паршиале инстансную переменную `@items`, т.к. такая переменная существовала бы только в случае обращения к экшену `ItemsController#index`, в то время как в экшене `PagesController#frontpage` инстансная переменная, которая содержит товары, называется `@latest_items`.

Паршиалы — мощный и простой в использовании инструмент, который позволит вам избежать дублирования кода. Если вы обнаруживаете, что начали копировать код из одного шаблона в другой — задумайтесь, а не тот ли это случай, когда вам следует вынести этот код в паршиал.

Формы и гем simple_form

В главе Контроллеры и ресурсы мы уже делали форму для создания товаров. Но тогда я сказал, что мы еще не умеем красиво выводить ошибки в случае, если валидации в модели не проходят. В идеале, мы хотим, чтобы наша форма с ошибками выглядела следующим образом:



К сожалению, стандартными средствами Rails, этого добиться довольно сложно, поэтому в этом разделе мы рассмотрим очень популярный гем `simple_form` [1], который я и рекомендую вам использовать. Для установки гема добавьте строку `gem 'simple_form'` в файл `Gemfile` и запустите в терминале команду `bundle install`.

После этого нам необходимо будет поправить три наших файла, в которых содержится форма для создания товара:

файл app/views/items/new.html.erb

```
<h1>Создание нового товара</h1>
<%= simple_form_for @item do |f| %>
  <%= render partial: "form", locals: { f: f } %>
  <%= f.submit "Создать товар" %>
<% end %>
```

файл app/views/items/edit.html.erb

```
<h1>Редактирование товара</h1>
<%= simple_form_for @item do |f| %>
  <%= render partial: "form", locals: { f: f } %>
  <%= f.submit "Сохранить товар" %>
<% end %>
```

файл app/views/items/_form.html.erb

```
<%= f.input :name,        required: :optional %>
<%= f.input :description, required: :optional %>
<%= f.input :price %>
<%= f.input :weight %>
```

В первых двух файлах мы просто поменяли название хэлпера `#form_for` на `#simple_form_for` — это необходимо сделать, если вы хотите, чтобы внутри блока мы могли использовать именно хэлперы гема `simple_form`, а не стандартные хэлперы Rails.

Затем, в третьем файле, где у нас собственно перечислены поля формы, мы использовали специальный метод `f.input` — он сгенерирует нам html для конкретного поля формы, название которого мы передали ему в виде символа в первом аргументе. Причем тип поля — текстовое поле, поле для пароля, textarea, выпадающий список, checkbox и т.д. — будет определен автоматически. В нашем случае, первое поле будет текстовыми полем; и html, сгенерированный для него, будет выглядеть так:

```
<div class="input string optional">
  <label class="string optional" for="item_name">Name</label>
  <input class="string optional" id="item_name" maxlength="255"
  name="item[name]" size="50" type="text"/>
</div>
```

Несложо заметить, что помимо самого тэга `<input>` появился еще тэг `<label>`, и оба этих тэга обернуты в `<div>`. Это сделано благодаря гему `simple_form` и поможет нам чуть позже задать такие css-стили для формы, чтобы она выглядела именно так, как на иллюстрации выше.

Второе поле, `description`, превратится в следующий html-код:

```
<div class="input text optional">
  <label class="text optional" for="item_description">
    Description
  </label>
  <textarea class="text optional" cols="40" id="item_description"
  name="item[description]" rows="20">
  </textarea>
</div>
```

Для обоих полей мы использовали опцию `required: optional`. Это сделано исключительно в демонстрационных целях. Дело в том, что если ее не указать, то `simple_form` добавит к тэгам `input` атрибут, который проинструктирует браузеры об обязательности заполнения этого поля. Таким образом, если мы нажмем кнопку "Создать товар" не заполнив поля, браузер не сделает запрос к серверу и не отправит форму. Вместо того, что было показано выше на иллюстрации, вы увидите стандартное сообщение браузера. На мой взгляд, это не очень хорошая идея, особенно для понимания того, как работают формы, поэтому хотя бы на время мы оставим опцию `required: optional`. В любом случае, я надеюсь вы поэкспериментируете с ней самостоятельно. Главное, что нужно понимать — эта опция никаким образом не влияет на валидации в моделе![1]

Оставшиеся два поля будут иметь атрибут `type="number"`, который был автоматически им присвоен, т.к. умный гем `simple_form` увидел, что тип данных для этих полей в БД — `float`.

Теперь, чтобы заставить нашу форму выглядеть именно так, как на иллюстрации, добавьте в файл `app/assets/stylesheets/application.css.scss`[2] следующий код:

```
$colors_red         : &960000;
$colors_pink        : &ffb8b8;
$colors_gray        : &aaa;
$colors_light_gray  : &a7a7a7;
$colors_text        : &000;

.input {
  margin-top: 1.5em; clear: both; line-height: 100%;
  label { display: block; font-size: 85%; abbr { display: none; }; }
  textarea { height: 100px; line-height: 120%; font-size: 80%; }
}
.hint { color: $colors_gray; font-size: 80%; display: block; line-height:
100%; margin-bottom: 1em; }
input.string, input.password, textarea { font-size: 1em; padding:
3px; width: 100%; display: inline-block; border: solid 1px $colors_light_gray;
}

.field_with_errors {
  input, textarea { background-color: &ffdcdc; }
  .hint { margin-bottom: 0; }
}

.error_notification { display: none; }
```

```
form {

  margin-bottom: 0 0 1em 0; width: 400px; padding-right: 10px;
  .error { color: $colors_red; font-size: 80%; margin-bottom:
  1em; clear: both; display: block; }
  label { color: $colors_text; }

}
```

Если вы не до конца понимаете этот код, то вполне вероятно, что вам следует познакомиться с языками CSS и SCSS [1]. Мы немного обсуждаем их далее в этой главе, но это обсуждение требует минимального понимания CSS.

Теперь, оставив первые два поля формы пустыми и добавив число 0 в последние два поля, вы можете нажать кнопку "Создать товар" и, наконец, вы должны будете увидеть именно ту форму с явной демонстрацией ошибок, которая была показана на иллюстрации в начале раздела.

Гем `simple_form` берет на себя задачу добавить в форму всю необходимую и полезную информацию (включая подсказки под каждым из полей и сообщения об ошибках в случае непрошедших валидаций) и оставляет за программистом полное право добавить для сгенерированного им html любые стили (css), чтобы форма могла выглядеть именно так, как задумал дизайнер сайта. Приведенный выше css — это лишь один из самых обычных вариантов. Вы можете использовать его полностью, можете немного модифицировать на свой вкус, а можете написать полностью свои css-стили для формы. Чтобы знать, какие css-селекторы использовать, вам достаточно заглянуть в исходный код страницы с формой и посмотреть, что сгенерировал гем `simple_form`.

В этом разделе мы рассмотрели только самые основы использования `simple_form`. Документация к гему [2] очень хорошо структурирована и там можно найти примеры создания различных типов полей для формы и опций к ним.

Обращу ваше внимание, что названия полей формы и сообщения об ошибках в нашем примере выводятся на английском языке. Чтобы вывести название полей и ошибки на русском, вам нужно посмотреть как работать с библиотекой `I18n` в документации к ActiveRecord (для сообщений об ошибках) и в документации к simple_form [3] (для названия полей).

[1] В дальнейшем, наш пример будет подразумевать, что модели `Item` есть валидация `validates :name, :description, presence: true`

[2] Скорее всего, у вас есть файл `app/assets/stylesheets/application.css`. В этом случае, вам нужно будет переименовать его в `application.css.scss`, т.к. приведенные ниже стили написаны на языке SCSS, а не на обычном CSS.

[1] *Языки SCSS и SASS (официльный сайт)*

[2] *Гем simple_form на GitHub*

[3] *Описание работы гема simple_form с I18n*

Уведомления flash и гем frontend_notifier

Иногда в веб-приложении возникает необходимость сообщить пользователю об ошибке или об успехе определенного действия таким образом, чтобы он обязательно заметил это. Например, когда пользователь пытается зайти на сайт с помощью своего логина и пароля и вводит их неправильно — нет особого смысла подсвечивать каждое или одно из этих полей — тем более, что их всего два в форме входа. Гораздо логичнее сообщить ему, что введенные им данные для аутентификации — неверные. Или же, в случае успешного входа, можно показать ему приветственное сообщение. Еще один пример: если пользователь пытался создать товар, но создаваемая модель не прошла валидацию мы, конечно, можем подсветить поля формы и пояснить в чем была ошибка в каждом поле. Но также неплохо было бы сообщить ему сначала, что он в принципе допустил ошибку при заполнении формы и что мы отметили поля с ошибками красным цветом!

Именно для таких задач в Rails-контроллерах придумали специальный метод `#flash`, который ведет себя похожим на хэш образом. Вообще говоря, я предлагю вам думать о `flash`, как о переменной в которой содержится хэш. Но есть одна особенность: все, что мы запишем в этот хэш, будет нам доступно при следующем запросе, несмотря на то, что все остальные переменные (даже инстансные) из контроллера полностью исчезают, когда пользователь делает новый запрос к сайту. Приведу простой пример. Предположим пользователь делает запрос к экшену `#sign_in`

```
def sign_in
  ...
  flash[:success] = "Welcome, dear user!"
  redirect_to root_path
end
```

Из кода видно, что мы редиректим пользователя на главную страницу сайта, то есть браузером будет отправлен еще один запрос к нашему приложению. А так как в нашем приложении главная страница обслуживается экшеном `PagesController#frontpage`, запрос уйдет именно туда:

```
class PagesController < ApplicationController
```

```
  ...

  def frontpage
    ...
    flash[:success] # => "Welcome dear user!"
  end

end
```

Получается, содержимое `flash` доступно и внутри экшена `PagesController#frontpage` после следующего запроса. Пока что, правда, нам это никак не помогло, потому что нам нужно придумать, каким образом показать это сообщение пользователю. К счастью, `flash` доступен и в шаблонах, поэтому мы можем добавить в самый верх нашего шаблона `pages/frontpage.html.erb` следующую строку кода:

```
<p class="successMessage"><%= flash[:success] %></p>
```

Разумеется, если `flash[:success]` окажется пустым — как произойдет в случае, если пользователь не залогинивался, а просто зашел на главную страницу сайта — он не увидит никакого сообщения.[1]

Теперь рассмотрим, как нам мог бы пригодиться `flash` при создании товаров в экшене `ItemsController#create`. Добавим соответствующее сообщение внутрь этого экшена[2]:

```
class ItemsController < ApplicationController

  ...

  def create
    @item = Item.create(params[:item])
    if @item.errors.empty?
      flash[:success] = "Item created!"
      redirect_to @item
    else
      render "new"
    end
  end

end
```

Т.к. редирект происходит на страницу с созданным товаром, то мы знаем, что в этом случае будет использоваться шаблон `items/show.html.erb`, поэтому прямо в него мы и добавим вывод сообщения об успешном создании товара:

```
<p class="successMessage"><%= flash[:success] %></p>
<h1><%= @item.title %></h1>
<p>Описание: <%= @item.description %></p>
<p>Цена: <b><%= @item.price %></b></p>
```

По другому дело будет обстоять, если валидации не пройдут и нам нужно будет показать

сообщение об ошибке. Как видно из кода выше, в этом случае у нас не будет происходить редиректа и контроллер просто отрендерит шаблон. Это означает, что нового запроса в контроллер не поступит. Поэтому, было бы нелогично пытаться сохранить сообщение через 1 запрос. На помощь нам придет специальный метод `flash.now`:

```
class ItemsController < ApplicationController

  ...

  def create
    @item = Item.create(params[:item])
    if @item.errors.empty?
      flash[:success] = "Item created!"
      redirect_to @item
    else
      flash.now[:error] = "You have errors in your form, please
      fix them and resubmit!"
      render "new"
    end
  end

end
```

И, естественно, не забудем добавить вывод этой ошибки в шаблон `items/new.html.erb`:

```
<p class="errorMessage">
  <%= flash[:error] %>
</p>
<h1>Создание нового товара</h1>
<%= form_for @item do |f| %>
  <%= render partial: "form", locals: { f: f } %>
  <%= f.submit "Создать товар" %>
<% end %>
```

Обратите внимание, что обращение к сообщению происходит уже обычным способом, без использованиz метода `#now`.

Таким образом, правило использования `flash` будет звучать так: **если вы делаете редирект — используйте обычный `flash`, если вы рендерите какой-то шаблон и вам нужно вывести сообщение об ошибке — используйте `flash.now`**.

Наконец, несложно догадаться, что каждый раз добавлять в разные шаблоны код для вывода сообщений `flash` — утомительно. Гораздо логичней было бы один раз добавить этот код в лэйаут, т.е. в файл `app/views/layouts/application.rb`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>StoreApp</title>
    <%= stylesheet_link_tag "application", :media => "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
  </head>
```

```
  <body>
    <p class="successMessage"><%= flash[:success] %></p>
    <p class="errorMessage">  <%= flash[:error]   %></p>
    <%= yield %>
  </body>
</html>
```

Главное — не забывать записывать все flash-сообщения в хэш под ключами `:success` и `:error`.

Использование гема frontend_notifier

Для облегчения работы с `flash` я сделал гем frontend_notifier. Он будет выводить сверху страниц сайта аккуратные сообщения на зеленом или красном фоне (в зависимости от названия ключа в `flash`), которые будут автоматически исчезать через несколько секунд. И, при желании, вы сможете изменить внешний вид этих сообщений.

Чтобы начать им пользоваться, вам нужно выполнить следующие шаги:

1. Добавьте в файл `Gemfile` строку `gem 'frontend_notifier'` и затем запустите команду `bundle install` из терминала.
2. Добавьте в самый верх файла `app/assets/stylesheets.css.scss` строку `@import "frontend_notifier";`
3. Добавьте в самый верх файла `app/assets/javascripts/application.js` строку `#= require lib/_frontend_notifier`
4. Добавьте в ваш лэйаут, после открывающего тэга `<body>` следующую строку: `<%= render partial: "shared/frontend_notifier" %>`

Не забудьте перезапустить приложение. После этого все ваши flash-сообщения на сайте будут выводиться в специальном блоке сверху. Если ключ сообщения называется `:error`, `:alert` или `:warning`, то сообщение будет выведено на красном фоне. Если ключ называется `:success` или `:notice`, то сообщение будет выведено на зеленом фоне. Естественно, вам необходимо будет использовать именно эти ключи, т.е. вы не увидите сообщения, если в вашем контроллере вы напишите например так:

```
flash[:error_message] = "You have errors in your form, please
fix them and resubmit!"
```

О ключе `:error_message` гему ничего не известно!

С гемом `frontend_notifier` сообщения также можно выводить, обратившись к объекту `FrontendNotifier`, который станет доступен во всех ваших Javascript-файлах. Это бывает

полезно, когда вы делаете ajax-запросы. Чтобы понять как работать с гемом через Javascript, посмотрите на код этого объекта.

[1] Если вы используете гем `Devise` для аутентификации пользователей, то полезно помнить, что этот гем добавляет сообщения об успехе или ошибке входа в `flash[:notice]` и `flash[:alert]` соответственно.

[2] Вы можете использовать русскоязычные сообщения во `flash`, но тогда вам следует сделать одну из двух вещей: 1) добавить в самый верх файла строчку `#encoding: utf-8` или 2) использовать библиотеку `I18n` и хранить все русскоязычные сообщения в отдельном файле.

Почему haml лучше чем erb?

В разделе Что такое шаблон и из чего он состоит? я уже упоминал, что в Rails можно использовать различные шаблонизаторы и что по умолчанию используется Erb. Однако так сложилось, что в Rails-сообществе большинство программистов начали использовать haml [1]. В этом разделе мы очень быстро рассмотрим основные преимущества и отличия, но чтобы дать вам почувствовать вкус, сравните два шаблона, каждый из которых сгенерирует совершенно одинаковый html-код — первый написан на erb, второй — на haml:

шаблон app/views/items/index.html.erb

```
<h1>Все товары</h1>
<div class="sortOrder">
  Сортировка:
  <%= link_to "Новые сверху",  items_path(order: "desc") %> |
  <%= link_to "Старые сверху", items_path(order: "asc")  %>
</div>
<ul>
  <% @items.each do |i| %>
    <li><%= link_to i.title, i %></li>
  <% end %>
</ul>
<%= will_paginate @items %>
```

шаблон app/views/items/index.html.erb

```
%h1 Все товары
.sortOrder
  = link_to "Новые сверху",  items_path(order: "desc")
  |
  = link_to "Старые сверху", items_path(order: "asc")

%ul
  - @items.each do |i|
    %li= link_to i.title, i

= will_paginate @items
```

Наверное многие согласятся, что второй пример выглядит гораздо аккуратнее. В haml нет

открывающих и закрывающих тэгов. Вместо этого, для создания пары тэгов используется символ % и имя тэга, а сам **тэг всегда автоматически закрывается**. Так, в первой строке мы создали пару тэгов `<h1></h1>`. Текст "Все товары" был автоматически обернут в эти тэги. Если бы нам понадобилось, чтобы внутри тэгов `<h1></h1>` было несколько строк текста, то нужно было бы каждую строку отделить отступом (обычно в два пробела), вот так:

```
%h1
  Все товары
  Еще одна строка
```

Кстати, именно отступами в haml определяется вложенность тэгов. Можно было бы вложить внутрь одного тэга другой, вот таким образом:

```
%h1
  Все товары
  %small Еще одна строка
```

Главное помнить, что нельзя совмещать запись содержимого тэга на одной строке с вложенностью, т.е. вот такая запись вызовет ошибку:

```
%h1 Все товары
  %small Еще одна строка
```

Поскольку тэги `<div></div>` используются в html-страницах чаще других, в haml придумали упрощенную запись — достаточно указать класс div-а через точку. Таким образом, запись

```
.sortOrder
```

эквивалентна записи

```
%div.sortOrder
```

и генерируется в html

```
<div class="sortOrder"></div>
```

Наконец, Ruby-кодом в шаблонах haml считается все, что идет после знака `=`, причем сам этот знак может ставиться как в самом начале строки, так и после какого-либо тэга. Например вот так:

```
%li= link_to i.title, i
```

Если вы хотите выполнить Ruby-код, но не выводить его в шаблон, достаточно заменить

знак `=` на знак `-` — это будет эквивалентно erb-тэгам `<% %>`. Например, ниже мы вызовем метод `#each` и передадим ему блок (то что возвращает сам метода `#each` не нужно выводить в шаблон, помните?):

