

УДК 004.738.5:004.438Ruby on Rails
ББК 32.973.2
Д30

Дементьев В. М.

Д30 Проектирование приложений Ruby on Rails слой за слоем. 2-е изд. / пер. с англ. В. М. Дементьева. – М.: ДМК Пресс, 2026. – 382 с.: ил.

ISBN 978-5-93700-426-0

В этой книге вы познакомитесь с принципами, лежащими в основе фреймворка Ruby on Rails, получите практические рекомендации по управлению разработкой приложений и сможете найти эффективные способы преодоления проблем, связанных с ростом сложности проекта. Второе издание дополнено главами про конечные автоматы и различные аспекты внедрения функционала, опирающегося на искусственный интеллект (LLM), а оригинальные главы первого издания расширены новыми разделами. Все примеры кода и описания библиотек были обновлены до актуальных. В книге используется версия Rails 8.1.

Читателю необходимо понимать базовые принципы организации кода Rails и иметь практический опыт разработки веб-приложений. Издание будет полезно как опытным разработчикам Rails-приложений, так и новичкам, приступающим к освоению Ruby on Rails.

УДК 004.738.5:004.438Ruby on Rails
ББК 32.973.2

First published in the English language under the title 'Layered Design for Ruby on Rails Applications. Second Edition – (9781806114238)'.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-80611-423-8 (англ.)
ISBN 978-5-93700-426-0 (рус.)

© 2025 Packt Publishing
© Перевод, оформление, издание,
ДМК Пресс, 2026

Содержание

От издательства	11
Предисловие	12
Часть I. Погружение в мир Rails и его абстракций	15
Глава 1. Rails как инструмент для создания веб-приложений	16
Путешествие «клика» через слои абстракции	16
От запросов к абстракциям в коде	17
Rack	20
Rails on Rack	21
Маршрутизация в Rails	25
С означает «Controller»	26
За пределами HTTP: фоновые задачи	27
О необходимости фоновых задач	27
Фоновые задачи как единицы работы	29
Задачи по расписанию	30
Сердце веб-приложения – база данных	31
Влияние абстракций на производительность базы данных	32
Абстракции на уровне базы данных	33
Итоги	35
Проверь себя	35
Упражнение	35
Глава 2. Активные модели и записи	36
Обзор Active Record: от работы с базой до всего подряд	36
Объектно-реляционное отображение	37
От отображения к моделированию	41
От моделирования к чему угодно	46
Active Model – секретный ингредиент Active Record	47
Active Model как интерфейс	47
Active Model как спутник Active Record	50
Производительность Active Model и простых Ruby-классов	53
Active Model для предоставления знакомого Active Record-подобного интерфейса	55
В поисках всемогущества	55
Итоги	57
Проверь себя	57
Упражнение	58

Глава 3. Больше адаптеров, меньше связи с реализацией	59
Active Job как универсальный интерфейс очереди задач	59
Адаптеризация очередей	61
Сериализуй это	63
Адаптеры и плагины в Active Storage	67
Адаптеры и плагины	69
Адаптеры в вашем коде	71
Итоги	74
Проверь себя	74
Упражнение	74
Глава 4. Антипаттерны в Rails?	75
Колбэки, колбэки повсюду	75
Колбэки под контролем (в контроллерах)	76
Колбэки Active Record выходят из-под контроля	80
Озабоченность консёрнами в Rails	89
Разделяем поведение, а не код	91
Консёрны остаются модулями со всеми их недостатками	93
Композиция объектов	94
О глобальном и текущем состоянии	99
Текущее «всё подряд»	100
Итоги	104
Проверь себя	105
Упражнение	105
Глава 5. Когда абстракций Rails уже недостаточно	106
Проклятие толстых (тонких) контроллеров и тонких (толстых) моделей	106
От толстых контроллеров к толстым моделям	107
Пример толстого контроллера	108
Рефакторинг в соответствии с принципом тонких контроллеров и толстых моделей	110
От толстых моделей к сервисам	112
Сервисы общего назначения и специализированные абстракции	116
Связь между многоуровневой архитектурой и слоями абстракции	118
Итоги	121
Проверь себя	121
Часть II. Выделение абстракций из моделей	122
Глава 6. Абстракции слоя данных	123
Использование объектов запросов для вынесения (сложных) запросов из моделей	124
Выделение объектов запросов	126
Скоупы и объекты запросов	130
Объекты запросов общего пользования и Arel	132
Место объектов запросов в многоуровневой архитектуре	137

