Любителям железных дорог,
больших и маленьких

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки всё равно случаются. Если вы найдёте ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьёзно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнётесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие

Ruby on Rails является одним из самых эффективных инструментов для разработки веб-приложений. Философия фреймворка, впервые появившегося на свет ещё в 2004 году, направлена на повышение производительности разработчиков и, как следствие, повышение скорости выпуска продуктовых релизов. Ruby on Rails – это фулстек-фреймворк (от англ. full-stack – «полный стек»), предоставляющий всё необходимое для разработки как серверной, так и клиентской части приложений.

В сердце архитектуры Rails лежит популярный принцип проектирования программного обеспечения – **Model-View-Controller**[1] (MVC). Данный принцип подразумевает разделение программы на три компонента: модель, отвечающая за работу с данными и бизнес-логику; представление, отвечающее за вывод информации для конечного пользователя; и, наконец, контроллер, который интерпретирует действия пользователя и может обновлять состояние модели или представления.

Помимо MVC, другой ключевой особенностью Ruby on Rails является приоритет соглашения над конфигурацией, знаменитый «convention-over-configuration» (далее – CoC). Фреймворк минимизирует объём кода и действий, необходимых для настройки приложения, при условии следования соглашениям об именовании. Такой подход позволяет значительно уменьшить количество решений, которых нужно принимать разработчику.

Вместе MVC и CoC образуют так называемый **Путь Rails** (*The Rails Way*) – идеологию разработки, которая позволяет последователям *Пути* больше фокусироваться на написании кода, имеющего непосредственное отношение к его продукту, нежели бороться с технологической сложностью фреймворка и его компонентов.

Следование *Пути Rails* помогает очень быстро пройти фазу от идеи до рабочего прототипа или даже минимального продукта, но грозит серьёзными *пробуксовками* в дальнейшем. Высокая скорость разработки – это не только благо, но и риск оказаться в ситуации, когда кодовая база превращается в запутанный лабиринт, полный ловушек и тупиков, который с трудом поддаётся изменениям и поддержке. Данная книга предлагает читателям стратегию и практические рекомендации по контролю роста сложности разработки приложений на Ruby on Rails и сохранению кодовой базы в поддерживаемом состоянии.

В процессе чтения вы познакомитесь с возможностями и принципами, лежащими в основе Rails, которые помогут вам раскрыть весь потенциал

---

[1] Модель–представление–контроллер.

фреймворка. Затем вы узнаете, как разделять ответственность в коде путём выделения новых слоёв абстракции, причём делать это так, чтобы не идти наперекор *Пути*. Таким образом, вы откроете для себя **Расширенный Путь Rails**, подход к проектированию Rails-приложений, одновременно следующий философии фреймворка и позволяющий избегать проблемы роста, сохраняя продуктивность разработки на высоком уровне.

По завершении вы станете лучше ориентироваться в проектировании веб-приложений с упором на долгосрочную продуктивную разработку, а также повысите степень владения фреймворком Ruby on Rails и его принципами.

Для кого эта книга

Данная книга будет особенно полезна разработчикам Rails-приложений, которые уже познали проблемы роста в проекте и ищут эффективные способы преодоления этих проблем.

Разработчики, которые только начали разрабатывать продукты с Ruby on Rails или находятся на этапе запуска MVP, также найдут данную книгу полезной – они узнают, какие опасности поджидают их на пути построения *волшебного монолита* и как их избежать.

Для эффективной работы с книгой вам потребуется понимание базовых принципов организации кода Rails-приложений (например, описанных в официальной документации), а также практический опыт в написании веб-приложений.

Технические требования

Примеры кода и описание работы фреймворка ориентируются на последние версии Ruby и Rails. На момент написания данной книги это Ruby 3.4 и Rails 8.0 соответственно. Большинство примеров актуальны и для более ранних версий.

Примеры кода также доступны в репозитории на GitHub: https://github.com/PacktPublishing/Layered-Design-for-Ruby-on-Rails-Applications. Все примеры интерактивны, так что не бойтесь экспериментировать с предложенными идеями.

Код на GitHub соответствует последнему изданию английской версии книги и может незначительно отличаться от кода в русскоязычном издании.

Комментарий к переводу

При работе над русскоязычным изданием у переводчика (по совместительству автора оригинального текста) возникла непростая задача: органично вписать обилие англоязычных терминов в адекватный перевод на русском

языке, который не был бы калькой с английского, набором разговорных англицизмов (рука не повернулась написать «вьюха») или мешаниной слов на двух языках («в этом mailer'е мы определили action...»).