```
- @items.each do |i|
```

В целом, я рекомендую использовать haml по нескольким причинам:

- Автоматически закрывающиеся тэги — ваша страница всегда будет валидной.
- Лаконичность шаблонов — их приятно читать и приятно исправлять!
- Прекрасная поддержка в Rails-коммьюнити.

Документацию по языку haml [2] можно прочитать буквально за полчаса и, потратив 1 день на привыкание, вы уже никогда не захотите возвращаться к этому страшному монстру, которым является Erb.


[1] Официальный сайт языка haml.


[2] Документация по языку haml.

Assets

Что такое Assets и Asset Pipeline?

Assets (по русски обычно произносится как "ассеты") в Rails приложении — это файлы, которые тем или иным образом были вставлены в веб-страницу, которую рендерит Rails-приложение и которые должны быть загружены браузером, чтобы страница выглядела именно так, как задумал автор. Если говорить конкретно, то ассеты это изображения, css-файлы и javascript-файлы.

Все эти файлы в Rails-приложении должны находиться в папке `app/assets/` в соответствующих поддриректориях `images/` (для изображений), `stylesheets/` (для css-файлов) и `javascripts/` (для javascript-файлов).

Asset Pipeline — это специальный механизм Rails, который умеет автоматически делать следующие вещи с ассетами: 1) сжимать их (уменьшать размер), чтобы браузеру приходилось качать меньший объем данных 2) соединять файлы вместе, чтобы браузер делал меньше запросов к серверу 3) применять к файлам т.н. препроцессинг (например преобразовывать scss в css и coffeescript в javascript) 4) создавать уникальные "отпечатки пальцев" каждого ассета, чтобы браузер мог перезагрузить ассет в случае, если он изменился

(вместо того, чтобы использовать закэшированную версию). Рассмотрим каждый из этих пунктов подробнее.

Уменьшение размера ассетов

Этот пункт относится только к css и javascript-файлам. В продакшен режиме на сервере ваша задача — загрузить страницу пользователю максимально быстро. И не только потому, что пользователь может уйти с вашего сайта не дождавшись загрузки страницы, а еще и потому, что поисковые системы при ранжировании страниц учитывают скорость их загрузки. Одним из способов сократить время загрузки страницы — уменьшить размер файлов, которые отдаются сервером, в частности css и javascript файлов. Механизм Asset Pipeline умеет делать именно это: он берет css файл, удаляет в нем все пробелы, переносы строк и комментарии и, таким образом, сокращает его размер. Примерно то же самое Asset Pipeline делает и с javascript-файлами, правда в этом случае доступны более эффективные механизмы для сжатия (например, длинные имена переменных и функций внутри файла могут заменяться на короткие).

По умолчанию, в соответствии с дефолтными настройками Rails, сжатие файлов происходит только в продакшен среде. На вашей локальной машине файлы сжаты не будут и, если вы решите заглянуть в исходный код страницы и щелкните по ссылке на один из css или javascript-файлов, вы увидите несжатую версию.

Конкатенация ассетов

Еще один способ ускорить загрузку страницы — уменьшить количество запросов, которые браузер должен сделать на сервер, чтобы загрузить страницу. Когда браузер получает от Rails-сервера ответ в виде html-страницы, он не останавливается на этом. После получения ответа, он должен прочесть весь html и найти места, где упоминаются изображения, css-файлы или javascript-файлы (и, возможно другие, более редкие типы ассетов, такие как шрифты), и для каждого такого упоминания сделать еще один запрос на сервер, чтобы получить собственно файл с упомянутым именем. Таким образом, чтобы сократить количество запросов, которые браузер должен сделать на сервер, мы могли бы объединить несколько ассетов в один и вернуть их браузеру в виде одного файла.

Предположим у нас есть два файла: `application.css` и `items.css` — оба содержат какие-то css-стили и оба нужны для правильного отображения страниц нашего сайта. Мы могли бы внутри лэйаута `app/views/layouts/application.erb` добавить оба эти файла в страницу:

```
...
<%= stylesheet_link_tag "application", "items" %>
```

```
...
```

и в этом случае браузер сделает два запроса к серверу — один для файла `application.css` и второй для файла `items.css`. Чтобы превратить это в один запрос к серверу, нам необходимо добавить в файл `application.css` следующую строку:

```
/*
*= require items
*/
...
```

и затем нам останется удалить упоминание о файле `items.css` из тэга в лэйауте:

```
<%= stylesheet_link_tag "application" %>
```

Таким образом, файл `items.css` окажется вложенным в `application.css` и браузер скачает с сервера только последний файл, выполнив всего-лишь один запрос, вместо двух.

Похожим образом можно вложить один javascript-файл в другой. Вот как будет выглядеть код в файле `application.js`, если мы захотим вложить в него файл `items.js`:

```
//= require items
...
```

Как видите, отличие состоит только в символах комментария, которые предваряют директиву `require`: в css комментарии ограничиваются последовательностью `/* */`, а в javascript-е комментариями считаются строки, начинающиеся на `//`.

Asset Pipeline определяет еще две интересных директивы — `require_tree` и `require_self`, почитать подробнее о которых можно в Rails Guides [1].

Препроцессинг

С развитием Rails стали популярны "надстройки" над css и javascript, которые позволяли в более элегантной форме писать код css и javascript-файлов, а затем, с помощью соответствующих библиотек, эти файлы компилировались в настоящие css и javascript-файлы, понятные браузерам. Например язык scss [2] позволил использовать при описании стилей страницы вложенные селекторы, которые, естественно, не поддерживаются в обычном css. А язык Coffeescript [3] позволил избавиться от ненавистных многими фигурных скобок, которые являются обязательными в языке Javascript.

Описание этих языков выходит за рамки книги, но полезно знать, что в Rails можно без каких-либо дополнительных усилий использовать именно эти языки, а не стандартные css и javascript. Для этого вам достаточно назвать ваш файл таким образом, чтобы у него было двойное расширение в имени файла, например `application.css.scss` или

`application.js.coffee`. Увидев такой файл, Asset Pipeline, перед тем как позволить браузеру его скачать, скомпилирует его в обычный `.css` или `.js` соответственно. Именно это и называется препроцессингом.

"Отпечатки пальцев" ассетов

Когда браузер впервые загружает веб-страницу, он скачивает все ассеты, которые были в ней упомянуты. Однако, иначе дело обстоит при повторной загрузке страницы. Браузер помнит имена всех ассетов, которые были им когда-либо загружены и, более того, хранит копии этих ассетов на машине пользователя. Когда браузер видит, что страница просит его загрузить ассет, который он уже загружал ранее — он просто использует локальную копию этого файла и не делает дополнительного запроса к серверу. Причем это происходит независимо от того, просматривает пользователь ту же самую страницу или зашел на другую страницу, использующую те же самые ассеты.

На первый взгляд это может показаться идеальным поведением — ассеты грузятся один раз, а все остальные обращения к страницам вашего сайта будут выдавать пользователю результат практически молниеносно, т.к. браузеру не нужно будет по новой загружать изображения, css и javascript-файлы. Однако, если вы внесете изменения в какой-то ассет — поставите новый логотип на сайт или исправите какие-либо css-стили — то браузер не загрузит новую версию. Он продолжит использовать локальную копию старой версии этого ассета, и ваши пользователи вместо нового логотипа будут продолжать видеть старый. Чтобы избежать такого поведения, в Rails был придуман специальный механизм, называемый *asset fingerprint* и суть его заключается в следующем: к имени каждого файла с ассетом присоединяется некая длинная строка, состоящая из чисел и букв. Каждый раз, когда ваш ассет изменяется и вы деплоите ваше приложение с новой версией ассета, Rails понимает, что содержимое файла изменилось, и изменяет добавляемую к имени файла строку. Таким образом, измененный ассет всегда будет иметь другое имя и браузер будет вынужден его загрузить по новой. В качестве примера можно привести имя файла `application.css`, который в продакшен-среде будет называться `application-908e25f4bf641868d8683022a5b62f54.css`[1]. Если вы измените содержимое этого файла, задеплоите приложение и посмотрите в исходный код страницы, вы увидите, что файл изменил свое имя на `application-a94cf78aca801e724bb1700a92c1a217.css`

[1] Важно понимать, что эта длинная часть имени файла состоящая из чисел букв, будет отличаться в вашем приложении и для каждого файла. Здесь это сочетание чисел и букв приведено в качестве примера.

[1] Подробное описание Asset Pipeline, Rails Guides.

[2] Официальный сайт, посвященный языку sass (scss).

[3] Официальный сайт и документация языка Coffeescript.

Как правильно вставить изображение в страницу

Чтобы понять, как правильно вставлять изображение в страницу, нам для начала нужно обсудить какие у программиста есть варианты, если он хочет это сделать. Вариантов, с точки зрения html, на самом деле всего два: использовать тэг `<img>` или свойство `background-color` в css. Чуть ниже мы рассмотрим каждый из этих вариантов и разберемся, как с ними быть в Rails приложении. Но, в любом случае, нам нужно понять задачу, которая перед нами стоит: т.к. в Rails ассеты имеют "отпечатки пальцев"[1], нам нужно знать правильное имя файла с изображением (включая часть с отпечатком), чтобы загрузить этот файл в страницу. Правильное имя файла в любой момент времени знает только наше Rails-приложение, а значит мы не можем самостоятельно указать имя файла в тэге или в css-свойстве и нам понадобятся для этого специальные хэлперы.

Если вы хотите вставить изображение с помощью тэга `<img>`, вам следует использовать специальный хэлпер `#image_tag`. Он сгенерирует вам тэг `<img>` с правильным отпечатком в имени файла. Например код:

```
<%= image_tag "logo.png" %>
```

в браузере пользователя превратится в

```
<img src="logo-908e25f4bf641868d8683022a5b62f54.png"/>
```

Вот и вся наука. Просто всегда используйте хэлпер `#image_tag` и все будет в порядке.

Не сложнее дело обстоит и с загрузкой картинок в css. Предположим, вам надо установить фон для вашей страницы используя файл `app/assets/images/background.png`. Тогда в вашем css файле вы могли бы использовать для этого erb-тэги (Asset Pipeline позволяет это делать) и специальный хэлпер `#asset_path`:

файл app/assets/stylesheets/application.css

```
body { background-image: url(<%= asset_path 'background.png'
%>); }
...
```

После процессинга файла, эта строка превратится в:

```
body { background-image: url("/assets/background-a94cf7...png"); }
```

Если вы используете scss вместо css, то вам даже не понадобятся и erb-тэги. Вместо них в scss есть готовый scss-хэлпер `image-url`:

файл app/assets/stylesheets/application.css.scss

```
body { background-image: image-url('background.png'); }
...
```

Этот scss-код будет преобразован в тот же самый готовый css, который был продемонстрирован чуть выше.

Как видите, вставлять изображения в Rails-приложения — крайне просто. Главное не лениться и разобраться в том, почему именно нам нужно использовать все эти специальные хэлперы и как именно работают ассеты. Надеюсь этот и предыдущий раздел помогли вам в этом.

[1] Отпечатки пальцев только что обсуждались нами в разделе Что такое Assets и Asset Pipeline? — убедитесь, что вы понимаете, что это такое.

CSS, SCSS и как правильно структурировать стили

В этом разделе я буду подразумевать, что вы все же решились использовать scss вместо обычного css и покажу несколько полезных приемов, которые помогут вам грамотно организовать стили в вашем приложении.

Импорт файлов

В разделе Что такое Assets и Asset Pipeline? я уже показывал, как производить конкатенацию нескольких файлов в один — для этого мы использовали директиву `*=require`. Если вы перешли на scss, вы можете заменить эту директиву на специальное выражение `@import`. Вот как мы могли бы вложить файл `items.css.scss` в `application.css.scss`:

файл app/assets/stylesheets/application.css.scss

```
@import "items";
```

(Не забудьте поставить точку с запятой в самом конце выражения.)

Я часто импортирую таким образом вспомогательные файлы, содержащие переменные с различными цветами и mixins (о том что это такое — чуть ниже). Например:

```
@import "colors";
@import "mixins";
```

Сами файлы при этом называются `_colors.css.scss` и `_mixins.css.scss`, т.е. в начале имени файла стоит символ нижнего подчеркивания. Это нужно затем, чтобы Rails не тратил время и компилировал их до того, как эти файлы будут вложены внутрь `application.css.scss`. Иначе получится ситуация, когда Rails проделает двойную работу: скомпилирует файлы до вложения, а затем еще один раз — после. Чтобы этого избежать, мы поставим в самое начало имен файлов, которые мы не хотим компилировать отдельно, символ нижнего подчеркивания, а затем добавим в конфигурацию Rails (файл `config/application.rb`) следующую строку:

```
...
class Application  Rails::Application
  ...
  config.assets.precompile = [/^[^_]/]
end
...
```

Эта строка содержит *регулярное выражение*, которое объясняет Rails, что можно компилировать любые ассеты, кроме тех, чье имя начинается на символ нижнего подчеркивания.

Переменные и mixins

В языке css нет переменных. На практике это означает, например, что если мы хотим использовать один и тот же цвет в нескольких местах (например, чтобы у двух блоков на сайте был одинаковый фон), мы должны будем скопировать значение этого цвета из свойств одного селектора в свойства другого:

```
.leftBlock  { background-color: &1a1a1a; }
.rightBlock { background-color: &1a1a1a; }
```

Мало того, что это не очень удобно — искать где указан цвет одного блока, копировать его, затем вставлять значение в нужное место — это создаст дополнительные проблемы, если в какой-то момент мы решим поменять цвет этих блоков на другой. Нам придется проделать двойную работу. А если таких блоков не 2, а 10? К счастью, в scss существует понятие переменных, которые мы можем с легкостью начать использовать. например так:

```
$standard_block_color: &1a1a1a;
.leftBlock  { background-color: $standard_block_color; }
```

```
.rightBlock { background-color: $standard_block_color; }
```

В своих стилях я предпочитаю выносить все цвета в отдельный файл `_colors.css.scss` и затем импортировать его:

файл app/assets/stylesheets/_colors.css.scss

```
$colors_text          : &000;

$colors_link          : &006abc;
$colors_link_visited  : &bc0089;
$colors_link_active   : &af0c00;

$colors_green         : &288e00;
$colors_yellow        : &FFF5A1;
$colors_red           : &960000;
$colors_gray          : &aaa;
$colors_light_gray    : &a7a7a7;
$colors_pink          : &ffb8b8;
$colors_salad         : &d4ffcc;
```

файл app/assets/stylesheets/application.css.scss

```
@import "colors";
...
```

Еще одним удобством css являются *mixins*. Они чем-то напоминают переменные, но вместо одиночных значений в mixins можно записать целый набор свойств. Например, я мог бы создать mixin, который делает любой блок, в который он добавлен, зеленого цвета с розовым шрифтом размера 12px:

```
@import "colors";
@mixin acid_block { background-color: $colors_green; color: $colors_yellow;
font-size: 12px; }

/* вкладываем этот mixin в селектор блоков с помощью директивы
@include */
.leftBlock { width: 200px; @include acid_block; }
.footer    { width: 100%; height: 70px;  @include acid_block; }
```

Почему я пишу все свойства для селектора на одной строке

Полагаю, что вы уже успели заметить — я предпочитаю писать свойства для каждого селектора на одной строке. Некоторые предпочитают записывать каждое свойство на новой строке, например так:

```
a {
  color: blue;
```

```
  font-size: 1em;
  text-decoration: underline;
}
```

Мне такой подход не нравится по нескольким причинам. Во-первых, когда ваш файл со стилями начнет расти, вам станет гораздо сложнее ориентироваться в нем. Представьте себе 100 селекторов, стили каждого из которых написаны с новой строки! Визуально будет очень сложно найти нужный вам селектор, и вам придется почти каждый раз пользоваться поиском по файлу. Во-вторых, такой стиль записи делает гораздо менее читабельными вложенные селекторы. Сравните:

```
.leftBlock {
  background-color: $colors_green;
  font-size: 12px;
  a {
    color: blue;
    font-size: 1em;
    text-decoration: underline;
  }
}
```

и

```
.leftBlock {
  background-color: blue; font-size: 12px;
  a { color: blue; font-size: 1em; text-decoration: underline; }
}
```

Во втором варианте мы указали свойства на одной строке, вложенный селектор — на другой. Я считаю это гораздо более лаконичным и правильным вариантом.

Но, как бы там ни было, если вы работаете над проектом с другими программистами и верстальщиками, самое важное — чтобы каждый участник команды писал css в одинаковой манере. Если один из вас будет переносить каждое свойство на новую строку, а второй будет писать свойства на одной строке, в вашем проекте начнется бардак.

Как загружать только необходимые странице стили

Одна из инетересных тем, которая не получила большого распространения в Rails сообществе — как загружать в том или ином шаблоне или группе шаблонов только нужные ему стили. Очевидным способом было бы использовать хэлпер `#stylesheet_link_tag` в самом верху шаблона и перечислить в качестве аргументов к этому хэлперу нужные css-файлы. Однако тут мы сталкиваемся с небольшой проблемой: css-файлы желательно подключать в самом верху html-страницы, между тэгами `<head></head>`, а это можно сделать только в лэйауте. Задавшись этой задачей, я написал гем [specific_assets](), который

позволяет с легкостью делать описанное.

Предположим, мне нужно, чтобы браузер подгружал стили из файла `orders.css` только на страницах, имеющих отношение к заказам. После установки гема `specific_assets`, у меня есть несколько вариантов. Я мог бы добавить следующую строку в каждый шаблон в папке `app/views/orders`:

```
<% add_css "orders" %>
```

или я мог бы вызвать тот же самый метод `#add_css` в экшенах контроллера `OrdersController`:

```
class OrdersController  ApplicationController

  def new
    ...
    add_css "orders"
  end

  def show
    ...
    add_css "orders"
  end

  ...

end
```

Я также мог бы один раз вызвать в контроллере метод класса `#add_css`:

```
class OrdersController  ApplicationController

  add_css "items"

  def new
    ...
  end

  def show
    ...
  end

  ...

end
```

Любой из этих вариантов отрендерит страницу с тэгом, который подключит стиль из файла `app/assets/stylesheets/orders.css.scss`.

Сам метод `add_css` стал доступен в шаблонах именно благодаря гему `specific_assets`.

Причина, по которой иногда бывает важно подгружать css файлы только на отдельных страницах, снова состоит в скорости загрузки страницы. Чем меньше ассетов грузит

браузер, тем лучше! Даже если предположить, что ассет `orders.css.scss` будет вложен в `application.css.scss` и браузер сделает один запрос к серверу вместо двух, мы все равно попадем в ситуацию, когда браузеру придется скачать по новой файл `application.css.scss` в случае изменений в файле `items.css.scss` — произойдет это именно потому, что мы вложили один ассет в другой. Если этого не делать, а воспользоваться гемом `specific_assets`, мы сможем избежать ситуаций, когда браузер будет делать бесполезные загрузки обновленных ассетов, изменения в которых не влияют на внешний вид и поведение текущей страницы.

Javascript и Coffeescript

Начиная с версии 3.1 в Ruby On Rails по умолчанию начал использоваться Coffeescript. Это всего лишь означает, что когда вы генерируете новый контроллер, к нему в директории `app/assets/javascripts/` будет автоматически сгенерирован одноименный файл с расширением `.js.coffee`. Если вам не нравится Coffeescript, вам достаточно будет переименовать файл и убрать из расширения слово `.coffee` — тогда этот файл не будет обрабатываться препроцессором Coffeescript.

С точки зрения assets, использование Javascript или Coffeescript файлов несильно отличается. Вы, почти точно также как и с файлами стилей, можете вкладывать одни файлы в другие с помощью директивы `//=require`. Если вы используете гем `specific_assets`, который мы обсуждали в разделе CSS, SCSS и как правильно структурировать стили вам становится доступным метод `add_js`, который можно вызывать как в шаблонах, так и в контроллерах.

Наконец, если вы хотите сделать тяжелый фронтэнд и научить его работать с RESTful контроллерами в Rails, вам, скорее всего, нужно будет придерживаться следующего алгоритма:

1. Найти Javascript-фреймворк, который удолетворяет вашим требованиям.
2. Проверить отзывы других людей о том, насколько удобно и легко использовать его вместе с Rails.
3. Найти документацию или статьи по использованию этого фреймворка с Rails.

Самые известные Javascript-фреймворки для фронтэнд-разработки это Backbone.js [1] и Ember.js [4] и в списке литературы к этому разделу я привел несколько ссылок на статьи и Ruby-гемы, которые позволят вам успешно подключить их к Rails.

 *[1] Официальный сайт Backbone.js*

 *[2] Книга "Learn Backbone.js on Rails" (на английском)*

 *[3] Ruby-гем backbone-rails*

 *[4] Официальный сайт Ember.js*

 *[5] Beginning Ember.js on Rails*

 *[6] Ruby-гем ember-rails*

Глава 8. Тестирование (моделей и контроллеров) и зачем это нужно

Что такое тестирование?

Предположим в своем классе `Item` вы написали метод, который считает цену товара, включая налог:

```
class Item

  ...

  def price_with_tax
    price + price*0.3
  end

end
```

Этот код кажется относительно простым и прямолинейным. Но представим себе на секунду воображаемое преступное государство, которое садистски издевается над бизнесменами и тратит огромное количество их времени на то, чтобы они вычисляли ставку налога по крайне запутанной схеме. Вы, как умный программист и законопослушный бизнесмен, не желающий попасть в кровавые лапы государства, решаете автоматизировать этот процесс и помещаете всю логику вычисления налога в специальный метод, в том же классе. Назовем этот метод `#tax`:

```
class Item

  ...

  def price_with_tax
    price + price*tax
  end

  private

    def tax
      # сложная логика вычисления налоговой ставки
      # этот метод возвращает число от 0 до 1,
      # которое, собственно, и является налоговой ставкой
      # например 0.3
    end

end
```

Как видите, вместо множителя `0.3` мы подставили вызов метода `#tax`.

Теперь, конечно, вы можете надеяться, что ваш метод все считает правильно. Но если вы допустили ошибку при написании кода метода `#tax` и рассчеты получатся неверными — об этом вы узнаете только когда к вам в дверь постучат люди в форме и потребуют заплатить штраф. Если вы подоходите к вопросу ответственно (или правда очень боитесь налоговой инспекции) вы, разумеется, для начала, протестируете все вручную на вашей локальной машине: зайдете в приложение через браузер, купите парочку ненастоящих товаров, убедитесь, что налог рассчитан правильно и только потом задеплоите приложение на продакшен сервер.

Проблема с таким подходом заключается в том, что если вы продолжите работать над приложением в будущем, код в вашем классе будет меняться. И это необязательно должен быть код в методе `#tax`. Может так случиться, что при рассчете ставки налога, метод `#tax` использует другие методы и атрибуты модели `Item`, изменение поведения которых повлияет на результат, возвращаемый методом `#tax`. Конечно, вы можете снова протестировать все через браузер перед очередным релизом. Но если представить, что это далеко не единственная вещь, которую вам придется проверять — а только одна из многих — то ручное "прощелкивание" сайта через браузер перед каждым релизом быстро превратится сначала в утомительную, а затем и в неподъемную задачу с огромным потенциалом пропустить какие-то важные баги.

Хорошо было бы как-то автоматизировать процесс тестирования метода `#price_with_tax`. Для этого программисты придумали писать проверяющий код (тоже на языке Ruby!). Проверяющий код можно запустить в любой момент, причем, помимо проверки метода `#price_with_tax`, в нем могут находиться и проверки для других методов в вашем приложении. Все что вам потребуется сделать перед очередным релизом — запустить одной командой из терминала все тесты для вашего приложения и дождаться их выполнения. Это будет фактически эквивалентно ручному "прощелкиванию" всего вашего сайта. Если тесты не покажут ошибок, то значит все в порядке и налоговая служба не будет иметь к вам претензий.

Таким образом тесты — это специальный код, написанный на том же самом языке, единственная задача которого заключается в том, чтобы проверить, как выполняется основной код приложения. Пользователи вашего приложения ничего не знают о существовании этого тестового кода и никак с ним не взаимодействуют. Тесты нужны только программисту, чтобы сэкономить его время и усилия в будущем и помочь отловить различные баги.

Полагаю, общие фразы не сильно способствуют пониманию, так что прямо в следующем разделе мы посмотрим, как выглядит тестовый код и что именно там нужно писать.

Как написать тест? Rspec и другие библиотеки для тестирования

Для написания тестов в языке Ruby существуют различные библиотеки, самые популярные из которых — Test::Unit [1] (встроена в язык Ruby) и Rspec [2]. Их синтаксис отличается, но базовый подход к тестированию во всех библиотеках остается один и тот же:

1. Внутри теста вы вызываете тот метод, который вы тестируете.
2. Затем вы проверяете, что возвращает этот метод и соответствует ли возвращаемое значение тому, что вы ожидаете.

Поскольку мы обычно тестируем только один отдельно взятый метод, такой тип тестирования называют unit-тестированием[1]. В этом разделе мы коснемся базовых понятий и попробуем написать наш первый тест на Rspec. Я настоятельно рекомендую вам не ограничиваться этой главой и познакомиться с библиотекой Rspec подробнее, прочитав ссылки, приведенные в списке литературы к этому разделу. В частности, есть прекрасная книга The Rspec Book [3], в деталях раскрывающая секреты использования Rspec.

Рассмотрим короткий пример теста для метода `Item#price_with_tax`, написанный на Rspec:

```
it "calculates the correct price for the item that includes a tax" do
  item = Item.new(price: 10)
  item.price_with_tax.should == 13
end
```

В первой строке мы использовали специальный метод `#it`, который символизирует собой начало теста. Этот метод доступен благодаря тому, что у вас в Rails приложении установлен гем Rspec[2]. Метод носит имя `#it`, потому что он символизирует собой тестируемый объект (в нашем случае — объект класса `Item`). В качестве аргумента этот метод принимает описание теста, которое может быть произвольной фразой на английском языке — эту фразу пишут таким образом, чтобы она сочеталась со словом *it* по правилам английской грамматики. Таким образом, первую строку можно было бы прочитать следующим образом: "Класс Item рассчитывает правильную цену товара с учетом налога".

Далее, тело теста находится в блоке, который принимает метод `#it`. Во второй строке мы создаем новый объект класса `Item` и присваиваем ему цену. Создавать новый объект нам требуется потому, что метод `#price_with_tax`, который мы будем тестировать, является инстансным методом и может быть вызван только на объектах класса `Item`, а не на самом классе.

После того, как объект создан и записан в локальную переменную `item`, мы, в третьей

строке, вызываем на нем метод `#price_with_tax`. Затем, мы имеем возможность вызвать на том значении, которое вернется из этого метода, специальный метод `#should` — его Rspec автоматически добавил ко всем объектам. То есть, поскольку мы находимся внутри теста Rspec, мы могли бы написать `1.should` или `"hello".should` — метод `#should` можно "прикрепить" к чему угодно. И этот метод — самая важная часть нашего теста, потому что именно он определяет "пройдет" тест или выдаст ошибку. Как несложно догадаться, в данном выражении в третьей строке нашего теста метод `#should` проверяет, что значение возвращаемое методом `#price_with_tax` должно быть равно 13.

Но откуда взялось число 13? Все просто: перед тем как писать тест, вы вручную (то есть, например, на листке бумаги) рассчитали налоговую ставку для товара стоимостью 10 рублей и вывели, что ставка должна быть равна 30%. Соответственно, теперь в нашем тесте мы проверяем, что цена этого товара с учетом налога должна быть равна 13. Если формула в вашем методе `Item#tax` верна, то тест "пройдет" и ошибки не возникнет.

Вас наверняка мучает вопрос: в каком же файле должен находиться этот код и как его запускать? Если вы успешно установили Rspec в ваше Rails-приложение, у вас в корневой директории должна была появиться папка `spec/`. Зайдите в нее и создайте еще одну папку `models/` в которой создайте файл `item_spec.rb` (название папок и файлов должно быть именно таким). Внутрь файла `item_spec.rb` поместите следующие строки:

```
require 'spec_helper'

describe Item
end
```

Сейчас нам пришлось сделать все это вручную, потому что мы сгенерировали модель `Item` еще до того, как установили Rspec в Rails-приложение. В будущем, каждый раз, когда вы будете создавать новую модель с помощью команды `rails g model`, в папке `spec/models` будет появляться соответствующий файл для тестов (его еще называют "спекой", от spec), в котором уже будут содержаться приведенные выше строки.

Все, что нам нужно сделать теперь — поместить приведенный код теста внутрь блока передаваемого методу `#describe`, вот так:

файл spec/models/item_spec.rb

```
require 'spec_helper'

describe Item

  it "calculates the correct price for the item that includes
  a tax" do
    item = Item.new(price: 10)
    item.price_with_tax.should == 13
  end
```

```
end
```

Теперь мы можем запустить наш тест в терминале с помощью команды `rspec spec` (вам нужно находиться в корневой директории проекта). В вашем терминале должно появиться одно из двух: либо сообщение *1 example, 0 failures*, либо сообщение об ошибке, которое вам следует внимательно прочесть!

Сообщения об ошибке при запуске тестов Rspec могут быть двух типов. Во-первых это можеть быть ошибка в вашей программе (например вы вызвали какой-то метод на объекте `nil`), и в этом случае вам нужно заняться исправлением вашей программы — вполне возможно, что ошибка произошла вовсе не в том методе, который вы тестировали. Во-вторых, это может быть ошибка Rspec, говорящая о том, что ожидаемое `#should` значение не соответствует тому, что вернул вам тестируемый метод. В нашем случае это означало бы, что вы либо неправильно рассчитали налоговую ставку вручную, либо в вашей формуле в методе `Item#tax` была допущена ошибка.

В дальнейшем мы будем добавлять новые тесты внутрь блока `#describe` "покрывая" тестами новый функционал нашей модели. И каждый новый тест будет начинаться со слова `#it`.

[1] Существует еще интеграционное тестирование, но в этой книге мы его касаться не будем.

[2] Чтобы установить гем Rspec, вам необходимо добавить строку `gem "rspec-rails"` в файл `Gemfile` и запустить команду `bundle install`. После этого, установка Rspec еще не закончена — следуйте инструкциям из документации.

[1] *Документация по библиотеке Test::Unit*

[2] *Документация по библиотеке Rspec*

[3] *The Rspec Book*

[4] *Гид по тестированию Rails-приложений, Rails Guides*

Тестовая среда и база данных

Когда вы запускаете команду `rspec spec` загружаются не только файлы с вашими тестами из директории `spec/` — предварительно загружается все Rails приложение целиком. Причем оно загружается в специальной среде, которая называется `test`[1]. Это, в свою очередь, означает, что для тестов используется отдельная база данных.

Rspec в Rails работает таким образом, что каждый раз перед тем, как запускаются тесты, тестовая база данных очищается. Это нужно затем, чтобы не держать в ней много бесполезных данных из предыдущих запусков тестов (а "прогонять" тесты следует как

минимум перед каждым коммитом, который вы делаете в git-репозиторий с вашим приложением). Откуда в тестовой базе берутся данные, спросите вы? Дело в том, что в ваших тестах вам наверняка понадобится сохранять вновь созданные модели. Предположим, вам потребуется тест, который будет проверять, что после добавления нового товара, счетчик товаров категории обновился:

файл spec/models/item_spec.rb

```
require "spec_helper"

describe Item do

  it "updates category items_counter when a new item is added
  into this category" do

    # Сохраняем категорию в БД
    category = Category.create(...)

    # Сохраняем товар в БД
    item     = Item.create(category_id: category.id, ...)

    # Проверяем счетчик, до добавления товара он был равен 0
    category.reload.items_count.should == 1

  end

end
```

Здесь, в первой и второй строках теста мы, с помощью метода `.create`, создали записи в БД для категории и товара. Нам было никак не обойтись без метода `.create`, т.к. обновление счетчика в нашей модели `Item` было завязано на коллбэк `after_create`:

```
class Item  ActiveRecord::Base

  ...

  after_create
    category.increment!(:items_count)
  end

end
```

Так как тесты используют базу данных из среды test, то нам важно не забыть применить к ней все те же самые миграции, которые вы прогнали для development базы данных. Иначе тестируемые вами модели столкнутся с тем, что для них нет соответствующих таблиц и полей в тестовой базе данных. Для того чтобы применить миграции к вашей тестовой базе данных, есть специальный rake-таск:

```
rake db:test:clone
```

который просто применит текущую схему в файле `db/schema.rb` к вашей тестовой базе данных. Я взял за привычку выполнять эту команду одновременно с `rake db:migrate` каждый раз, когда я добавляю в проект новые миграции. Делается это так:

```
rake db:migrate && rake db:test:clone
```

[1] Мы обсуждали что такое среды (environments) в разделе Среды (Environments) главы Создание Rails-приложения

Структурирование тестов

Чем больше unit-тестов будет появляться у вас в отдельном файле, тем сложнее будет в нем ориентироваться. Кроме того, с ростом количества тестов вы будете замечать, что какие-то куски кода начинают повторяться, нарушая тем самым принцип DRY. В этом разделе я покажу несколько приемов, которые позволят вам избежать подобных проблем.

Вложенные describe

В коде, который мы рассматривали в предыдущих разделах вы, полагаю, обратили внимание на метод `#describe`, который является частью DSL Rspec-а. Смысл его использования заключается в том, что он объединяет группу тестов под одним названием. Например, если мы пишем тесты для модели `Item`, то в spec-файле обязательно должен быть метод `#describe`, принимающий в качестве аргумента сам класс `Item`. Однако, иногда бывает полезно попытаться объединить тесты еще по какому-то признаку. Например, мы могли бы создать вложенный `#describe`, который бы содержал все тесты, проверяющие правильное поведение счетчиков товаров в категориях:

файл spec/models/item_spec.rb

```
require "spec_helper"

describe Item do

  describe "handling items_counter in category"

    it "updates category items_counter when a new item is added
    into this category" do
      category = Category.create(...)
      item     = Item.create(category_id: category.id, ...)
      category.reload.items_count.should == 1
    end

    it "updates category items_counter when a new item is destroyed"
    do
```

```
      category = Category.create(...)
      item     = Item.create(category_id: category.id, ...)
      item.destroy
      category.reload.items_count.should == 0
    end

  end

end
```

В первом тесте мы проверили, что счетчик товаров в категории увеличивается на единицу, когда мы создаем товар. Во-втором тесте мы проверили, что счетчик категории уменьшается на единицу, если мы удаляем товар.

Обратите особое внимание, что мы придумали такое описание для вложенного `#describe`, что оно выстраивается в грамматически верную английскую фразу вместе с описаниями конкретных тестов: *Item handling items_counter in category updates category items_counter when a new item is added into this category*. Немного сумбурно, но в целом — верно!

У группировки тестов с помощью вложенных `#describe` есть не только косметическое преимущество — помимо того, что сгруппированные тесты выглядят понятнее для программиста, они еще помогают нам грамотно использовать методы `#before` и `#after` и избежать дублирования кода. Речь об этом пойдет ниже.

Выполнение кода перед и после тестов, методы #before и #after

Если внимательно посмотреть на код, приведенный выше — можно заметить, что первые две строки из каждого теста явно повторяются. Rspec дает нам возможность избежать подобного рода повторений и выполнять один и тот же код **каждый раз перед запуском очередного теста**. Вот как мы могли бы переписать наш код с использованием специального метода `#before`:

файл spec/models/item_spec.rb

```
require "spec_helper"

describe Item do

  describe "handling items_counter in category"

    before(:each) do
      @category = Category.create(...)
      @item     = Item.create(category_id: category.id, ...)
    end

    it "updates category items_counter when a new item is added
```

```
    into this category" do
      @category.reload.items_count.should == 1
    end

    it "updates category items_counter when a new item is destroyed"
    do
      @item.destroy
      @category.reload.items_count.should == 0
    end

  end

end
```

Метод `#before` наравне с аналогичным ему методом `#after` в документации Rspec называют хуками (hooks).[1] Код, переданный в блоке методу `#before` будет выполнен по новой *перед каждым следующим тестом*. Выполнение перед каждым *следующим* тестом произойдет именно потому, что мы передали методу `#before` в качестве аргумента символ `:each`. Мы могли бы передать ему в качестве аргумента `:all`, но тогда этот код выполнился бы один раз перед всеми тестами, а это нам, в данном случае, сосвем не подходит (убедитесь, что вы понимаете почему!).

Вы также могли заметить, что мы сделали переменные `item` и `category` — инстансными (поставили символ @ перед ними). Если бы мы этого не сделали, то переменные внутри блока, переданного `#before(:each)`, были бы недоступны внутри каждого из тестов.

Также обратите внимание на то, что `#before(:each)` находится внутри вложенного `#describe` и не имеет отношения к тестам, которые могут находиться за его пределами. В реальном приложении, у вас могут быть разные блоки `#before(:each)`, один из которых может иметь отношение к `#describe` находящемуся на самом верхнем уровне, а второй может относиться к вложенному `#describe`.

В целом, хуки `#before` и `#after` в сочетании с вложенными `#describe` — это прекрасный способ структурировать тестовый код и избежать его дублирования. Однако это способ не единственный. Если вам интересно дальше исследовать этот вопрос, я рекомендую вам изучить как создавать свои собственные матчеры и макросы [3].

[1] Подробнее о хуках в Rspec вы можете прочитать в документации [2].

[1] *Документация по методам #describe и #it.*

[2] *Документация по методам-хукам #before и #after.*

[3] *Транскрипция скринкаста Railscats по созданию собственных матчеров и макросов в Rspec.*

Rspec Матчеры

Мачтеры — это специальные методы Rspec, которые облегчают нам проверку возвращаемых значений. Эти методы передаются в качестве аргумента методам `#should` и `#should_not`. В предыдущих разделах мы использовали `#should` например следующим образом:

```
category.reload.items_count.should == 1
```

Вместо этого мы могли бы использовать специальный матчер `#eq()` (название метода в данном случае — это сокращение от глагола "equals", то есть "равняется"):

```
category.reload.items_count.should eq(1)
```

Целиком такое выражение, на одном конце которого есть метод `#should` а на другом — какой-нибудь матчер (передающийся в виде аргумента методу `#should`[1]), называется *expectation* (ожидание). Если expectation выполняется, то тест "проходит"; если expectation не выполняется, то говорят, что тест "фэйлится" и, в этом случае, Rspec выведет в терминал причину, почему оно не выполнилось. Если приводить общую формулу, по которой конструируются expectations, то она будет выглядеть так:

```
actual.should matcher(expected)
actual.should_not matcher(expected)
```

Как можно догадаться метод `#should_not` инвертирует expectation и выводит ошибку только если значение слева *совпадает* с матчером справа.

Посмотреть весь список доступных матчеров, а также объяснения того, как они работают и в каких случаях они используются, можно в документации к RSpec [1] и RSpec Rails [2]. Я же приведу несколько примеров с различными матчерами, которые помогут вам понять общий принцип:

```
1.should                    eq(2)              # Тест зафэйлится
(1  2).should               be_true            # Тест пройдет
["a", "b", "c"].should_not  include("d")       # Тест пройдет
["a", "b", "c"].should      have(3).items      # Тест пройдет
5.should_be                 be_kind_of(String) # Тест зафэйлится
"hello".should_not          be_nil             # Тест пройдет
```

В Rspec также есть возможность создавать свои собственные матчеры. Обсуждение этого вопроса выходит за рамки книги, но вы можете обратиться к соответствующей странице документации RSpec. [3]

[1] Напоминаю, что в языке Ruby передавать значения для аргументов в метод можно без круглых скобок — в таких случаях между названием метода и аргументами просто ставится пробел.

[1] *Стандартные матчеры Rspec*

[2] *Стандартные матчеры Rspec Rails*

[3] *Создание собственных матчеров*

Тестирование моделей

Написание тестов для моделей в Rails ничем не отличается от написания тестов для любого класса Ruby. Самое важное в данном случае — понять что именно нужно тестировать, а какие тесты писать бессмысленно. В этом разделе мы рассмотрим несколько случаев и, затем, я приведу пример типичного теста для модели.

Основное правило, которому нужно следовать при написании любых тестов, в том числе тестов для моделей — тестировать только те методы, которые написали лично вы. Например, у всех моделей ActiveRecord есть методы `.create` и `#save`, сохраняющие записи в БД. Теоретически, мы могли бы создать тест для нашей модели `Item`, в котором мы создаем и сохраняем новый товар, а затем пробуем найти этот же товар в БД, например так:

```
describe Item do

  ...

  it "saves item record in DB" do
    item = Item.create(...)
    Item.find(item.id).should_not be_nil
  end

end
```

Однако смысла писать такой тест довольно мало. Программисты, которые придумали метод `#save`, уже протестировали его работоспособность в тестах, написанных для Rails. Мы можем быть уверены, что метод действительно сохранит запись в БД.

Все вышесказанное имеет отношение не только к методам `#create` и `#save`, а к любому функционалу, доступному вам по умолчанию в моделях ActiveRecord: валидациям, ассоциациям и методам, которые выполняют запросы на поиск (`.find`, `.where`, `.order` и так далее). Все это работает надежно и уже протестировано за вас.

Если, однако, вы пишите свой собственный метод — например кастомную валидацию — то ее совершенно обязательно протестировать. Например, следующая валидация проверяет, что в позиции заказа (`Position`) количество единиц товара (`#quantity`) не может быть больше 10 для телевизоров, больше 15 для компьютеров и может быть любым для всех остальных товаров:

```
class Position  ActiveRecord::Base

  ...
  validate :validates_quantity_for_various_items

  private

    def validates_quantity_for_various_items
      if item.name == "TV"  quantity > 10
        errors.add(:quantity, "Can't have more than 10")
      elsif item.name == "PC"  quantity > 15
        errors.add(:quantity, "Can't have more than 15")
      end
    end

end
```

Здесь мы создали свой собственный метод, который стал валидацией. Чтобы убедиться, что эта валидация работает, нам необходимо написать тест:

```
describe Position do

  ...

  it "validates quantity for various items" do
    tv = Item.create(name: "TV", ...)
    pc = Item.create(name: "PC", ...)
    position1 = Position.create(item_id: tv.id, quantity: 11, ...)
    position2 = Position.create(item_id: pc.id, quantity: 16, ...)
    position1.errors[:quantity].should_not be_blank
    position2.errors[:quantity].should_not be_blank
  end

end
```

В тесте мы сначала создали два товара — телевизор и компьютер — и затем попытались добавить их в корзину, создав позиции для каждого товара. Так как в поле `#quantity` были указаны недопустимые значения, наша модель должна была добавить ошибки в `#errors` — что мы, собственно, и проверяем.

Также следует писать тесты в тех случаях, когда мы комбинируем использвание тех или иных стандартных методов. Предположим мы написали специальный метод `Item#destroy_with_orders`, который в отдельных случаях будет одновременно с удалением товара удалять и все заказы, в которые товар был добавлен, если это была последняя позиция в этом заказе:

```
class Item  ActiveRecord::Base

  has_many :positions
  has_many :orders, through: :positions

  def destroy_with_orders
    orders.each { |o| o.destroy if o.positions == 1 }
```

```
    self.destroy
  end

end
```

В этом примере мы использовали стандартный метод `#destroy` для удаления товара. Однако мы также скомбинировали его вместе с еще несколькими вызовами `#destroy`, теперь уже на объекте класса `Order`. В итоге у нас получился свой собственный метод `#destroy_with_orders`, а это означает, что его нужно протестировать:

```
describe Item do

  ...

  it "destroys each order for this item while destroying the item
  itself if order has no other items in it" do
    o1 = Order.create(...)
    o2 = Order.create(...)
    i1 = Item.create(...)
    i2 = Item.create(...)

    # Один товар в первом заказе
    o1.items  i1
    # Два товара во втором заказе
    o2.items  i1
    o2.items  i2

    i1.destroy_with_orders

    Order.find(i1.id).should     be_nil
    Order.find(i1.id).should_not be_nil
  end

end
```

Наконец, в качестве примера, я приведу здесь код реальной модели из одного из моих Rails-приложений, а также файл с тестами для нее. В тестовом коде используются *factories* (фабрики), речь о которых пойдет в разделе Использование фабрик. Кроме этого момента, остальной код должен быть вам понятен:

код модели Answer в одном из реальных Rails-приложений

```
class Answer  ActiveRecord::Base

  attr_accessible :user, :question_id, :body

  belongs_to :question
  belongs_to :user

  has_many :votes, as: :votable, dependent: :delete_all
  has_many :comments, as: :commentable, dependent: :delete_all

  validates :question, :user, presence: true
  validates :body, length: { minimum: 20 }
```

```
  validate  :validates_uniqueness_of_question_and_user, on: :create

  after_create do
    transaction do
      user.increment!(:answers_count)
      question.increment!(:answers_count)
    end
    QuestionMailer.new_answer(self).deliver if question.user
    question.user.notifications.answers_to_my_questions
  end

  after_destroy do
    transaction do
      user.increment(:answers_count, -1)
      user.increment(:votes_count, -votes_count)
      user.save
      question.correct_answer_id = nil
      question.save
    end
  end

  def set_correct!
    transaction do
      write_attribute(:correct, true)
      if question.correct_answer
        question.correct_answer.undo_correct!(save_question: false)
      end
      question.correct_answer = self
      self.save
      question.save
      user.increment!(:correct_answers_count)
    end
  rescue ActiveRecord::RecordInvalid => invalid
    raise invalid, "Something went wrong when trying to set the
    correct Answer to the Question ##{question.id}"
  end

  def undo_correct!(options={save_question: true})
    transaction do
      write_attribute(:correct, false)
      question.correct_answer = nil
      self.save
      question.save if options[:save_question]
      user.increment!(:correct_answers_count, -1)
    end
  rescue ActiveRecord::RecordInvalid => invalid
    raise invalid, "Something went wrong when trying to undo the
    correct Answer the Question ##{question.id}"
  end

  private

    def validates_uniqueness_of_question_and_user
      errors.add(:base, I18n.t("models.answer.errors.uniqueness"))
      if Answer.where(user_id: user_id, question_id: question_id)
      first
    end

end
```

пример тестов для модели Answer

```
require 'spec_helper'

describe Answer do

  before(:each) do
    @a = FactoryGirl.create(:answer)
  end

  it "updates user answers_counter when creating an answer" do
    @a.user.answers_count.should == 1
  end

  it "updates user answers_counter when destroying the answer" do
    @a.destroy
    @a.user.answers_count.should == 0
  end

  it "updates user's votes_count decreasing it to the amount of
  votes given for this answer" do
    10.times { Vote.create(user: FactoryGirl.create(:user), votable_id:
    @a.id, votable_type: "Answer") }
    another_answer = FactoryGirl.create(:answer, user: @a.user)
    2.times { Vote.create(user: FactoryGirl.create(:user), votable_id:
    another_answer.id, votable_type: "Answer") }
    @a.reload.destroy
    @a.user.reload.votes_count.should == 2
  end

  it "does not allow user to create a second answer to the same
  question" do
    Answer.create(question_id: @a.question.id, user: @a.user,
    body: String.random(21)).errors.should_not be_empty
  end

  describe "sets correct answer" do

    it "to itself in the associated question if it is marked as
    correct" do
      @a.set_correct!
      Question.find(@a.question.id).correct_answer.should == @a
    end

    it "to nil in the associated question if it is UNmarked as
    incorrect" do
      @a.set_correct!
      @a.undo_correct!
      Question.find(@a.question.id).correct_answer.should be_nil
    end

    it "to itself and UNmarks the previously correct answer" do
      @a.set_correct!
      a2 = FactoryGirl.create(:answer, question: @a.question)
      a2.set_correct!
      @a.correct.should be_false
      Question.find(@a.question.id).correct_answer.should == a2
    end

    it "and updates user's correct_answers_count" do
```

```
      @a.set_correct!
      User.find(@a.user.id).correct_answers_count.should == 1
      @a.undo_correct!
      User.find(@a.user.id).correct_answers_count.should == 0
    end

    it "to nil when the answer is destroyed" do
      @a.set_correct!
      @a.destroy
      Question.find(@a.question.id).correct_answer_id.should be_nil
    end

  end

end
```

Тестирование контроллеров

Тестирование контроллеров несколько отличается от обычного unit-тестирования. Мы точно также будем тестировать отдельные методы в классе — в случае контроллеров это экшены — только способ, которым мы будем вызывать эти методы, будет несколько другим. Мы не можем просто взять и создать экземпляр контроллера и затем вызвать на нем нужный экшен. Следующий код в тесте будет неправильный:

Это неправильный способ вызвать метод в контроллере

```
items_controller = ItemsController.new
items_controller.create
```

Экземпляры контроллеров в Rails должны создаваться специальным способом, когда в Rails поступает запрос из браузера. В случае тестов, мы как бы симитируем такой запрос и заставим Rails думать, что пользователь зашел на определенную страницу и выполнил определенное действие.

Вспомним наш RESTful контроллер `ItemsController` и экшен `#create` в нем:

```
class ItemsController

  ...

  def create
    @item = Item.create(params[:item])
    if @item.errors.empty?
      redirect_to @item
    else
      render "new"
    end
  end

end
```

Так как этот код писали мы сами, нам нужно создать к нему тесты, которые проверят следующие два сценария:

1. Если валидации при создании модели прошли (т.е. в форме нет ошибок), контроллер должен сделать редирект на страницу товара.
2. Если валидации не прошли, контроллер должен отрендерить шаблон `new.html.erb`.

Создадим файл `spec/controllers/items_controller_spec.rb` и приступим к написанию тестов.

Начнем со сценария (1). Вспомним, что чтобы запрос пользователя отправился в экшен `#create` нам необходимо использовать POST-запрос. Именно так — `post` — называется в Rspec Rails один из специальных методов[1], которыми можно симитировать настоящий запрос пользователя к контроллеру. Таким образом, файл с тестами на данный момент примет следующий вид:

файл spec/controllers/items_controller_spec.rb

```
require 'spec_helper'

describe ItemsController do

  it "redirects to the item's page after the item is created" do
    post :create
  end

end
```

В качестве аргумента метод `#post` принимает название экшена, в который будет отправлен POST-запрос. Однако этого недостаточно. Нам надо как-то передать данные воображаемой формы, которую наш воображаемый пользователь заполнил. Для этой цели, мы через запятую передадим методу `#post` хэш, который в контроллере автоматически превратится в `params`:

```
describe ItemsController do

  it "redirects to the item's page after the item is created" do
    post :create, item: { name: "TV", description: "flatscreen TV",
                          price: 10000, weight: 10 }
  end

end
```

Причем слово `item` идущее после запятой является единственным ключем передаваемого хэша, а значением для этого ключа является вложенный хэш. То есть мы могли бы

поставить дополнительные фигурные скобки и ничего бы не изменилось:

```
post :create, { item: { name: "TV", description: "flatscreen TV",
                        price: 10000, weight: 10 }}
```

Как вы помните из главы Контроллеры и ресурсы, когда мы создаем в контроллерах новую модель, мы передаем методу `#create` хэш из атрибутов. В нашем случае в контроллере мы имеем запись `@item = Item.create(params[:item])` . Таким образом в `params[:item]` у нас окажутся как раз те атрибуты, которые мы указали во вложенном хэше в тесте.

И так, запрос сконструирован. Остается понять, как проверить поведение контроллера. Для этого Rspec Rails предоставляет специальный метод `#response`, который возвращает объект с информацией о том, что ответил контроллер. Используя стандартный матчер[2] Rspec Rails `redirect_to` мы можем проверить то, что собирались проверить в сценарии (1):

```
describe ItemsController do

  it "redirects to the item's page after the item is created" do
    post :create, item: { name: "TV", description: "flatscreen TV",
                          price: 10000, weight: 10 }
    response.should redirect_to(assigns(:item))
  end

end
```

В этом выражении вам должно быть понятно все, кроме `assigns(:item)`. Метод `assigns` просто позволяет нам получить доступ к инстансным переменным внутри тестируемого контроллера. В данном случае, нам нужен был доступ к переменной `@item`, потому что без нее `redirect_to` не смог бы сгенерировать правильный путь. Чтобы было немного понятнее, я мог бы переписать все это следующим образом:

```
describe ItemsController do

  it "redirects to the item's page after the item is created" do
    post :create, item: { name: "TV", description: "flatscreen TV",
                          price: 10000, weight: 10 }
    @item = assigns(:item)
    response.should redirect_to(@item)
  end

end
```

Вот теперь наш тест готов и должен успешно выполниться.

Приступим ко второму тесту и сценарию (2). В нем, чтобы произошел рендер шаблона `new.html.erb`, нам нужно передать в контроллер невалидные значения для полей. Например мы знаем, что в нашей модели есть валидация, проверяющая наличие поля `#name`, поэтому чтобы спровоцировать модель на непрохождение этой валидации, мы сделаем запрос в контроллер не указав значение для `#name`:

```
describe ItemsController do

  it "renders 'new' template if valiadtions fail after trying
  to create an item" do
    post :create, item: { description: "flatscreen TV",
                          price: 10000, weight: 10 }
    response.should render_template("new")
  end

end
```

Здесь мы использовали стандартный матчер Rspec Rails `#render_template` и, как несложно догадаться, он проверяет, что именно отрендерил контроллер.

Как видно из этого примера, при тестировании контроллера мы проверяем только что сделал сам контроллер — то есть куда произошел редирект или какой шаблон был отрендерен. Мы не проверяем, сохранилась ли модель и какие ошибки были в нее добавлены. Это не забота тестов для контроллера.

Ниже я приведу пример контроллера `QuestionsController` и spec-файла для него в одном из моих Rails-приложений. Из него видно, что мне нравится группировать тесты для каждого экшена в отдельный вложенный `#describe`. Я также довольно часто использую в тестах для контроллера *factories*, речь о которых пойдет в следующем разделе.

код контроллера QuestionsController в одном из реальных Rails-приложений

```
class QuestionsController  ApplicationController

  before_filter :authenticate_user!,        only: [:new, :edit,
  :create, :update, :destroy]
  before_filter :find_question,             only: [:show, :edit,
  :update, :destroy]
  before_filter :check_modify_permissions!, only: [:edit, :update,
  :destroy]
  before_filter :find_section_by_subdomain, only: [:index, :show,
  :new]
  before_filter :find_sections,             only: [:new, :edit]

  def index
    order = (params[:asc] ? "ASC" : "DESC")
    @questions = Question.order("created_at #{order}").includes(:user)
    paginate(:page => params[:page])
    @questions = @questions.where(section_id: @section.id) if @section
    @questions.paginate(page: params[:page], per_page: 30)
  end

  def show
    @section ||= @question.section
    render_404 and return if @section  (@section != @question.section)
    @answer  = Answer.new(question_id: @question.id)
    @answers = Answer.where(question_id: @question.id).includes(:user)
    order("votes_count DESC").all(include: { comments: :user })
  end
```

```
def new
  @question = Question.new(section_id: @section.try(:id), screencast_id:
  params[:screencast_id])
end

def create
  @question = Question.create(params[:question].merge({user:
  current_user}))
  if @question.errors.empty?
    flash[:success] = t("controllers.questions.success.create")
    redirect_to @question
  else
    find_sections
    flash.now[:error] = t("controllers.questions.errors.create")
    render "new"
  end
end

def edit
end

def update
  @question.update_attributes(params[:question])
  if @question.errors.empty?
    flash[:success] = t("controllers.questions.success.update")
    redirect_to @question
  else
    flash.now[:error] = t("controllers.questions.errors.update")
    render "edit"
  end
end

def destroy
  @question.destroy
  flash[:success] = t("controllers.questions.success.destroy")
  if !params[:reason].blank?  @question.user  @question.user
  notifications.answers_to_my_questions
    QuestionMailer.question_deleted(@question, params[:reason])
    deliver
  end
  redirect_to questions_path
end

private

  def find_question
    @question = Question.where(id: params[:id]).first
    render_404 unless @question
  end

  def check_modify_permissions!
    unless question_has_modify_permissions?
      render_403
    end
  end

  def find_sections
    @sections = Section.where(active: true).map { |s| [s.id,
    s.title] }
```

```
    end

end
```

```
require 'spec_helper'

describe QuestionsController do

  describe "index action" do

    before(:each) do
      @section = FactoryGirl.create(:section)
      10.times { FactoryGirl.create(:question, section: @section) }
      10.times { FactoryGirl.create(:question) }
    end

    it "paginates questions for the current section" do
      @request.host = "#{@section.subdomain}.hasBrains.local"
      get :index
      assigns(:questions).should have(10).questions
    end

    it "paginates questions for all sections" do
      get :index
      assigns(:questions).should have(20).questions
    end

  end

  describe "show action" do

    before(:each) do
      @question = FactoryGirl.create(:question)
    end

    it "shows question's page" do
      get :show, id: @question.id
      response.should render_template("show")
    end

    it "renders 404 if the question is found but the section is
    wrong" do
      another_section = FactoryGirl.create(:section)
      @request.host = "#{another_section.subdomain}.hasBrains.local"
      get :show, id: @question.id
      response.should render_404
    end

  end

  describe "create action" do

    before(:each) do
      login_user
    end

    it "redirects to question's page if validations pass" do
      post :create, question: FactoryGirl.attributes_for(:question)
      merge({section_id: FactoryGirl.create(:section).id})
```