Отделение моделей от хранилища данных с помощью репозитория	138
Итоги	141
Проверь себя	141

Глава 7. Моделирование состояний и переходов 142

Скрытые состояния и переходы	142
Конечные автоматы на службе у моделей	145
От скрытых машин состояний ко встроенным	147
От встроенных конечных автоматов к изолированным машинам состояний и рабочим процессам	158
Когда конечные автоматы приносят больше вреда, чем пользы	165
Итоги	166
Проверь себя	166
Упражнения	167

Глава 8. Обработка пользовательского ввода за пределами моделей 168

Объекты форм: ближе к интерфейсу, дальше от схемы данных	169
Формы ввода и модели	169
Использование Active Model для абстракции объектов форм	178
Объекты фильтров, или Построение запросов на основе пользовательского ввода	194
Фильтрация в контроллерах	195
Перенос фильтрации на уровень модели	196
Выделение объектов фильтров	197
Сравнение объектов фильтров, объектов форм и объектов запросов	200
Итоги	201
Проверь себя	201
Упражнения	201

Глава 9. Выделение презентационной логики из моделей 203

Использование презентеров для отделения моделей от представлений	203
Хелперам место в библиотеках	205
Презентеры и декораторы	206
Презентеры как слой абстракции	211
Сериализаторы как презентеры для вашего API	215
Преобразование модели в JSON	216
Сериализаторы как презентеры для API	217
Итоги	222
Проверь себя	223

Часть III. Слои абстракций на каждый день 224

Глава 10. Модели и слои авторизации 225

Авторизация, аутентификация и другие аспекты безопасности	225
Разница между аутентификацией и авторизацией	226

Линии обороны веб-приложения.....	227
Модели авторизации	228
Безмодельная авторизация.....	229
Классические модели авторизации	229
Обеспечение контроля доступа, или Необходимость абстракций авторизации.....	234
Внедрение политик	235
Формирование авторизационного слоя абстракции	236
Авторизация в шаблонах представления	241
Влияние авторизации на производительность.....	245
Проблема N + 1 авторизации на уровне представления.....	245
Авторизация на основе выгрузки данных	247
Итоги	248
Проверь себя	248
Упражнение.....	249
Глава 11. Формирование абстрактного слоя уведомлений.....	250
От Action Mailer к многоканальной связи с пользователем	250
Action Mailer в действии	251
Место рассылщиков почты в многоуровневой архитектуре.....	252
Выделение абстрактного слоя для работы с уведомлениями	256
Самодельная абстракция.....	256
Использование сторонних библиотек для организации работы с уведомлениями	260
Моделирование пользовательских настроек уведомлений	267
Битовые атрибуты и объекты-значения	268
Хранилище настроек уведомлений	270
Использование отдельной таблицы для настроек уведомлений	271
Итоги	271
Проверь себя	272
Упражнения	272
Глава 12. HTML под контролем абстракций	273
V в MVC Rails: шаблоны и хелперы	273
Пользовательский интерфейс без программного интерфейса	275
Переиспользование и дизайн-системы	280
Компонентный подход.....	283
Превращаем фрагменты и хелперы в компоненты	283
Компоненты интерфейса как слой абстракции	286
Компоненты интерфейса без HTML	291
Компоненты как связующее звено между командами.....	292
Итоги.....	292
Проверь себя	293
Глава 13. Абстракции в эпоху рассвета ИИ	294
AI на расстоянии одного API-вызова?	295