В большинстве случаев автор придерживался существующей литературной терминологии, а также использовал такие интернет-источники, как, скажем, словарь проекта Веб-стандарты (https://github.com/web-standards-ru/dictionary) и MDN (https://developer.mozilla.org/ru). Оттуда, например, взят вариант перевода слова callback как «колбэк».

Во всех случаях при первом упоминании переводного термина даётся его оригинал, а иногда и обоснование выбранного перевода.

Наконец, *тональность* книги (как, кстати, и содержание) также претерпела некоторое изменение в сторону академичности. Например, шутливое при переводе на русский выражение «What a gem!» («Ай да гем!») пропало в пользу более лаконичного «Библиотека».

Буду рад вашим комментариям по поводу перевода. Хорошего чтения!

Часть I

ПОГРУЖЕНИЕ В МИР RAILS И ЕГО АБСТРАКЦИЙ

Первая часть данной книги посвящена самому фреймворку Ruby on Rails. Вы узнаете об архитектурных паттернах, решениях, лежащих в основе Rails, а также о концепциях и соглашениях, на которых построен фреймворк. Вы также познакомитесь с противоречивыми аспектами и ограничениями Ruby on Rails, которые мы попытаемся разрешить в последующих частях.

Эта часть состоит из следующих глав:

- главы 1 «Rails как инструмент для создания веб-приложений»;
- главы 2 «Активные модели и записи»;
- главы 3 «Больше адаптеров, меньше связи с реализацией»;
- главы 4 «Антипаттерны в Rails?»;
- главы 5 «Когда абстракций Rails уже недостаточно».

Глава 1

Rails как инструмент для создания веб-приложений

Ruby on Rails является одним из самых популярных инструментов для создания веб-приложений, большого класса компьютерных программ. В данной главе мы обсудим, в чём отличие и особенности данного класса программ и как это влияет на проектирование приложений. Вначале мы поговорим о модели взаимодействия «запрос–ответ» для HTTP-коммуникации и её связи с многоуровневой архитектурой, а также о компонентах Rails, отвечающих за работу с HTTP. Затем мы рассмотрим процессы приложения, которые происходят вне цикла «запрос–ответ», и, наконец, дойдём до уровня работы с данными.

Мы рассмотрим следующие темы:

- «Путешествие клика через слои абстракции»;
- «За пределами HTTP: фоновые задачи»;
- «Сердце веб-приложения – база данных».

В результате у вас будет более полное понимание основных принципов построения веб-приложений и того, как они влияют на архитектуру кода на базе Ruby on Rails.

Путешествие «клика» через слои абстракции

Основная задача любого веб-приложения – это обработка сетевых запросов. Таким образом, подразумевается передача данных через сеть интернет, а слово «запрос» указывает на то, что полученные сервером данные должны быть неким образом обработаны, и клиент должен быть проинформирован о результате этой обработки.

Рассмотрим, например, такое повседневное действие, как переход по ссылке на странице в браузере. Всего лишь один «клик» вызывает длинную цепочку операций, от определения IP-адреса по доменному имени до отображения новой страницы пользователю. Для современных приложений цепочка удлиняется ещё за счёт наличия промежуточных серверов (прокси, балансировщики нагрузки, CDN и т. д.). Для целей данной главы следующая, упрощённая схема *путешествия «клика»* будет достаточной.

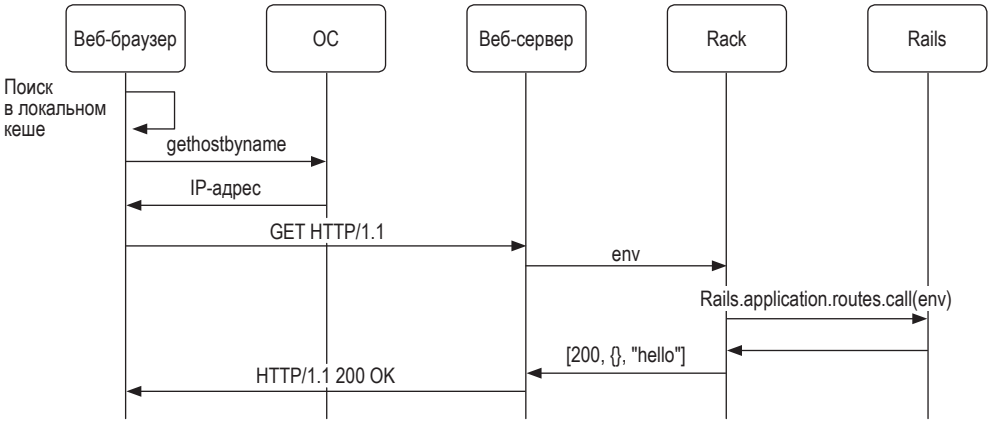

Рис. 1.1 ❖ Упрощённая схема путешествия «клика»

Часть этого путешествия, имеющая отношение к Rails, начинается в *веб-сервере*, например Puma (https://github.com/puma/puma). Веб-сервер отвечает за непосредственное обслуживание сетевых соединений, трансформацию HTTP в понятный для Ruby приложений формат, вызов кода Rails приложения, а затем отправку HTTP-ответа.