```
      response.should redirect_to(assigns(:question))
    end

    it "renders #new form if validations fail" do
      post :create, question: FactoryGirl.attributes_for(:question)
      merge({title: ""})
      response.should render_template("new")
    end

  end

  describe "update action" do

    before(:each) do
      @question = FactoryGirl.create(:question)
      sign_in(@question.user)
    end

    it "redirects to question's page if validations pass" do
      put :update, id: @question.id, question: { title: "A different
      title for the question" }
      response.should redirect_to(@question)
    end

    it "renders #edit form if validations fail" do
      put :update, id: @question.id, question: { title: "" }
      response.should render_template("edit")
    end

  end

  describe "destroy action" do
    it "redirects to questions index after it destroys a question" do
      question = FactoryGirl.create(:question)
      sign_in(question.user)
      delete :destroy, id: question.id
      response.should redirect_to(questions_path)
    end
  end

  after(:all) { Section.delete_all }

end
```

[1] Помимо метода `#post` в тестах для контроллеров доступны также методы `#get`, `#put` и `#delete`. Каждый из них выполнит соответствующий своему названию HTTP-запрос.

[2] В документации к Rspec Rails можно прочитать все о стандартных матчерах, которые доступны для тестирования контроллеров в Rails [2].

[1] Документация по тестированию контроллеров с помощью Rspec Rails

[2] Стандартные матчеры для тестирования контроллеров в Rspec Rails

Использование фабрик

В примерах кода в предыдущих разделах вы наверняка заметили, что я использовал странный класс `FactoryGirl`. Он стал доступен в моих спеках благодаря тому, что я установил гем, который называется `factory_girl_rails` [2] — то же самое понадобится сделать и вам. Этот класс позволит нам быстро создавать объекты ActiveRecord без монотонного перечисления всех полей и их значений — то есть, по сути, `FactoryGirl` представляет из себя *фабрику* (factory) объектов.

Сначала давайте разберемся, зачем нам нужны фабрики. Напишем небольшой тест для экшена `ItemsController#show`, который будет проверять, что этот экшен рендерит шаблон `show.html.erb`:

```
describe ItemsController do

  ...

  it "renders 'show' if the item is found" do
    get "show", id: 1
    response.should render_template "show"
  end

end
```

Проблема в данном случае заключается в том, что, конечно же, в нашей БД нет записи о товаре с id 1 (это тестовая БД, которая очищается перед каждым запуском тестов!) и тест зафэйлится. Поэтому, перед тем как делать запрос к контроллеру с помощью метода `#get`, нам необходимо создать товар. Вот такой тест успешно пройдет:

```
describe ItemsController do

  ...

  it "renders 'show' if the item is found" do
    item = Item.create(name: "Car", description: "really fast",
    price: 10000, weight: 950)
    get "show", id: item.id
    response.should render_template("show")
  end

end
```

Все бы хорошо, но в спеке для нашего контроллера много разных тестов. Для некоторых из них — в частности для тестов к экшенам `#edit`, `#update`, `#destroy` и `#index` — потребуются существующие записи в БД. Каждый раз создавать эти объекты вручную, прописывая значения для всех атрибутов — неудобно. Мы, конечно, могли бы вынести создание товара в хук `#before(:each)`, но этого будет недостаточно в случае, если нам понадобится создать товары не только в тестах к контроллеру `ItemsController`, но также и в других тестах (что вполне вероятно). Тут нам на помощью придут фабрики.

Мы создадим в директории `spec/factories` файл `factories.rb` и добавим в него следующий код:

```
FactoryGirl.define do
  factory :item do

    name        "Car"
    description "very fast"
    price       10000
    weight      950

  end
end
```

В принципе этот код говорит сам за себя — не вдаваясь в подробности DSL-синтаксиса `FactoryGirl`, можно понять, чего мы пытаемся добиться этим кодом. Все что нам остается сделать теперь для создания нового объекта, это использовать специальный метод `FactoryGirl.create` внутри нашего теста:

```
describe ItemsController do

  ...

  it "renders 'show' if the item is found" do
    get "show", id: FactoryGirl.create(:item).id
    response.should render_template("show")
  end

end
```

Мы убрали одну лишнюю строку и создали объект прямо при отправке запроса[1].

Кроме того, мы могли бы вынести создание объекта в `#before(:each)` именно так, как мы и хотели сделать ранее:

```
describe ItemsController do

  ...

  before(:each) do
    @item = FactoryGirl.create(:item)
  end

  it "renders 'show' if the item is found" do
    get "show", id: @item.id
    response.should render_template("show")
  end

end
```

Теперь переменная `@item` будет доступна внутри каждого теста в нашем файле и перед началом каждого теста в нее будет записан совершенно новый свежесозданный товар.

В этом разделе мы затронули только верхушку айсберга гема `factory_girl` и я

рекомендую вам обязательно изучить документацию [1]. Полезность гема `factory_girl` при написании тестов сложно переоценить.

[1] На самом деле, по правилам языка Ruby, порядок выполнения таков: сначала выполнится `FactoryGirl.create`, потом метод `#id` вернет присвоенный id нового объекта, только затем выполнится метод `#get`

[1] *Страница гема factory_girl на GitHub*

[2] *Страница гема factory_girl_rails на GitHub*

Test Driven Development

Когда вы почувствуете себя увереннее в работе с Rails и когда вы более менее освоитесь с написанием тестов, я рекомендую вам попробовать подход к разработке, который называется TDD — Test Driven Development. Суть его очень проста: вы должны будете сначала написать тестовый код, убедиться, что он фейлится, и только потом написать основной код, который заставит тест выполниться успешно.

У такого подхода есть несколько основных преимуществ:

- **Вы не забудете и не поленитесь написать тест**. Программисты, как известно — довольно ленивы. И если иногда это играет нам на руку, если мы, например, пытаемся автоматизировать какую-то монотонную задачу, то в случае с написанием тестов — лень нам точно вредит. Например, мы можем написать небольшой метод и подумать, что его вовсе не нужно тестировать. Это не так: даже простой с виду код следует тестировать, потому что в будущем другие части программы могут повлиять на результат выполнения этого кода непредсказуемым способом. Поставив на первое место написание теста, мы сводим к нулю вероятность того, что лень нам как-то помешает.
- **Вы не "подгоняете" тест под код, написанный в методе.** Еще один соблазн, который существует — "подогнать" тест и написать его таким образом, чтобы он выполнялся успешно с тем кодом, который у вас написан. При написании тестов, однако, следует помнить, что вы должны проверять и предсказывать поведение вашего кода, а не пытаться его угадать. Когда вы первым делом пишите тест, вы на самом деле пишите спецификацию поведения вашей программы, то есть предсказываете как она должна себя вести. "Угадать" поведение программы без наличия кода логически невозможно, т.к. угадывать нечего.
- **Вы начинаете лучше понимать свою собственную программу и у вас**

появляется план действий. Без заранее написанных тестов у вас гораздо больше свободы отступить от плана и сделать что-нибудь неправильно или необдуманно.

TDD — не универсальная идеология разработки. Многим программистам она не нравится. Я нахожу ее относительно полезной, но не использую слишком фанатично. Если вы хотите подробнее узнать о TDD и попытаться построить свою работу в соответствии с принципами TDD, в списке литературы приведено несколько ссылок.

[1] Статья о TDD в Википедии (на английском)

[2] Статья о TDD в Википедии (на русском)

[3] Вопрос о TDD в Ruby на StackOverflow: ответы пользователей + несколько полезных ссылок на дополнительные материалы

[4] Статья "The Three Rules Of TDD"

[5] Список книг, посвященных TDD (внимание, примеры в этих книгах обычно не на языке Ruby!)

Глава 9. Руби-гемы, Bundler и Github

Кто и зачем создает гемы?

И так, вы работаете над своим Rails-приложением. Вашему Rails-приложению нужно какое-то решение для того, чтобы выводить меню: аккуратно подсвечивать активные пункты меню, следить за иерархией и т.д. Вы пишите несколько классов, которые справляются именно с этой задачей, кладете их в папку `/lib` и, затем, со спокойной совестью идете спать. На следующий день вам поступает новый заказ на создание другого сайта, где тоже требуется меню. Вы, конечно же, копируете нужные файлы из вашего первого веб-приложения и — все готово. Но тут, во время тестирования второго сайта, вы замечаете небольшой баг в своем решении, который не проявлялся в работе первого сайта. Вы, конечно же, исправляете его, но теперь вам нужно скопировать исправленные сайты из второго приложения в первое, чтобы этот баг был исправлен и там. Несложно представить себе ситуацию, когда таких приложений не два, а пять или десять — и вот уже обычное копирование файлов занимает значительное время и отнимает силы.

Вместо ручного копирования файлов, вы могли бы сделать из вашего решения официальную библиотеку, которую затем смогут использовать другие программисты в работе над своими проектами. В мире Ruby такие библиотеки называются *гемами* (gems). Если создать из вашего решения гем[1], то, вместо ручного копирования отдельных файлов в каждый проект, вам нужно будет просто указать название гема в файле `Gemfile` в корне вашего Rails-приложения. Если вы выпускаете новую версию гема (например с исправленными ошибками), в каждом проекте вам достаточно будет запустить команду `bundle update`, которая скачает все необходимые файлы обновленного вами гема.

Помимо преимущества автоматизации установки и обновления, вы также даете возможность другим программистам участвовать в развитии библиотеки. Возможно, кому-то ваш гем покажется необходимым, но его функционал будет устраивать не полностью. Тогда этот программист сможет добавить свой собственный код в ваш гем и попросить вас "принять" этот код. В итоге, выигрывают все: вы получаете новый функционал и можете использовать его в своих проектах; программисту, который добавил новый функционал, не пришлось изобретать решение с нуля — он просто добавил немного кода к вашему гему. Нередка также ситуация, когда кто-то находит ошибку в чужом геме и исправляет ее — в итоге снова выигрывают все: и автор гема, и тот, кто исправил ошибку, и те, кто этот гем используют.

Пример с решением для создания меню в Rails-приложении — вовсе не выдуманный. В какой-то момент мне нужно было сделать именно это и ни один из существующих гемов не удовлетворял моим требованиям. В итоге, я написал свой собственный гем mmmenu, выложил его на Github и люди начали им пользоваться, дополнять и вносить исправления.

Таким образом, если попытаться суммировать все сказанное, то:

- Гемы создаются программистами, работающими над своими приложениями
- Гемы рождаются из решений, которые можно было бы использовать в нескольких приложениях.
- Другие программисты, которые используют гем, имеют возможность вносить в него исправления и дополнения, тем самым принося пользу всем участникам.

[1] Тема создания собственных гемов выходит за рамки этой книги, однако вы можете прочитать несколько статей, посвященных этому вопросу — ссылки на статьи приведены в списке литературы к этом разделу.

[1] *Статья "Make your own gem"*

[2] *Статья "Making Ruby Gems"*

[3] *Статья "Ruby gems — what, why and how"*

[4] *Гем "jeweler", который помогает облегчить процесс создания ruby-гемов*

Как найти гем с нужным мне функционалом?

Для большинства задач, которые может потребоваться решить новичку, уже существуют гемы, написанные и протестированные другими программистами. Первое правило, которому вы должны следовать, когда вам нужно решить какую-то проблему в вашем приложении — задайте себе вопрос, мог ли кто-то другой столкнуться с похожей проблемой и придумать решение. Например, если вы хотите сделать регистрацию/аутентификацию пользователей на своем сайте, велика вероятность того, что та же задача стояла перед многими другими разработчиками веб-приложений. А значит, наверняка есть готовое решение.

Чтобы проверить гипотезу о существовании готового решения в виде Ruby-гема, первое место, куда вам нужно отправиться — это сайт The Ruby Toolbox [1]. Предположим, вы хотите найти гем для аутентификации. Тогда вы просто вводите в поле поиска сверху на сайте запрос Rails authentication и первым в результатах поиска окажется самый популярный гем, соответствующий вашему запросу — на момент написания этой книги этим гемом

является `Devise`.

Если вы не можете найти ничего на сайте "The Ruby Toolbox", возможно вам стоит спросить о том, существует ли какое-нибудь подобное решение на сайте StackOverflow. Вот пример вопроса на этом сайте, посвященного поиску решения в виде Ruby-гема.

Наконец, не забудьте попробовать поиск в Google. Также убедитесь, что вы прошлись по всем способам, перечисленным в главе Что дальше: как разбираться в проблемах и кому задавать вопросы?

После того, как вы нашли нужный вам гем, обязательно прочтите документацию к этому гему, перед тем как его устанавливать. Убедитесь, что он совместим с используемой вами версией Rails.

[1] Сайт "The Ruby Toolbox" - поиск гемов по ключевым словам

Что такое Github?

Github — это сайт, где программисты могут выкладывать в открытый доступ свои git-репозитории. Теоретически, сделать это можно было бы полностью самостоятельно, без такого сайта. Однако Github стал де-факто стандартом в мире Ruby и Rails и практически все Ruby-гемы выложены также и на Github-е.

Сайт предоставляет несколько важных функций, которые разработчики активно используют. Во-первых, используя сайт, можно с удобством "клонировать" репозитории, вносить в них исправления или дополнения и затем предлагать эти изменения автору исходного репозитория. Помимо этого, к каждому репозиторию прилагаются страницы Wiki и Issues. Они не обязаны быть ничем заполнены, но, как правило (особенно для популярных гемов), на них можно найти документацию (Wiki) и список багов и обсуждений к ним (Issues).

Ко всем репозиториям выводится статистика: можно посмотреть сколько людей добавили его в избранное (Star), сколько людей "склонировали" репозиторий (Fork), а также посмотреть когда был последний коммит. Последнее особенно важно для того, чтобы понять, насколько устарел код в репозитории и поддерживается ли этот репозиторий автором в настоящее время. Если последний коммит был год назад — это повод задуматься о том, насколько стабильно и безопасно будет работать этот гем с вашим приложением. Для некоторых гемов последний коммит не имеет большого значения и они прекрасно работают с новыми версиями Rails. Для других гемов, постоянная поддержка критична и необходима.

Помимо всего прочего, на главной странице любого репозитория на Github по умолчанию выводится файл README, в который обычно помещают самую необходимую документацию по работе с этим гемом. Поэтому, если вы хотите разобраться как

использовать тот или иной гем, первым местом, где следует искать информацию, является страница гема на Github.

Установка гемов и Bundler

Установить гемы в систему можно двумя основными способами: с помощью команды `gem install` и через Bundler. Первый способ используется редко и, в основном, для гемов, которые не имеют отношения к вашему Rails приложению. Второй способ является стандартным для всех гемов, которые вам нужны для работы вашего Rails-приложения. Рассмотрим оба варианта.

Установка гемов с помощью gem install

Этот способ может вам понадобиться, когда вы, например, устанавливаете самые основные гемы, требующиеся вам для разработки. Например гем `rails` или гем `bundler` (который, кстати, обеспечит работу следующего способа установки гемов). Если вы используете RVM [3] (а я надеюсь, что это так!), вам достаточно выполнить в терминале следующую команду:

```
gem install bundler
```

для установки гема `bundler` и всех его зависимостей (других гемов, которые используются этим гемом). Программа `gem` подключится к удаленному хранилищу гемов, скачает гем и установит его в систему.

Все установленные гемы, опять таки в случае, если вы используете RVM, будут находиться в вашей домашней директории в специальной папке `~/.rvm/gems/ruby-[version]/gems/`, где `[version]` — ваша активная версия Ruby. Не редактируйте эту папку вручную.

Если вам понадобится удалить гем из системы, вам достаточно выполнить следующую команду:

```
gem uninstall bundler
```

Это удалит гем `bundler` из системы.

Установка гемов с помощью Bundler-а

Практически в любом приложении написанном на Ruby, используются раздличные Ruby-

гемы. Уж тем более не является исключением Rails-приложение. Каждый гем, в свою очередь, может иметь свои собственные зависимости. Чем их больше и чем больше гемов в приложении, тем сложнее за ними уследить. Ко всему прочему, если над приложением работают несколько людей, может сложиться ситуация, когда у одного человека на машине установлены все нужные гемы, а у другого их нет и приложение не запускается.

До появления Bundler-а, разработчики просто перечисляли друг другу в письмах или устно, какие гемы требуется установить в систему, чтобы данное приложение без проблем запустилось. Bundler позволил автоматизировать этот процесс. Теперь в любом Rails-приложении существует специальный файл — `Gemfile`. В нем, используя простой синтаксис, перечислены гемы, которые необходимы для работы данного приложения. Вот пример содержимого `Gemfile` в только что созданном Rails приложении:

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

gem 'rails', '3.2.11'

# Bundle edge Rails instead:
# gem 'rails', :git => 'git://github.com/rails/rails.git'

gem 'sqlite3'

# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails',   '~> 3.2.3'
  gem 'coffee-rails', '~> 3.2.1'

  # See https://github.com/sstephenson/execjs#readme for more
  supported runtimes
  # gem 'therubyracer', :platforms => :ruby

  gem 'uglifier', '>= 1.0.3'
end

gem 'jquery-rails'
```

Как видите — все довольно просто. После слова `gem` в кавычках идет название гема, а через запятую, снова в кавычках — его версия.[1]

Находясь в корневой директории Rails приложения, вы можете выполнить команду `bundle install` и Bundler автоматически скачает и установит все гемы, перечисленные в файле Gemfile[2]. Причем установит он их в ту же самую папку `~/.rvm/gems/ruby-[version]/gems/` (если явно не указано другое место). Таким образом, гемы использующиеся вашим Rails приложением, будут автоматически установлены и к вам в систему. Если вам потребуется добавить какой-то гем к вашему приложению, просто добавьте его в `Gemfile` и запустите `bundle install` еще раз — в этот раз Bundler скачает и установит только добавленный гем.

Вам также может потребоваться обновить версии установленных гемов. В Rails-приложении это рекомендуется делать относительно регулярно. Команда `bundle update` скачает и установит новые версии гемов, для которых в `Gemfile` не была указана конкретная версия. Если вы указываете в `Gemfile` конкретную версию, например:

```
gem 'rails', '3.2.11'
```

то Bundler думает, что вы хотите всегда использовать именно ее и не обновляет гем и его зависимости.

Таким образом, другому программисту — вашему коллеге, который скачал к себе на машину ваше приложение — достаточно будет один раз запустить команду `bundle install` и все нужные гемы будут установлены к нему в систему. После чего он сможет без проблем запустить приложение у себя на локальной машине.

[1] На самом деле в Gemfile используется язык Ruby, а `gem` — это метод, которому передаются аргументы (обычно — название гема и версия).

[2] Команда `bundle install` автоматически запускается во время создания Rails-приложения, так что если вы не добавляли ничего нового в `Gemfile`, запускать команду повторно не имеет смысла.

 *[1] Официальный сайт RubyGems.org (хранилище Ruby-гемов)*

 *[2] Официальный сайт гема Bundler*

 *[3] Официальный сайт утилиты RVM (Ruby Version Manager)*

Как научиться пользоваться гемом: читайте документацию

Мой опыт обучения людей Ruby On Rails показал, что многие новички испытывают сложности с использованием различных гемов именно по той причине, что они не осознают важность чтения документации. Первое, что вы должны сделать перед установкой гема — открыть страницу гема на Github-е и внимательно прочесть файл README (он отображается на странице репозитория по умолчанию, под списком файлов и папок).

Как правило в README к каждому гему могут содержаться следующие инструкции:

- Как установить гем в Rails-приложение (обычно — добавить строку в `Gemfile`)
- Как сгенерировать в Rails-приложении необходимые гему файлы
- Какие строки следует добавить в ваши контроллеры или модели для использования гема.