AI в моделях: суммаризация статей	295
AI в контроллерах: перевод по запросу	297
Недостатки внедрения AI без соответствующих абстракций.....	300
Агенты к вашим услугам.....	301
От API к агентам.....	302
Автономные агенты и рабочие процессы.....	310
Погружение в агенты.....	320
RAG, MCP и другие	332
Контекст-инжиниринг вашей предметной области.....	332
Слой представления для AI-клиентов	336
Итоги.....	339
Проверь себя	339

Глава 14. Конфигурация как первоклассная сущность

приложения	340
Виды настроек и источников данных конфигурации	340
Файлы, секреты, зашифрованные хранилища и многое другое	341
Настройки и секреты.....	344
Окружения приложения и провайдеры данных	346
Многоуровневая архитектура и конфигурация	346
Использование объектов предметной области для укрощения настроек приложения.....	347
Отделение кода приложения от источников конфигурации	347
Использование классов конфигурации	353
Итоги.....	357
Проверь себя	358
Упражнение.....	358

Глава 15. Сквозь слои и дальше

Разнообразие инфраструктурного уровня в Rails.....	359
Инфраструктурные абстракции и реализации	360
Сквозь уровни: логирование и мониторинг	361
Логирование	361
Отслеживание исключений.....	365
Инструментация	366
Вынесение низкоуровневой реализации в отдельный сервис.....	372
Отпочковываем веб-сокеты от Action Cable с помощью AnyCable	372
Обработка изображений на лету, но не в Rails.....	373
Итоги.....	376
Проверь себя	377

Предметный указатель	378
-----------------------------------	-----

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 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, которые 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 избежать 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 которые 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 версии Ruby и Rails. На момент написания данной книги это Ruby 3.4 и Rails 8.1 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-Second-Edition>. Все примеры интерактивны, так что не бойтесь экспериментировать с предложенными идеями.



Код на GitHub соответствует последнему англоязычному изданию 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, например, словарь проекта Веб-стандарты (<https://github.com/web-standards/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 упоминается 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. В начале 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.

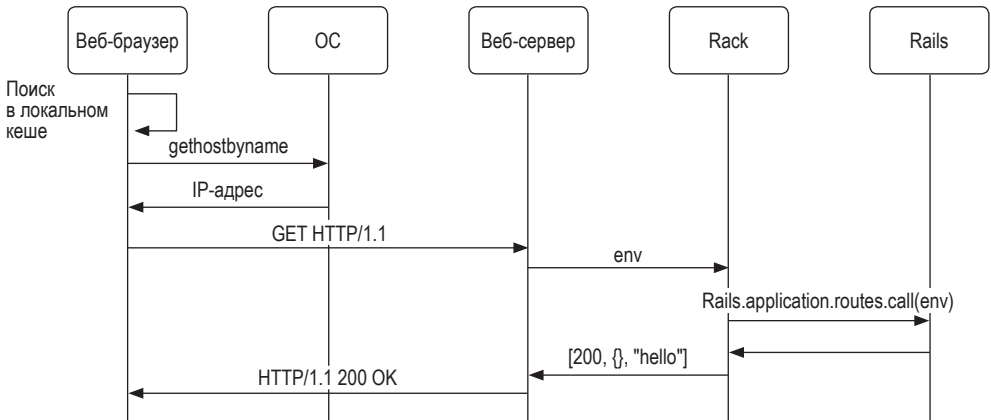


Рис. 1.1 ❖ Упрощённая схема путешествия «клика»

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 представляют сетевые запросы. Независимость в данном случае означает, что обработка каждого конкретного запроса с точки зрения кода является автономным процессом, не зависящим от остальных запросов, происшедших 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 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 сле-

дует 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*.

Вышеупомянутые свойства имеют ещё одно следствие, которое становится особенно полезным в эпоху разработки с ИИ-ассистентами: ограниченность контекста, который необходимо держать в голове (или что там у ИИ?). Грамотно определённые интерфейсы и границы между слоями абстракций, наряду с устоявшимися соглашениями, значительно повышают качество кода, генерируемого ИИ, и сокращают число требуемых итераций.