Модели клиент-серверного взаимодействия

Веб-приложения могут использовать различные модели клиент-серверного взаимодействия, не только «запрос–ответ». Асинхронное взаимодействие (как правило, поверх протокола WebSockets) также популярно в Rails-приложениях, особенно с включением в стек по умолчанию технологии Hotwire (https://hotwired.dev/). Однако, как правило, альтернативные варианты общения между клиентом и сервером являются второстепенными, в то время как схема «запрос–ответ» остаётся основной. Поэтому в данной книге мы будем рассматривать только её.

От запросов к абстракциям в коде

Жизненный цикл веб-приложения состоит из начальной загрузки (настройка и инициализация компонентов) и фазы обработки запросов.

Во время фазы обработки запросов приложение выступает в роли исполнителя множества независимых **единиц работы**, которые из себя пред-

ставляют сетевые запросы. Независимость в данном случае означает, что обработка каждого конкретного запроса с точки зрения кода является автономным процессом, не зависящим от остальных запросов, происходивших до или выполняющихся одновременно с текущим. Отсюда также следует, что использование общего состояния при обработке запросов сведено к минимуму. В контексте Ruby это означает, что в процессе обработки запроса мы создаём множество объектов, время жизни которых не выходит за границы запроса.

Обработка сетевых запросов как независимых единиц работы позволяет интерпретировать роль веб-сервера как работу конвейерной линии: мы кладём исходный материал (данные запроса) на ленту, по которой он проходит через множество станков, а на выходе мы получаем изделие в упаковке (ответ клиенту). Эффективность работы линии зависит от того, насколько грамотно мы разбили весь процесс сборки на этапы, какие именно станки мы поставили.

Переходя обратно от инженерии реального мира к миру нулей и единиц, мы можем сказать, что *станками* при проектировании *сборочных линий* веб-приложений будут являться **слои абстракции**. Именно через них проходит запрос, формируя ответ. Эффективность же определяется тем, насколько выделенные нами слои абстракции помогают в написании и поддержке кода.

Но что вообще такое *хорошая* абстракция? Ответ на этот вопрос мы будем формировать на протяжении всей книги. Тем не менее уже сейчас мы можем сформулировать некоторые базовые свойства.

- Во-первых, мы хотим, чтобы абстракция отвечала **принципу единственной ответственности**. При этом мы допускаем, чтобы эта ответственность была достаточно широкой (т. е. никаких ограничений по числу публичных методов и прочих количественных характеристик – они имеют мало общего с дизайном ПО). В то же время мы будем стремиться к тому, чтобы ответственности разных слоёв абстракции не пересекались, то есть будем также следовать **принципу разделения ответственности**.
- Во-вторых, слои должны быть **слабо связаны** между собой и не должны иметь циклических или обратных зависимостей. Если мы представим слои как стопку тетрадей, которые мы прошиваем ниткой так, как происходит через них обработка сетевого запроса, игла должна делать лишь одно движение вниз, а затем одно вверх.
- В-третьих, абстракции должны служить средством **инкапсуляции**, отделять интерфейс от реализации. Именно выделение общего интерфейса является самым сложным в формировании хорошего слоя абстракции, не стоит игнорировать эту сложность и пытаться «срезать углы» – усилия, потраченные на нахождение хорошего интерфейса, окупятся с лихвой.

- В-четвёртых, абстракции должны быть спроектированы так, чтобы их можно было **тестировать в изоляции**. Как правило, это свойство следует само собой из предыдущих, но я бы хотел акцентировать на нём особое внимание: зачастую именно фокус на *тестируемости* помогает спроектировать хороший интерфейс для абстракции.