Если вы вдруг поймаете себя на том, что не понимаете о чем идет речь в документации по установке и использованию гема — это скорее всего связано с тем, что вы еще не до конца понимаете устройство Rails. В этом случае, попытайтесь выделить ключевые слова в предложениях, которые вам непонятны, и попробовать найти соответствующие темы на сайте Rails Guides или в Google. Также, иногда помогает задать вопрос касающийся работы с гемом, в соответствующей Группе Google (ссылка на группу может быть указана прямо в файле README).

Глава 10. Настройка сервера и деплой приложения

В этой главе мы рассмотрим, как запустить ваше приложение на настоящем (боевом) сервере. Для полного понимания этой главы вам потребуются некоторые знания ОС Linux и командной строки. В списке литературы к этой главе приведены материалы, которые помогут вам приобрести эти знания.

Выбор хостинг-провайдера

И так, вы написали свое приложение и настало время показать его всему интернетику. Как вы уже догадались, вам понадобится сервер, на котором это приложение будет запущено. Это должен быть сервер под управлением одного из дистрибутивов ОС Linux (я предпочитаю дистрибутив Ubuntu Server). Сам сервер, скорее всего, будет не совсем настоящий. Дело в том, что для запуска небольших приложений, нагрузка на которые пока будет невелика (т.е. у вас вряд ли сразу появится миллион пользователей) достаточно т.н. VPS — Virtual Private Server. Суть этого понятия заключается в следующем: представьте себе один большой мощный сервер; на этом сервере установлено одновременно несколько, возможно одинаковых, операционных систем и все они одновременно запущены. Каждая ОС принадлежит какому-то одному человеку, то есть получается, что один сервер делят между собой несколько клиентов хостинг-провайдера. Получается, что у каждого клиента есть собственная ОС со своими собственными настройками, и клиенты никак друг от друга не зависят.

В дальнейшем, если нагрузка на ваше приложение возрастет, вы всегда сможете "переехать" на настоящий сервер, заняв его целиком. У вас также будет возможность забрать себе больший кусок виртуального сервера — заплатить дополнительные деньги за большее количество оперативной памяти, дискового пространства и т.д. Как бы там ни было, на данный момент мы сосредоточимся на поиске приемлемого хостинга, предоставляющего VPS начального уровня.

Я бы рекомендовал выбирать хостинг-провайдеров за рубежом и на то есть несколько причин. Во-первых, цены за рубежом зачастую ниже, за счет того, что инфраструктура в США и Европе помогает компаниям экономить больше денег. Во-вторых, если ваш сайт станет популярным, тот факт, что вы держите информацию на иностранном сервере, станет дополнительной страховкой — в нашей стране, как известно, маски-шоу еще никто не отменял. В третьих, у известных хостинг провайдеров за рубежом прекрасная служба

поддержки и коммьюнити — мне лично это не раз помогало решить очень срочные проблемы с сервером (многие из которых, кстати, возникали по моей глупости).

Я уже долгое время (на данный момент — 5 лет) пользуюсь VPS-хостингом Linode.com и могу порекомендовать их как надежный, удобный и очень недорогой хостинг [1]. Стоимость самого дешевого VPS — 512 RAM, 24Gb места на диске, 200Gb трафика — на данный момент составляет $20 в месяц. На Linode.com прекрасная служба поддержки, форум пользователей, а также IRC-канал с клиентами и администраторами, которым можно задать быстрый вопрос о хостинге, настройке вашего приложения или ОС Linux. Кроме всего прочего, их сервера находятся не только в США, но и в Великобритании, что означает относительно невысокий пинг (время отклика) с Россией.

Помимо покупки VPS у вас есть альтернатива: приобрести подписку на сервис, который включает в себя готовую платформу для деплоя Rails-приложений. Таким, например, является Heroku. Это избавит вас от головной боли по настройке чего бы-то ни было (вы сможете пропустить это главу, ура!), но будет стоить гораздо дороже. Кроме того, рано или поздно вы все равно столкнетесь с проблемами, которые выходят за рамки решений, предоставляемых подобными сервисами, и вам понадобятся знания ОС Linux и Rails-стэка. На мой взгляд, гораздо выгоднее купить обычный VPS и при этом еще получить знания!

[1] Я привел ссылку на Linode.com с реферальным кодом. Если вы решите приобретать VPS у них, я буду вам благодарен, если вы зарегистрируетесь на их сайте, зайдя именно по этой ссылке. Тем самым вы поддержите мой проект hasBrains.ru с бесплатными скринкастами по Ruby, Ruby On Rails, Linux и другим темам.

Установка ОС и необходимого софта

Перед тем как ваше приложение сможет начать работать на сервере, сервер необходимо соответствующим образом настроить и установить на него весь необходимый софт. Если вы приобрели VPS от Linode.com, то установка самой ОС — дело двух минут. В веб-интерфейсе вам просто нужно будет выбрать операционную систему, которую вы хотите установить на вашу машину (берите Ubuntu Server) и это будет сделано автоматически, без каких либо дополнительных действий с вашей стороны.

Затем вы сможете подключиться к вашей машине используя логин `root` и пароль, который вы выбрали при установке системы:

```
ssh root@your-server-address
```

Выполнив эту команду, вы должны увидеть приветственное сообщение установленной системы и командную строку, ожидающую ввода новой команды — теперь мы готовы приступить к настройке сервера.

Пользователи, rsa-ключи и права доступа

Как и в любой другой ОС, в Linux есть такое понятие как пользователи. В первый раз мы подключились к вашему серверу с помощью пользователя `root`, который имеет неограниченные права. Мы будем делать так не всегда, а вернее сказать — почти никогда. Пользователь `root`, однако, будет нам полезен на время настройки сервера, чтобы нам каждый раз не приходилось набирать команду `sudo`.

Первым делом, я предлагаю создать нового пользователя, от имени которого будет запускаться наше приложение. Обычно, я предпочитаю называть этого пользователя `deploy`. Чтобы сделать это, вам нужно выполнить следующую команду:

```
useradd -m deploy
```

Флаг `-m` означает создание домашней директории пользователя `/home/deploy` — она нам понадобится через несколько секунд.

Создав пользователя `deploy`, нам нужно установить для него пароль. Это можно сделать командой `passwd deploy` — после нажатия `Enter` терминал попросит вас ввести новый пароль. Пароль нам понадобится использовать один раз — прямо сейчас — чтобы подключиться и настроить подключение пользователя с помощью rsa-ключей. Вам нужно открыть новое окно терминала, ввести там команду `ssh deploy@your-server-address` и, затем, только что придуманный пароль. Далее, все команды, которые нужно будет вводить во втором окне терминала от имени пользователя `deploy` я буду предварять префиксом `deploy$` — вам набирать этот префикс не нужно.

Кроме вышеперечисленных действий, нам не помешает сделать дефолтным шеллом для этого пользователя bash[1], а не `sh`. Вам нужно открыть файл `/etc/passwd` в текстовом редакторе, например в nano (командой `nano /etc/passwd`) и исправить следующую строку:

```
deploy:x:1001:1001::/home/deploy:/bin/sh
```

на

```
deploy:x:1001:1001::/home/deploy:/bin/bash
```

Сохранить и закрыть файл можно нажав Ctrl+X.

Первое, что нам потребуется сделать на сервере в качестве пользователя `deploy` — это создать пустую папку `~/.ssh` следующей командой:

```
deploy$ mkdir ~/.ssh
```

Чтобы иметь возможность подключаться к серверу без того, чтобы каждый раз вводить

пароль, нам необходимо сгенерировать *rsa-ключи* [2] и загрузить публичный ключ на сервер. Это нужно сделать на *локальном компьютере*. Если вы используете ОС Linux или MacOSX, то сгенерировать ключи можно следующим образом:

```
# не забудьте - эту команду нужно выполнить НА ВАШЕЙ ЛОКАЛЬНОЙ МАШИНЕ
# в окне терминала, который никуда не подключен!

ssh-keygen -t rsa
```

После ввода этой команды терминал задаст вам несколько вопросов — в частности, где должны находиться файлы с ключами, а также предложит вам выбрать passphrase. Все это можно смело проигнорировать и нажимать `Enter`. Тогда сгенерированные файлы будут по умолчанию находиться в директории `~/.ssh/` и называться `id_rsa` и `id_rsa.pub`.

И так, ключи сгенерированы. Один из них — приватный ключ `id_rsa` должен всегда находиться *только* на вашем компьютере — никуда его не загружайте и никому не показывайте. Второй файл — `id_rsa.pub` — это публичный ключ, который должен быть на сервере под именем `/home/deploy/.ssh/authorized_keys`. Самый простой способ отправить его на сервер, это использовать следующую команду:

```
cd ~/.ssh
scp id_rsa.pub deploy@your-server-address:/home/deploy/.ssh/authorized_keys
```

Выполнив эту команду, вернитесь во второй терминал (тот, в котором вы подключены к серверу под пользователем `deploy`) и наберите команду `deploy$ exit` — это заставит ssh отключиться от сервера. Затем попробуйте подключиться снова командой `ssh deploy@your-server-address` — если вы все сделали правильно, то сервер должен пустить вас не спросив пароль.

Пользователь `deploy` необходим нам для того, чтобы ваш сервер был в относительной безопасности. У этого пользователя будет доступ на запись и изменение файлов только к папке, в которой будет находиться ваше приложение. Если злоумышленник каким-либо образом хакнет ваше приложение, то ущерб, который он сможет нанести серверу, будет ограничен теми правами, которые есть у пользователя `deploy`.

[1] Шелл — это специальная программа в UNIX-системах, которая принимает команды от пользователя. Самым популярным шеллом является bash, но для новых пользователей по умолчанию выбирается шелл sh, который сильно ограничен в своем функционале. Если вы не совсем понимаете о чем идет речь, рекомендую вам обратиться к статье Командная оболочка UNIX, Wikipedia (на русском)

[1] Статья про RSA в Википедии (на русском)

[2] Создание RSA ключей

[3] Командная оболочка UNIX, Wikipedia (на русском)

RVM systemwide

Чтобы на вашем сервере нормально работал Ruby последней стабильной версии, нам необходимо поставить RVM — Ruby Version Manager [1]. Это специльный софт, который за вас скачает, скомпилирует и установит Ruby в систему.

В нашем случае, нам нужно установить RVM для всех пользователей (на локальной машине он, скорее всего, установлен у вас только для одного пользователя — вас). Но прежде чем мы сможем это сделать, нам необходимо временно дать права администратора нашему пользователю `deploy` — для этого находясь на сервере под пользователем `root` выполните следующую команду:

```
visudo
```

У вас откроется стандарный текстовый редактор (nano или vi) и в нем — файл `/etc/sudoers`. Найдите там строку `root ALL=(ALL:ALL) ALL` и добавьте после нее следующую строку:

```
deploy ALL=(ALL:ALL) ALL
```

Затем сохраните и закройте файл.[1] После этого подключитесь к серверу пользователем `deploy` как мы делали в предыдущем разделе.

Чтобы начать установку RVM, вам необходимо выполнить следующую команду:

```
\curl -L https://get.rvm.io | sudo bash -s stable
```

Вам потребуется ввести пароль для пользователя `deploy`, который вы придумали в прошлом разделе. После этого RVM начнет скачивать необходимые файлы и устанавливать их. Когда RVM закончит установку, вы получите сообщение об успешном завершении процесса и дальнейшую инструкцию, состоящую из двух пунктов:

1. Необходимо добавить пользователя `deploy` в группу `rvm` — команда `usermod -a -G rvm deploy`
2. Выполнить команду `source /etc/profile.d/rvm.sh`, чтобы загрузить RVM в текущую сессию.

К этим двум шагам нам потребуется добавить еще и третий: добавьте строку `PATH=$PATH:/usr/local/rvm/bin` в самый конец файла `/root/.bashrc` (после этого перезайдите на сервер или наберите команду `source /root/.bashrc`).[2]

И так, RVM установлен в систему и теперь вы должны получить возможность выполнить

следующие команды для установки Ruby версии 1.9.3 (наиболее популярная версия Ruby на сегодняшний день):

```
deploy$ sudo apt-get install build-essential # установка компилятора
deploy$ sudo apt-get install libtool         # установка зависимостей
deploy$ rvm pkg install zlib    # установка zlib (нужна для RubyGems)
deploy$ rvm pkg install openssl # установка библиотеки openssl

# установка ruby
deploy$ rvm install 1.9.3 --with-openssl-dir=/usr/local/rvm/usr
```

Проверить успешность установки можно командой `deploy$ ruby -v` — если она вернула вам версию Ruby, то значит все в порядке. На последок не забудьте убрать пользователя `deploy` из файла `sudoers`, аналогично тому, как мы туда его добавили.

Напоследок нужно вручную установить гем `Bundler`, который будет необходим сценарию деплоя:

```
deploy$ gem install bundler
```

[1] В текстовом редакторе nano — команда `Ctrl+X` и Yes при ответе на вопрос сохранить ли файл; в текстовом редакторе vi — команда `:wq`.

[2] Это нам обязательно потребуется чуть позже, чтобы мы смогли установить nginx и passenger.


[1] Официальный сайт RVM


[2] Документация по установке RVM


[3] Статья о команде sudo, Wikipedia (на русском)

Установка и настройка БД

Для вашего Rails приложения вам потребуется установить одну из СУБД, которую поддерживает ActiveRecord. В этом разделе мы установим и настроим MySQL, т.к. эта СУБД является самым распространенным выбором для приложений Rails.

Установка MySQL невероятно проста. Все что вам необходимо сделать — это, находясь под пользователем `root`, выполнить следующие команды:

```
apt-get install mysql-server
apt-get install libmysqlclient-dev ruby-dev
```

MySQL будет скачан и установлен. Во время установки вас могут попросить придумать пароль для пользователя `root` в MySQL — придумайте сложный пароль и запишите его где-нибудь.

После установки, нам еще нужно добавить MySQL в "автозагрузку", чтобы он

автоматически запускался при старте системы. Для этого выполните простую команду `update-rc.d mysql defaults`.

Теперь нам остается создать саму базу данных, в которой будут находиться таблицы для нашего приложения, а также специального пользователя[1], для подключения к ней. Чтобы сделать это, вам необходимо сначала запустить клиент `mysql`. Просто наберите в терминале команду `mysql -p`, введите пароль — программа подключится к уже запущенному MySQL-серверу и позволит вам выполнять команды на языке SQL.

В первую очередь создадим базу данных, которую будет использовать ваше Rails приложение. Обычно БД называют по имени приложения + той среды (environment), в которой оно будет запущено. Вот какую команду вам нужно выполнить:

```
CREATE DATABASE store_app_production;
```

Затем создадим пользователя `production` (так я предпочитаю называть пользователя БД, который предназначен для подключения Rails приложения):

```
CREATE USER production IDENTIFIED BY 'my_super_secure_password';
```

Не забудьте записать пароль — он нам понадобится. Наконец, разрешим пользователю выполнять любые операции[2] над созданной нами БД:

```
GRANT ALL ON store_app_production.* TO 'production'@'localhost';
```

Теперь вам остается только набрать команду `exit` и выйти из консоли mysql.

[1] Под пользователем имеется в виду не пользователь в вашей системе, а именно пользователь MySQL.

[2] На самом деле — это не идеальное решение, которое может подвергуть сохранность вашей БД опасности. Я предпочитаю создавать двух пользователей — `production` и `migration`, давая первому ограниченные права. Это нужно затем, чтобы удалять или создавать таблицы целиком можно было только при деплое приложения, используя среду `migration`.

Установка и настройка Nginx и Passenger

Последнее из того что нам нужно установить — это сервер `nginx` и специальный модуль `Passenger` к нему, который умеет запускать Rails-приложения. Процесс установки автоматизирован: специльный гем `passenger` скачает и установит за вас сервер nginx. Таким образом, установка сводится к двум командам (запускать под пользователем `root`):

```
gem install passenger
passenger-install-nginx-module
```

Перед тем как начать установку, программа проверит установлены ли в вашей ОС все

необходимые зависимости и выделит красным цветом те, которые вам потребуется доустановить. У меня в новой Ubuntu Server красным были выделены библиотеки `OpenSSL` и `Curl`. Нажав `Enter`, можно получить информацию о том, как их установить. В моем случае, мне пришлось выполнить следующие команды:

```
apt-get install libopenssl-ruby
apt-get install libcurl4-openssl-dev
apt-get install libssl-dev
```

После этого, можно заново попытаться запустить установщик Passenger командой `passenger-install-nginx-module`. Вам будет предложено скачать и установить `nginx` автоматически (1) или кастомизировать установку (2) — выбирайте пункт (1). Через некоторое время вас спросят, куда вы хотите установить `nginx`, предложив в качестве папки по умолчанию `/opt/nginx` — соглашайтесь и нажимайте `Enter`.

Наконец, после того как nginx установлен, нам будет необходимо сделать так, чтобы он запускался самостоятельно при старте системы — для этого нужно сделать из него сервис. **Шаг 1:** создайте файл `/etc/init.d/nginx` и откройте его в текстовом редакторе (например в nano):

```
nano /etc/init.d/nginx
```

Скопируйте и вставьте в этот файл следующий код:

```
#! /bin/sh

### BEGIN INIT INFO
# Provides:          nginx
# Required-Start:    $all
# Required-Stop:     $all
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: starts the nginx web server
# Description:       starts nginx using start-stop-daemon
### END INIT INFO

PATH=/opt/nginx/sbin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

DAEMON=/opt/nginx/sbin/nginx
NAME=nginx
DESC=nginx
RUN_AS="deploy"

test -x $DAEMON || exit 0

# Include nginx defaults if available
if [ -f /etc/default/nginx ] ; then
        . /etc/default/nginx
fi

set -e
```

```
case "$1" in
  start)
        echo -n "Starting $DESC: "
        start-stop-daemon --start --quiet --pidfile /var/run/$NAME
        pid \
                --exec $DAEMON -- $DAEMON_OPTS
        echo "$NAME."
        ;;
  stop)
        echo -n "Stopping $DESC: "
        start-stop-daemon --stop --quiet --pidfile /var/run/$NAME
        pid \
                --exec $DAEMON
        echo "$NAME."
        ;;
  restart|force-reload)
        echo -n "Restarting $DESC: "
        start-stop-daemon --stop --quiet --pidfile \
                /var/run/$NAME.pid --exec $DAEMON
        sleep 1
        start-stop-daemon --start --quiet --pidfile \
                /var/run/$NAME.pid --exec $DAEMON -- $DAEMON_OPTS
        echo "$NAME."
        ;;
  reload)
      echo -n "Reloading $DESC configuration: "
      start-stop-daemon --stop --signal HUP --quiet --pidfile
      /var/run/$NAME.pid \
          --exec $DAEMON
      echo "$NAME."
      ;;
  *)
        N=/etc/init.d/$NAME
        echo "Usage: $N {start|stop|restart|reload|force-reload}"
        >2
        exit 1
        ;;
esac

exit 0
```

Теперь сохраните файл, закройте его и выполните команду `chmod +x /etc/init.d/nginx`, сделав таким образом этот файл *исполняемым*.

После этого nginx должен запускаться командой:

```
service nginx start
```

и останавливаться командой

```
service nginx stop
```

Шаг 2: добавьте этот скрипт в "автозагрузку". Сделать это можно следующей командой:

```
update-rc.d nginx defaults
```

Теперь при каждом запуске системы nginx будет стартовать автоматически, а если вам потребуется перезагрузить его вручную, вы всегда можете воспользоваться командой `service nginx restart`.