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? Ответ, как не сложно догадаться, зависит от многих факторов.

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 атомарных 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; 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 трудозатратны 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 представления¹. (Оставим за скобками вопрос о том, можно ли

¹ В разговорном русском языке чаще используется «вьюхи».

их считать *хорошими* согласно критериям, сформированным выше.) Такое 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 `Nash` (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 и приложениями. 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 – nepозволительная 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 элементов: код состояния в виде числа (200, 401 и т. д.), хеш HTTP-заголовков ответа и тело ответа, представленное *перечисляемым* («enumerable») объектом (т. е. любой Ruby-объект, который реализует метод `#each`, возвращающий строки). Почему формат тела ответа такой сложный, почему бы просто не передавать весь текст ответа как строку? Дело в том, что использование перечисляемого интерфейса позволяет в рамках стандарта реализовать потоковую передачу данных, тем самым сокращая потребление памяти.

Простейшим Rack-приложением является *лямбда* («lambda»), возвращающая фиксированный массив ответа. Вы можете проверить это самостоятельно, используя утилиту `rackup`¹:

```
$ rackup -s webrick --builder 'run ->(env) { [200, {}, ["Привет, Rack!"]] }'
[2024-07-25 11:15:44] INFO WEBrick 1.9.1
[2024-07-25 11:15:44] INFO WEBrick::HTTPServer#start: pid=85016 port=9292
```

Откройте страницу `http://localhost:9292` в браузере, и вы увидите надпись «Привет, Rack!» в пустом окне.

Rails on Rack

Rails-приложения реализуют интерфейс Rack, а это значит, что где-то во фреймворке реализован тот самый метод `#call`. Давайте узнаем, где.

В корне любого Rails-приложения вы можете обнаружить загадочный файл – `config.ru`. Что за странное расширение – «ru»? Нет, это не сокращение от «russian» или «ruby» (признайтесь, у вас были такие догадки); «ru» расшифровывается как «rack-up», а сам файл является конфигурационным файлом для Rack, а также файлом-загрузчиком приложения. Давайте заглянем внутрь него:

```
require_relative "config/environment"
run Rails.application
```

В коде выше `Rails.application` – это глобальный экземпляр Rails-приложения; команда `run` указывает, что именно на объекте приложения Rack должен вызывать `#call` – это и есть точка входа HTTP-трафика в Rails-приложение.

Итак, мы наконец-то определили, где «клик» в своём путешествии пересекает границу Ruby on Rails. Дальнейшие его передвижения будем рассматривать более пристально.

¹ Библиотеки `rackup` и `webrick`, как правило, присутствуют в системе; если они у вас ещё не установлены, вы можете сделать это с помощью команды `gem install rackup webrick`.

Чтобы оценить масштаб работы, которую проделывает фреймворк, обрабатывая каждый запрос, мы можем посмотреть на все методы, которые выполняются в процессе. Для этого мы воспользуемся библиотекой `trace_location`.

Библиотека: `trace_location`

Библиотека `trace_location` (https://github.com/yhirano55/trace_location) – это лучший друг любознательного разработчика. Она позволяет заглянуть под капот любого Ruby-метода. Особенно интересно с помощью неё *подглядывать* за другими библиотеками и фреймворками. Вы удивитесь, насколько сложными в реализации порой бывают *безобидные* и простые на вид интерфейсы (например, `user.save` из Active Record) – проектирование удобных интерфейсов для решения нетривиальных задач требует настоящего мастерства. Если же заглянуть внутри самой `trace_location`, то можно увидеть там использование TracePoint API (<https://rubyapi.org/3.4/o/tracepoint>) – одного из мощнейших инструментов интроспекции в Ruby.