С точки зрения разработчика, абстракция хороша, если у неё чёткий и понятный интерфейс, она помогает решать определённый класс задач, код, использующий эту абстракцию, удобно изменять, тестировать и диагностировать в нём ошибки. «Чёткий и понятный» интерфейс подразумевает отсутствие лишней когнитивной нагрузки на разработчика; другими словами, это интуитивно понятный интерфейс, или *простой*. Проектирование *простых* интерфейсов – это довольно сложная задача; именно поэтому часто можно слышать мнение, что внедрение новых абстракций в кодовую базу чаще усложняет поддержку, нежели облегчает её. Цель данной книги – как раз научить вас избегать этой ловушки и научиться выделять из кодовой базы действительно *хорошие* абстракции.

Ещё один вопрос, который хотелось бы обсудить сразу: а сколько слоёв абстракции было бы неплохо иметь? Здесь, как вы, наверное, догадались, нет однозначного ответа.

Обратимся снова к аналогии с конвейером. Число этапов (станков) растёт по мере того, как усложняется технологический процесс. В некоторых случаях разбиение одного, многозадачного этапа на несколько более простых позволяет ускорить процесс сборки. Аналогично и число слоёв абстракции, как правило, растёт с развитием проекта и усложнением бизнес-логики. В реальном мире эффективность конвейера измеряется скоростью сборки изделий; в цифровом – скоростью выпуска новых релизов, отвечающих требованиям конечных пользователей. Скорость выпуска релизов зависит от большого числа факторов, многие из которых не имеют никакого отношения к коду. Мы всё же можем спроецировать этот показатель на код, используя такую характеристику, как **поддерживаемость** – насколько трудозатратно внедрение нового функционала и поддержка существующего.

Увеличивается ли поддерживаемость кода с добавлением каждого нового слоя абстракции? Конечно, нет. Разве кто-то проектирует конвейер таким образом, что закручивание каждой отдельной гайки выносится в отдельный шаг? А имеет ли смысл внедрять новый слой абстракции в код только ради увеличения числа слоёв? Этими риторическими вопросами предлагаю закончить тему абстрактных веб-приложений (и конвейеров) и перейти к Ruby on Rails.

Rails предлагает три основных слоя абстракции из коробки: контроллеры, модели и представления[1]. (Оставим за скобками вопрос о том, можно ли их считать *хорошими* согласно критериям, сформированным выше.) Такое

[1] В разговорном русском языке чаще используется «вьюхи».

небольшое количество абстракций благоприятно влияет на скорость разработки на старте – всего три места для добавления нового кода (представьте, если бы у нас сразу был десяток слоёв абстракции, как у *взрослых* приложений). В этом и есть квинтэссенция **Пути Rails**. В книге мы рассмотрим, как этот путь расширять, то есть как постепенно вводить новые слои абстракции в код, сохраняя фокус на разработке продукта.

Прежде чем расширять *Путь Rails*, нам необходимо получше разобраться в нём самом.

Rack

Rack (https://github.com/rack/rack) – это компонент, который отвечает за преобразование сырых HTTP-данных в понятный Ruby-программам формат (и в обратную сторону). Более точно, Rack предоставляет *абстрактный интерфейс*, описывающий две самые главные сущности HTTP-взаимодействия: *запрос* и *ответ*.

Rack также предлагает универсальную схему интеграции между веб-серверами (такими как Puma и Unicorn) и Ruby-приложениями. Используя код, мы можем представить эту схему следующим образом:

```
request_env = { "HTTP_HOST" => "www.example.com", ... }
response = application.call(request_env)
[status, headers, body_iterator] in response
```

Данные HTTP-запроса представлены как объект класса Hash (далее мы будем для краткости такие объекты называть «хеш»). Данный хеш содержит *переменные запроса*, включающие в себя HTTP-заголовки и специфичные для Rack свойства (например, rack.input для чтения тела запроса). Интерфейс и терминология уходят корнями в эпоху CGI-серверов, когда данные запроса передавались через переменные окружения в процесс-обработчик.

Common Gateway Interface (CGI)

Common Gateway Interface (https://www.w3.org/CGI, «общий интерфейс шлюза») – это одна из первых попыток стандартизировать интерфейс взаимодействия между веб-серверами приложения. Согласно данному интерфейсу, совместимое приложение обязано считывать заголовки запроса из переменных окружения, а тело запроса – из стандартного устройства ввода (STDIN); ответ на запрос, в свою очередь, должен быть записан в стандартное устройство вывода (STDOUT). При этом для обработки каждого запроса CGI сервер запускает отдельный процесс приложения – непозволительная роскошь по современным меркам. Впоследствии возник стандарт FastCGI, который позволил использовать запущенный процесс приложения многократно.

Для совместимости с Rack от приложения требуется только одно – реализовать метод `#call`, принимающий на вход хеш с данными запроса. Rack ожидает в качестве возвращаемого значения массив (Array) из трёх элемен-