[1] Подробнее о том, что такое сервисы в ОС Ubuntu можно почитать в документации к команде update-rc.d [2] и вот в этом вопросе на AskUbuntu [3]. В этом же разделе я просто опишу ряд несложных шагов, которые позволят вам добиться желаемого результата.


[1] Официальный сайт Phusion Passenger с короткой инструкцией по установке.


[2] Документация к команде update-rc.d


[3] What's the recommended way to enable / disable services?

Деплой приложения

Под *деплоем* приложения обычно имеют в виду запуск новой версии этого приложения на сервере. Этот процесс включает в себя копирование, собственно, исходного кода (который находится в репозитории git), компилирование необходимых файлов (в случае Rails-приложения — ассетов), обновление базы данных, выполнение различных административных задач (например создание файлов или симлинков необходимых для работы приложения) и, наконец, перезапуск самого приложения. Все это мы рассмотрим далее.

К счастью, все описанные задачи — легко автоматизируются, так что весь деплой сведется к запуску вами всего лишь одной команды. Но, перед тем как мы сможем это сделать, необходимо подготовить всю инфраструктуру на сервере. Ранее в этой главе мы установили весь необходимый софт и настроили его. Теперь пора создать место, где будет находиться само Rails-приложение.

Что нужно сделать перед первым деплоем

Минимальный набор действий, который нам необходимо сделать, чтобы подготовить почву для запуска Rails приложения на сервере, состоит из следующих пунктов:

1. Создание git-репозитория для хранения кода
2. Создание структуры папок, где будет находиться приложение
3. Обновление конфигурации nginx

Рассмотрим каждый из них подробнее.

Создание git-репозитория

Ваше Rails-приложение на локальной машине лежит в git-репозитории, который мы создали для него (если это не так — перечитайте раздел Что следует сделать сразу после создания? главы Создание Rails-приложения). Наша задача сейчас — сделать копию этого репозитория на сервере. Копия репозитория на сервере понадобится нам для двух вещей: 1) именно из репо на сервере будет воссоздаваться новая версия приложения каждый раз, когда мы будем его деплоить и 2) репозиторий на сервере — удобный способ синхронизироваться с другим разработчиками (если над проектом работает несколько людей).

Первым делом вам следует зайти на сервер под пользователем `root` и установить в систему git[1]:

```
apt-get install git-core
```

После того, как git установится, выполните следующие команды:

```
# создаем папку, где будут лежать все репозитории
mkdir /var/repos

# создаем папку, где будет лежать репозиторий приложения
mkdir /var/repos/store_app.git

# создаем репозиторий (без дерева папок)
cd /var/repos/store_app.git
git --bare init

# Назначаем владельцем всех репозиториев пользователя deploy
chown -R deploy /var/repos
```

Теперь вернитесь **в терминал вашей локальной машины**, перейдите в папку с проектом и выполните следующие команды:

```
# Назначаем главным удаленным репозиторием тот,
# который мы только что создали на сервере
git remote add origin \
deploy@your-server-address:/var/repos/store_app.git

# Заливаем все содержимое в удаленный репозиторий
git push origin master
```

Далее, каждый раз перед тем, как задеплоить приложение (или чаще, если вы работаете с другими разработчиками в команде) вам будет нужно выполнять команду `git push`, которая зальет в удаленный репо новые коммиты, сделанные вами локально.

Создание структуры папок, где будет находиться приложение

Назначение папок, которые мы будем создавать ниже, может быть не сразу будет понятно вам до конца, хотя я попытаюсь объяснить это максимально четко. Ситуация прояснится, когда мы начнем деплоить проект с помощью `Capistrano`, а пока что просто выполните указанные действия и попытайтесь уловить основной смысл того, о чем я буду говорить.

Прежде всего, нам понадобится создать папку, где будут храниться все ваши проекты (потому что на одном сервере можно запустить несколько Rails-приложений!). Обычно ее создают в директории `/var` и называют `www`:

```
mkdir /var/www
chown deploy /var/www
```

Второй командой мы сделали пользователя `deploy` владельцем этой папки — это важно, т.к. наше приложение будет запущено именно этим пользователем и отсутствие у него необходимых привелегий повлечет за собой ошибку.

Теперь вам нужно залогиниться под пользователем `deploy`, перейти в папку `/var/www` и выполнить следующие команды по созданию внутренних папок, которые понадобятся для проекта:

```
deploy$ cd /var/www
deploy$ mkdir store_app
deploy$ mkdir store_app/production
deploy$ mkdir srore_app/production/shared
deploy$ mkdir srore_app/production/shared/config
deploy$ mkdir srore_app/production/shared/public
deploy$ mkdir srore_app/production/releases
```

Разберемся со всем по порядку:

- `/var/www/store_app` — основная папка приложения.
- `production` — папка для продакшен-версии приложения. Часто на сервере еще может быть staging-версия. Назвать эти папки можно было бы как угодно, но обычно их называют именно таким образом — `production` и `staging`.
- `production/shared` — папка, в которой будут находиться общие для всех релизов приложения файлы. Например файл с настройками для доступа к БД или файлы, которые загрузили пользователи сайта.
- `production/shared/config` и `production/shared/public` — просто внутренние папки, созданные для того, чтобы структура папки `shared` напоминала структуру папок Rails-приложения.

- `production/releases` — папка, которая будет содержать релизы приложения. Каждый раз, когда вы будете деплоить приложение с помощью `Capistrano`, внутри этой папки будет появляться еще одна вложенная папка, содержащая все файлы вашего Rails-приложения, которые были добавлены в git-репозиторий. Можно считать, что каждый такой релиз будет снимком одного текущего состояния (или лучше сказать коммита) вашего репозитория.

Вот и вся структура папок, которую вам потребуется создать вручную.

Обновление конфигурации nginx

Чтобы приложение реально заработало, нужно объяснить серверу `nginx` где оно находится и *чем* его нужно запускать (а запускать нужно модулем `Passenger`). Поэтому нам понадобится немного подредактировать файл `/opt/nginx/conf/nginx.conf`. Создайте резервную копию файла командой `cp /opt/nginx/conf/nginx.conf /opt/nginx/conf/nginx.conf.backup`, затем откройте основной файл, удалите все, что там есть и скопируйте туда следующее:

```
user deploy;
worker_processes 1;

error_log /var/log/nginx/error.log error;
pid       /var/run/nginx.pid;

events {
  worker_connections  1024;
}


http {
  client_max_body_size 20M;
  server_names_hash_bucket_size 64;

  passenger_root /usr/local/rvm/gems/ruby-1.9.3-p374/gems/passenger-3
  0.19;
  passenger_ruby /usr/local/rvm/wrappers/ruby-1.9.3-p374/ruby;

  passenger_max_pool_size 12;
  passenger_default_user deploy;

  include       mime.types;
  default_type  application/octet-stream;

  gzip      on;
  sendfile  on;

  keepalive_timeout  65;

  server {
```

```
    listen 80;
    server_name storeapp.com www.storeapp.com;
    root /var/www/store_app/production/current/public;
    passenger_enabled on;
    rails_env production;

    location ~ ^/(assets|images|javascripts|stylesheets|swfs|system)/
    {
      gzip_static on;
      expires     max;
      add_header  Cache-Control public;
    }

    error_page  404  /404.html;
    error_page  500 502 503 504  /500.html;

    client_max_body_size 20M;

    if (-f $document_root/maintenance.html) {
      rewrite  ^(.*)$  /maintenance.html break;
    }
  }
}
```

Теперь нам нужно не забыть заменить некоторые части приведенного кода, чтобы это заработало для вашего приложения.

Во-первых убедитесь, что строки

```
passenger_root /usr/local/rvm/gems/ruby-1.9.3-p374/gems/passenger-3
0.19;
passenger_ruby /usr/local/rvm/wrappers/ruby-1.9.3-p374/ruby;
```

соотвтетствуют тому, что было в вашем резервном файле (они там тоже есть). Дело в том, что установленная версия Passenger-а и Ruby у вас может отличаться, поэтому если вы просто вставите мой вариант — может так случиться, что ничего не заработает.

Во-вторых, обратите внимание на опцию `passenger_max_pool_size 12;` — она имеет прямое отношение к количеству оперативной памяти на вашем сервере и тому, сколько памяти занимает один экземпляр вашего приложения — чем больше памяти, тем больше значение. Чтобы понять, какое значение установить этой опции, ознакомьтесь с соответствующим разделом документации по Passenger-у [1].

В третьих, установите правильное доменное имя для вашего приложения в опции `server_name` — и не забудьте указать второе доменное имя с префиксом `www`!

Ну и последнее: в директиве `root /var/www/store_app/production/current/public;` замените название папки `store_app` на название той папки, где находится ваше приложение. Остальные части пути оставьте неизменными.

Теперь `nginx` готов запустить ваше приложение, но пока что мы еще не задеплоили приложение на сервер, поэтому запускать ему будет нечего. Мы исправим это недоразумение в следующем разделе.

Как я уже упоминал выше, три этих шага — это минимум того, что вам придется сделать. В этой главе мы не успели рассмотреть следующие важные вещи, которые вам придется со временем настроить самостоятельно:

- Бэкап — регулярное создание резервных копий файлов и БД
- Cron — автоматическое выполнение задач по расписанию (понадобится для регулярных бэкапов)
- LogRotate — архивирование Log-файлов
- Настройка Sendmail для отправки почты из Rails-приложений (например при помощи Postfix)
- Мониторинг работспособности сервера (например с помощью сервиса Pingdom)
- Управление разделами (partitions) на сервере.

[1] Естественно, пакет git не стоит в Ubuntu Server по умолчанию.

[2] О том, что такое staging-версия, читайте в разделе Staging или тестовая версия приложения.

[1] Описание опции passenger_max_pool_size в документации по Passenger-у

Автоматизация деплоя при помощи Capistrano

Capistrano — это специальный гем, который деплоит приложение на сервер в соответствии с определенным сценарием. В этом разделе мы создадим такой скрипт, чтобы в дальнейшем вы могли деплоить приложение одной командой из вашего терминала: `cap deploy`.

В целом деплой новой версии приложения на сервер происходит следующим образом:

1. Вы делаете `git push`
2. Затем запускаете команду `cap deploy`, которая
3. подключается к вашему серверу
4. создает новую папку релиза внутри `/var/www/store_app/production/releases`
5. создает симлинк [2] `/var/www/store_app/production/current`, который ссылается на папку с текущим релизом
6. копирует туда структуру всех папок и файлов вашего приложения в соответствии с последним коммитом

7. выполняет разные административные задачи, такие как миграция БД или создание симлинков
8. перезупскает приложение

Установка гема

Чтобы установить гем, нужно сделать две вещи. Во-первых добавьте в файл `Gemfile` в группу `development` следующие строки:

```
gem 'capistrano'
gem 'capistrano_colors'
gem 'rvm-capistrano'
```

и запустите `bundle install`. Во-вторых, выполните команду:

```
capify .
```

Находясь в домашней директории проекта. Команда должна показать следующий output:

```
[add] writing './Capfile'
[add] writing './config/deploy.rb'
[done] capified!
```

Если вы увидели эти строки, то все в порядке.

Написание сценария деплоя

Самая важная задача — это написать сценарий, по которому будет происходить деплой нашего приложения. Сценарий находится в файле `config/deploy.rb`. Этот файл вам и нужно открыть для редактирования. Вы можете смело удалять все, что там есть, т.к. я покажу вам готовую версию сценария для нашего приложения store_app. Чтобы понять, как писать сценарии деплоя для Capistrano, пожалуйста прочтите документацию и разберитесь в основах Capistrano [1] — только после этого продолжайте читать эту главу. Я прокомментирую кастомные таски и опции из сценария ниже, но без понимания как работают сценарии, вы не сможете разобраться в этом коде. И так, вот какой сценарий должен быть в файле `config/deploy.rb`

```
require "rvm/capistrano"
require "bundler/capistrano"
require 'capistrano_colors'
load    'deploy/assets'

set :rvm_type, :system
set :application, "store_app"
```

```
set :repository,  "/var/repos/store_app.git"
set :local_repository,
    "ssh://deploy@your-server-address/var/repos/store_app.git"
set :branch, "master"
set :user,   "deploy"
set :deploy_via, :remote_cache
set :keep_releases, 3
set :scm, :git
set :use_sudo, false

set :deploy_to, "/var/www/#{application}/production"
set :rails_env, "production"
set :branch, "master"

role :web, "your-server-address"
role :app, "your-server-address"
role :db,  "your-server-address", :primary => true

namespace :deploy do

  namespace :custom_symlinks do
    task :custom_configs do
      run "ln -nsf #{shared_path}/config/database.yml #{current_release}/config/"

    end
    task :user_files do
      run "ln -nsf #{shared_path}/public/uploads/ #{current_release}/public/"

    end
    task :default do
      user_files
      custom_configs
    end
  end


  task :start do ; end
  task :stop do ; end
  task :restart, :roles => :app,
       :except => { :no_release => true } do
          run "#{try_sudo} touch #{File.join(current_path,'tmp','restart
          txt')}"
       end

end

namespace :migrate do

  task :default do
    run "cd #{current_release}  bundle exec rake db:migrate"
  end

end

before "deploy:assets:precompile", "deploy:custom_symlinks"
after  "deploy:update", "deploy:cleanup"
```

Из приведенного выше сценария я хотел бы отметить два таска: `:custom_symlinks` и `:user_files`. Оба создают т.н. *симлинки* [2] (можно думать о них, как о ярлыках в Windows).

Если вы уже прочли документацию по Capistrano, то вы знаете, что когда приложение деплоится — для него создается отдельная папка внутри директории `/var/www/store_app/production/releases`, для которой затем создается новый симлинк `current`. Если в вашем приложении пользователи загружают какие-то файлы (например аватарки), которые вы хотите сделать доступными для просмотра на сайте, вам необходимо будет загружать их в папку `public/` внутри приложения. Однако каждый раз во время нового деплоя будет создаваться новая папка с релизом и новый симлинк `current` для нее. Таким образом, все загруженные пользователями файлы окажутся в папке `public/` старого релиза и будут недоступны. Чтобы избежать этой ситуации, мы сделаем так, что папка, в которую пользователи будут загружать файлы — пусть это будет `public/uploads` — на самом деле была бы симлинком на директорию `/var/www/store_app/production/shared/public/uploads` и попросим Capistrano (в таске `:user_files`) создавать этот симлинк каждый раз, когда вы деплоите приложение. В этом случае, файлы всегда будут физически находиться в папке `/var/www/store_app/production/shared/public/uploads`, в то время как в папке текущего релиза будет существовать симлинк. Это позволит каждому релизу "видеть" одни и те же файлы!

Похожая история и с таском `:custom_symlinks`. Дело в том, что в вашем репозитории не должно быть файла `config/database.yml`[1] — его нужно добавить в `.gitignore`. То есть, на вашей локальной машине файл существовать будет, но в репозитории он отсутствует. Делается это в целях безопасности: файл, содержащий пароль для доступа к серверной БД должен находиться только на сервере! Чтобы все работало как мы хотим, вам потребуется создать файл `database.yml` в папке `/var/www/store_app/production/shared/config` на сервере и записать в него следующее:

```
production:
  adapter: mysql2
  encoding: utf8
  database: store_app_production
  username: production
  password: [your password]
  host: 127.0.0.1 # ip-адрес должен быть именно такой!
  port: 3306
```

Таск `:custom_symlinks` в свою очередь, каждый раз при выполнении нового деплоя будет создавать симлинк на этот файл в папке `/var/www/store_app/production/current/config`.

Напоследок, перед тем как наконец запустить наш первый деплой, убедитесь, что в вашем `Gemfile` нет гемов, которые не нужны на продакшене. Например, вам на продакшене не нужен гем `sqlite3`, а вместо него нужен гем `mysql2`. Также в среде `production` нам обязательно понадобится гем `therubyracer`, который поможет сжимать Javascript. Таким образом, я бы отредактировал эту часть `Gemfile`-а следующим образом:

```
group :development do
  gem 'sqlite3'
end

group :production do
  gem 'mysql2'
  gem 'therubyracer'
end
```

Обратите внимание, что гем `coffee-rails`, который есть в `Gemfile` по умолчанию, требует установленных на сервере `Node.js` и `Coffeescript`. Если вам не нужен этот гем и вы пока не хотите утруждать себя установкой всего этого софта на сервер — просто закомментируйте его в `Gemfile`. **И не забудьте после внесения этих изменений запустить `bundle install`[2]!**

Наконец, мы готовы сделать наш первый деплой. Для этого:

1. Закомитьте изменения в git-репозиторий.
2. Сделайте `git push`, чтобы залить изменения в удаленный репо на сервере.
3. Выполните команду `cap deploy`, чтобы задеплоить приложение.
4. Выполните команду `cap deploy:migrations`, вместо предыдущей команды, в случае если у вас есть невыполненные на сервере миграции (при первом деплое это скорее всего будет именно так).

Если во время этого процесса вы не увидели никаких сообщений, выделенных красным цветом — значит все в порядке. Запустите nginx командой `service nginx start` (находясь под root-ом) и ваше приложение должно стать доступным по адресу вашего сервера через браузер.

Для работы над этой главой я создал полностью пустой сервер и выполнил все описанное по ходу ее написания. Поэтому мои условия должны были быть маскимально приблежены к "боевым" и вы не должны были увидеть никаких ошибок. Если же у вас все-таки не получилось задеплоить приложение, пожалуйста сделайте две вещи: 1) внимательно прочтите все сообщения об ошибках и попытайтесь самостоятельно понять, в чем могло быть дело 2) обратитесь к главе Что дальше: как разбираться в проблемах и кому задавать вопросы?.

[1] Убедиться, что его там нет и удалить его из репозитория (но не из файловой системы) можно командой `git rm --cached config/database.yml`

[2] Для успешного выполнения команды `bundle install` вам, скорее всего, понадобится поставить mysql-server и на локальную машину. Если ваша локальная машина — это Ubuntu Desktop, то установка СУБД MySQL будет идентична описанной в разделе Установка и настройка БД.

[1] *Wiki по Capistrano со ссылками на документацию и статьи*

[2] *Символьные ссылки, Wikipedia (на русском)*

Staging или тестовая версия приложения

Иногда те или иные баги становятся заметными только на продакшене. Это объясняется в первую очередь тем, что сама среда (environment) на продакшене и на локальной машине часто отличается — набором гемов, настройками и т.д. Если после деплоя проекта у вас вдруг обнаруживается критическая ошибка, пользователи вашего сайта будут не очень довольны — особенно если они не просто пользователи, а клиенты, которые платят вам деньги. Минимизировать риск появляения таких ошибок можно, если иметь на сервере — на специальном поддомене — еще одну копию приложения, с идентичными настройками среды, но с другой базой данных[1]. Такое приложение (и соответствующую ему среду) называют *staging*[2]-версией. В этом разделе я покажу как сделать такую staging-версию.

Staging-версия будет жить в директории `/var/www/store_app/staging` на сервере; соответственно, внутри вам следует создать всю структуру папок, которую мы делали для production-версии в предыдущем разделе.

Далее вам потребуется создать базу данных `store_app_staging`. Для этого подключитесь к MySQL-серверу с помощью команды `mysql -p` и наберите уже знакомые вам команды:

```
CREATE DATABASE store_app_staging;
CREATE USER staging IDENTIFIED BY 'my_super_secure_password';
GRANT ALL ON store_app_staging.* TO 'staging'@'localhost';
```

Выйдите из клиента MySQL командой `exit`. После того как база создана, надо не забыть добавить сведения о подключении к ней в файл `/var/www/store_app/staging/shared/config/database.yml`:

```
staging:
  adapter: mysql2
  encoding: utf8
  database: store_app_staging
  username: staging
  password: [your password]
  host: 127.0.0.1
  port: 3306
```

Вам также следует добавить внутрь блока `http {}` в файле `/opt/nginx/conf/nginx.conf` информацию о staging-приложении:

```
server {
  listen 80;
```

```
  server_name staging.storeapp.com staging.storeapp.com;
  root /var/www/store_app/staging/current/public;
  passenger_enabled on;
  rails_env staging;

  location ~ ^/(assets|images|javascripts|stylesheets|swfs|system)/ {
    gzip_static on;
    expires     max;
    add_header  Cache-Control public;
  }

  error_page  404  /404.html;
  error_page  500 502 503 504  /500.html;

  client_max_body_size 20M;

  if (-f $document_root/maintenance.html) {
    rewrite  ^(.*)$  /maintenance.html break;
  }
}
```

Теперь вернемся на локальную машину. Прежде всего вам потребуется создать новую среду — staging — и для этого вам достаточно скопировать файл `config/environments/production.rb` и сделать из него файл `config/environments/staging.rb`. Никаких изменений в него вносить не надо! Вам также нужно добавить ко всем группам `:production` в `Gemfile` группу `:staging`, вот так:

```
group :production, :staging do
  gem 'mysql2'
  gem 'therubyracer'
end
```

и запустить команду `bundle install`.

И последнее, что вам потребуется сделать — немного изменить сценарий деплоя в файле `config/deploy.rb` на локальной машине. Вместо вот этого кода:

```
set :deploy_to, "/var/www/#{application}/production"
set :rails_env, "production"
set :branch, "master"
```

там должен быть следующий код:

```
task :production do
  set :deploy_to, "/var/www/#{application}/production"
  set :rails_env, "production"
  set :branch, "master"
end

task :staging do
  set :deploy_to, "/var/www/#{application}/staging"
  set :rails_env, "staging"
  set :branch, "master"
end
```

```
# не забудьте добавить эту строку,
# она по умолчанию запускает деплой для staging
staging
```

Теперь деплой на продакшен вы должны будете запускать командой `cap production deploy`, в то время как деплой на staging будет происходить по умолчанию при запуске команды `cap deploy`. Не забудьте сделать коммит и `git push`. После перезапуска сервера `nginx` (`service nginx restart`) и выполнения `cap deploy` ваша staging-версия приложения должна заработать.

[1] Имеется в виду не другая СУБД, а просто другая база внутри той же самой СУБД. Например если продакшен база называется `store_app_production`, то БД для тестового приложения можно назвать `store_app_staging`.

[2] От слова "stage" — сцена.

Чеклист перед каждым деплоем

Чтобы минимизировать возможные ошибки на продакшене, я предлагаю вам пользоваться этим небольшим чеклистом каждый раз, когда вы собираетесь делать новый релиз вашего приложения (то есть деплоить его).

Перед каждым новым деплоем вы должны убедиться, что:

- Все тесты успешно проходят (`rspec spec`).
- Вы сделали коммит последних изменений в репозиторий.
- У вас нет *untracked* файлов в git-репозитории (`git status`).
- Последний коммит был залит в удаленный репозиторий на сервере (`git push`).
- У вас нет новых миграций — если они есть, то не забудьте деплоить с помощью `cap deploy:migrations`, вместо обычного `cap deploy`.
- Проект задеплоен на staging-сервер и работает там успешно (проверить вручную через браузер).

Потенциальные проблемы на сервере

[1] Книга "Запускаем Linux" (на русском)

[2] Книга "Командная строка Linux и сценарии оболочки" (на русском)

[3] Скринкасты "Научись Linux" (на русском)

[4] Книга "The Linux Command Line: A Complete Introduction" (на английском)

Глава 11. Что дальше: как разбираться в проблемах и кому задавать вопросы?

Одна из основных проблем, которая мешает новичкам быстро превращаться в грамотных разработчиков — отсутствие навыка самостоятельно находить нужную им информацию и правильно задавать вопросы. В этой короткой главе я расскажу о том, как начать ориентироваться в мире Ruby On Rails и искать ответы на возникающие у вас вопросы. А вопросы обязательно появятся. Никогда не рассчитывайте на тот факт, что ознакомившись с одним источником (книгой или скринкастами) вы сможете освоить материал.

Поиск ответов

Когда у вас возникает какая-то проблема при разработке приложения и вы не знаете как ее решить — у вас, по сути, есть два варианта: попытаться найти информацию о проблеме самостоятельно или спросить кого-нибудь. Первый вариант должен всегда предшествовать второму. Если вы не потратили на самостоятельный поиск решения хотя бы час вашего времени — не стоит задавать вопрос другому человеку. Это прежде всего неуважение к чужому времени, потому что вы как бы просите человека решить за вас проблему, на которую вы лично не хотите тратить свое время — причем просите решить бесплатно. Don't be one of those guys! Ну и, кроме того, вы лишаете себя тренировки навыка самостоятельного поиска ответов: чем больше ответов вы пытаетесь найти самостоятельно, тем быстрее и лучше у вас это будет получаться.

Ниже я попытался описать способы, которыми можно найти ответ самостоятельно.

Внимательное чтение ошибок

Пожалуй, самое распространенное поведение новичков — нежелание читать сообщения об ошибках, которые выдает им Ruby, Rails или Rspec. Новички пугаются сообщений об ошибках и ожидают, что программа должна просто работать. Сообщения об ошибках, однако, это важный фидбек. В них содержится много полезной информации, например номер строки и название файла, где произошла ошибка — вы не поверите, как много людей задают вопросы "почему это не работает?" не потрудившись сначала посмотреть в собственный код — в то место, которое указано в сообщении об ошибке!

Если вы столкнулись с ошибкой — прочитайте его целиком и попытайтесь понять, что оно означает. Со временем, вы научитесь быстро отделять важные части сообщений об ошибках от второстепенных и не имеющих значения — и ошибка уже не будет казаться такой страшной.

Эксперименты с кодом, debugging

Часто, после прочтения сообщения об ошибке, бывает полезно попробовать поизменять различным образом тот код, рядом с которым ошибка произошла, чтобы понять как на самом деле ведет себя программа в разных ситуациях. Это помогает докопаться до сути самой ошибки. Такой процесс в английском языке называется *debugging*, а по русски иногда говорят о том, что кто-то "дебажит" код. Процесс дебаггинга включает в себя вывод значений переменных в том или ином месте (самый примитивный способ это сделать — с помощью `puts`), замена одних значений на другие и т.д. Дебаггингу посвящено много книг и статей и если вам станет интересно — я привел несколько источников в списке литературы к этому разделу.

Ruby/Rails API и доки для гемов

Если ваша проблема заключается в том, что вы не знаете как пользоваться каким-то стандартным методом Ruby, Rails или какого-нибудь Ruby-гема, метод работает не так как вы ожидаете или же вы не знаете каким бы методом воспользоваться для решения вашей задачи — вам следует смотреть в официальную документацию (в списке литературы приведены ссылки на документацию по Ruby и Ruby On Rails). Такую документацию еще называют API, поэтому когда кто-то говорит про "Ruby API" или "Rails API" — он скорее всего имеет в виду именно документацию.

Документация к Ruby библиотекам (в т.ч. к Rails) обычно генерируется специальным гемом RDoc и поэтому сайты с документацией к разным гемам могут выглядеть практически одинаково. В любом случае, чтобы успешно пользоваться документацией, вам следует запомнить следующее: она состоит из ссылок на описание классов и методов + в ней обычно есть поиск по классам и методам, поэтому всегда достаточно легко найти то, что вас интересует.

Например, если я хочу узнать о том, как работает метод `#link_to` в Rails, я просто иду на сайт Rails API [5] и ввожу название метода в верхнем левом углу. Иногда метод с одинаковым названием может присутствовать в нескольких разных классах — в этом случае вам просто нужно выбрать тот класс, который вас интересует. Или, если вы не знаете, с каким классом вы работаете, попробуйте пройтись по каждому и понять, что именно вам

нужно.

Если вы не знаете, какой метод вам следует использовать для решения задачи, возможно вам стоит попробовать просмотреть все доступные в интересующем вас классе методы, и по названию метода сделать предположение, подходит ли он вам. Затем вы можете прочитать документацию по этому методу и потдвердить или опровергнуть свое предположение. Например, если мне нужно инвертировать порядок элементов массива (`Array`), то я зайду в документацию класса `Array` в Ruby API, просмотрю список всех методов, увижу там метод `#reverse` и сделаю предположение, что возможно это именно то, что мне нужно. После прочтения документации к этому методу, мне станет понятно, что `#reverse` — именно то, что я искал, потому что он превратит массив `["a", "b", "c"]` в `["c", "b", "a"]`.

Правильная формулировка вопроса в гугле

Наконец, если ни сообщение об ошибке, ни документация вам не помогли, возможно имеет смысл попробовать погуглить ответ. Самый простой способ сделать это правильно — найти самую важную часть сообщения об ошибке, добавить к ней несколько слов контекста (обычно название гема или библиотеки, в которой — или *из-за* которой — произошла ошибка) и вставить это в поиск гугла.

Например, при написании главы Настройка сервера и деплой приложения я столкнулся с ошибкой `Use RbConfig instead of obsolete and deprecated Config`, когда пытался установить один из Ruby-гемов. Подозревая, что дело может быть в не совсем верной установке RVM, я набрал в гугле следующий запрос: `rvm Use RbConfig instead of obsolete and deprecated Config` и сразу же в первых строках увидел вопросы с точно такой же ошибкой, которые кто-то уже задавал на StackOverflow.

Не вставляйте в поисковый запрос номера строк, имена файлов (если это файлы созданные вами) и название ваших переменных — это только усложнит вам нахождение страницы с потенциальным решением проблемы.

Если это не запрос содержащий сообщение об ошибке, старайтесь сделать его как можно короче. Если вам требуется, например, разобраться как позволить пользователям загружать файлы в Rails-приложении, плохой запрос выглядел бы следующим образом:

```
how to allow users to upload files in rails
```

в то время как хороший запрос выглядел бы так

```
rails uploading files
```

Наконец, не забудьте про сайт "The Ruby Toolbox" [6], который позволяет искать готовые решения по ключевым словам.

[1] Вопрос "How to debug ruby code?" на StackOverflow.

[2] Debugging Rails Applications, Rails Guides (на английском).

[3] How to Read Error Messages for Ruby on Rails.

[4] Документация по Ruby.

[5] Документация по Ruby On Rails.

[6] The Ruby Toolbox - поиск гемов по ключевым словам.

И так, вы исчерпали все возможности по самостоятельному поиску и ответа и готовы задать вопрос. Но как правильно это сделать, чтобы с большей долей вероятности получить правильный и быстрый ответ? В этом разделе я объясню несколько несложных правил, следуя которым вы сможете эффективно задавать вопросы, а также расскажу об основных местах в сети, где можно эти вопросы задать.

Как задавать вопросы?

На мой взгляд, вопросы которые мы задаем — они как рецепторы, которые принимают из внешней среды различные импульсы и затем передают их в мозг на обработку. Если вы используете неправильный рецептор для определенной цели — например вкусовой рецептор для восприятия света — импульс никогда не будет воспринят и мозг не получит никакой информации. Также и с вопросами: правильно поставленный вопрос — это уже значительная часть ответа. Хорошо продуманный и грамотно написанный вопрос не только способствует получению вами быстрого и точного ответа от других людей, но также помогает вам самим глубже обдумать проблему и, возможно, самостоятельно найти решение. Я очень часто слышал от своих учеников (и имел возможность убедиться в этом, когда задавал свои вопросы), что решения приходят во время написания вопроса.

Мастерство конструирования вопросов тоже приходит со временем. Однако то, как должен быть сконструирован вопрос, зависит не только от вас, но также и от того, где и для кого он пишется. В одном месте вы можете себе позволить написать длинный вопрос с большим количеством примеров, в другом — будет лучше, если вы для начала обойдетесь двумя предложениями. Ниже я приведу несколько общих правил, которые, по моему мнению, будут справедливы практически везде:

- Избегайте любезностей (например "Заранее благодарю за ответ" или "Приветствую вас, многоуважаемые товарищи программисты") в исходных вопросах — они не помогут вам заслужить уважение других участников дискуссии, зато вы потратите больше чужого времени, потому что людям

придется это читать. Сразу переходите к делу — если ваш вопрос уместен и построен грамотно — вы получите ответ.

- Не задавайте размытые вопросы вроде "Ничего не работает, не понимаю почему, помогите!!". Во-первых это не вопрос, а восклицание. Во-вторых такие вопросы никому не интересны, потому что они демонстрируют неспособность автора вести внятную дискуссию и думать самостоятельно.
- Не задавайте два вопроса одновременно — сосредоточьтесь на чем-то одном. Если вы не можете разбить ваш вопрос на несколько частей, почему вы ожидаете, что эту работу будет за вас проделывать кто-то еще?
- Старайтесь задать вопрос как можно более абстрактно — ни у кого нет времени разбираться в особенностях вашего приложения. Вы должны проделать умственную работу и выделить из вашей проблемы наиболее абстрактную часть.
- Соблюдайте форматирование, которое принято на том или ином ресурсе. Например на hasBrains люди иногда "забивают" на форматирование кода и пишут его в виде обычного текста. Стоит ли говорить, что читать такой вопрос становится гораздо менее приятно, а значит и вероятность получить ответ — снижается.
- Соблюдайте правила того ресурса, на котором вы задаете вопрос. Потрудитесь прочитать правила, прежде чем что-либо писать. Никому не нравится, когда посторонние люди приходят к вам в дом и начинают вести себя черт знает как. Поэтому, от вашего знания и соблюдения правил зависит вероятность получения вами ответа.
- Если там, где вы задаете вопрос, есть поле "Тема" (subject) или "Заголовок" (title) — продумайте, что поместить туда. Обычно заголовок должен вкратце отражать суть задаваемого вами вопроса, а само тело вопроса лишь раскрывать детали.
- Если вы задаете вопрос на сайте, который доступен другим людям (и, соответственно, ваш вопрос может в будущем помочь кому-то найти ответ) — подумайте о том, как сделать ваш вопрос максимально полезным тем, кто найдет его, например, через несколько месяцев. Этот пункт связан с пунктом об абстракции вопроса.
- Не делайте из вопроса "Войну и Мир". Текст вопроса должен быть максимально коротким (но не слишком!) — всегда думайте о том, что задавая вопрос, вы просите других людей потратить на его чтение время. И что у них есть возможность отказаться от этой сделки.
- Не торопитесь с публикацией вопроса — перечитайте его; убедитесь, что вы не

допустили грамматических, орфографических и пунктуационных ошибок; убедитесь, что в вашем коде нет ошибок.

- Прежде чем задать вопрос — убедитесь, что никто не спрашивал что-нибудь аналогичное на том же самом ресурсе. Обычно на сайтах с вопросами и форумах есть поиск. Или вы можете воспользоваться гуглом и ввести запрос наподобие `site:your-question-site.com How to ask questions?` (`site:` ограничит поиск указанным сайтом).
- Будьте вежливы. Если кто-то отвечает на ваш вопрос не так, как вам хотелось бы, и его ответ не решает вашу проблему — помните, что никто не был обязан отвечать на ваш вопрос; человек который это сделал — сделал это добровольно. Манеры, выдержка и вежливость еще никому не вредили.

Где задавать вопросы

Ниже я приведу список способов, которыми вы можете задать вопрос в сети. К каждому способу я добавил краткое описание, чтобы сориентировать вас по тому, какого типа вопросы там лучше задавать.

- **StackOverflow** начинался как сайт для программистов, на котором можно было задать любой вопрос по любому языку программирования или технологии с ним связанной, и быстро получить ответ. Со временем из этого сайта вырос StackExchange с большим количеством подсайтов посвященных не только программированию, но, например, администрированию UNIX-систем или Bitcoin. На StackOverflow можно задать любой вопрос (неважно, новичок вы или уже опытный программист), связанный с разработкой на Ruby и Ruby On Rails — при этом, безусловно, рекомендую вам следовать правилам, описанным выше.
- **Github issues** — это специальный раздел, который есть у каждого проекта на Github-е. Туда обычно постят багрепорты — сообщения об ошибках, которые вызваны именно этой библиотекой. Если вы абсолютно уверены, что дело не в вашем приложении, а в чужом Ruby-геме или библиотеке — попробуйте запостить тему в Github Issues. Вот, например, раздел Issues для проекта Ruby On Rails на Github.
- **Google Groups** многие программисты выбирают как удобное место для обсуждения их библиотеки. Часто в Google группу того или иного Ruby-гема можно написать предложения по улучшению или какой-нибудь специфический вопрос. Если у проекта есть Google Группа (обычно ссылка на нее указана в

документации), то лучше задать вопрос там, чем в Issues или в личном email-е автору.

- **IRC is not dead**. Вопреки распространенному мнению, чатики вовсе не пережиток 90-х. Люди активно пользуются IRC-каналами для профессионального общения, в т.ч. касающегося программирования — в том числе Ruby и Rails. На сервере freenode.net, например, есть каналы #ruby и #rubyonrails. Ответы, которые я там получал не раз мне помогали быстро решить насущную проблему. У хостинг-провайдера Linode.com также есть собственный irc-канал. Поэтому я рекомендую вам не медлить и скачать irc-клиент и хотя бы изредка посещать чаты.
- **Email автору библиотеки** должен быть одним из последних способов, которым вы пытаетесь найти ответ, но если вы абсолютно уверены, что ваша проблема (или багрепорт) никаким другим образом не может быть разрешена — почему бы не попробовать? К email-ам применимы все те же правила, что и к обычным вопросам.
- **Обращаться в службу поддержки хостинг провайдера** следует только в тех случаях, когда вы уверены, что проблема заключается не в вашем приложении и не в ошибочной настройке сервера, которую вы сами и произвели, а именно в хостинг-провайдере (что бывает довольно редко). Вы платите хостинг-провайдеру деньги за услуги, и это, безусловно, дает вам право беспокоить службу поддержки. Но не рассчитывайте, что она будет решать ваши проблемы за вас.
- Наконец, **мы всегда ждем вопросов на hasBrains** и с удовольствием отвечаем на вопросы новичков. Только не забудьте прочитать наши правила!

Основной принцип, который мне хотелось бы донести до вас, заключается в следующем: вопросы — это не просто способ решить проблему; это также способ получить новые знания, познакомиться и пообщаться с интересными людьми. Поэтому я надеюсь, что эта глава поможет вам не только ориентироваться в мире Ruby On Rails, но и в мире в целом. Удачи.