Для того чтобы проследить за тем, как Rails обрабатывает запросы, совсем необязательно запускать веб-сервер. Мы можем эмулировать работу с запросами прямо из консоли (`rails c`), используя следующий код:

```
request =
  Rack::MockRequest.env_for('http://localhost:3000')
TraceLocation.trace(format: :log) do
  Rails.application.call(request)
end
```

В результате выполнения данного кода `trace_location` запишет все вызовы методов в файл. Откройте его. Даже для нового Rails-приложения вы обнаружите тысячи строк логов вызовов – как видите, даже обработка простого GET-запроса является далеко не тривиальной задачей для фреймворка.

Помимо числа вызовов методов, мы можем также оценить число создаваемых объектов для обработки HTTP-запроса. Тут мы можем обойтись без дополнительных инструментов – всё необходимое есть в самом Ruby:

```
was_alloc = GC.stat[:total_allocated_objects]
Rails.application.call(request)
new_alloc = GC.stat[:total_allocated_objects]
puts "Total allocations: #{new_alloc - was_alloc}"
```

Даже для запроса, который ничего не делает и возвращает пустой ответ (`head :ok`), у меня получилось около **трёх тысяч** созданных объектов. Можно считать это число нижней границей для числа объектов, которые создаются в процессе обработки одного запроса Rails-приложением.

О чём нам говорят эти цифры? И хотя целью данной книги является демонстрация того, как грамотное внедрение и выделение абстракций помогают поддерживать кодовую базу в *здоровом* состоянии, мы не должны забывать о потенциальном влиянии архитектуры на производительность. Да, каждая новая абстракция на пути «клика» несколько увеличивает число вызовов методов и объектов, но в сравнении с теми величинами, которые уже есть в самом фреймворке, данными абстрактными *добавками* можно пренебречь. Таким образом, в Rails-приложениях **добавление слоя абстракции** само по себе **не ухудшает производительность** (чего не скажешь о человеческом факторе).

Запустим ещё раз наш трассировщик (`trace_location`), но в этот раз рассмотрим только вызовы метода `#call`:

```
TraceLocation.trace(format: :log, methods: [:call]) do
  Rails.application.call(request)
end
```

В лог-файле в этот раз будет всего пара сотен строчек:

```
[Tracing events] C: Call, R: Return
C /usr/local/lib/ruby/gems/3.1.0/gems/railties-7.0.3.1/lib/rails/engine.rb:528
[Rails::Engine#call]
  C /usr/local/lib/ruby/gems/3.1.0/gems/actionpack-7.0.3.1/lib/action_dispatch/
  middleware/host_authorization.rb:130 [ActionDispatch::HostAuthorization#call]
    C /usr/local/lib/ruby/gems/3.1.0/gems/rack-2.2.4/lib/rack/sendfile.rb:109
    [Rack::Sendfile#call]
      C /usr/local/lib/ruby/gems/3.1.0/gems/actionpack-7.0.3.1/lib/action_dispatch/
      middleware/static.rb:22 [ActionDispatch::Static#call]
        // ...
        R /usr/local/lib/ruby/gems/3.1.0/gems/actionpack-7.0.3.1/lib/action_dispatch/
        middleware/static.rb:24 [ActionDispatch::Static#call]
          R /usr/local/lib/ruby/gems/3.1.0/gems/rack-2.2.4/lib/rack/sendfile.rb:140
          [Rack::Sendfile#call]
            R /usr/local/lib/ruby/gems/3.1.0/gems/actionpack-7.0.3.1/lib/action_dispatch/
            middleware/host_authorization.rb:131 [ActionDispatch::HostAuthorization#call]
          R /usr/local/lib/ruby/gems/3.1.0/gems/railties-7.0.3.1/lib/rails/engine.rb:531
          [Rails::Engine#call]
```

Можно заметить, что каждый метод присутствует в файле дважды: в момент входа в него (помечен буквой «С» – «call»), а затем на выходе («R» – «return»). Более того, эти методы *вложены* в друг друга по подобию матрёшки: таким образом мы можем видеть в действии ещё одно свойство Rack – использование промежуточных обработчиков (**middlewares**).

Паттерн: Middleware

Middleware («промежуточное ПО» или «промежуточный обработчик») представляет собой компонент, который *оборачивает* выполнение некоторой процедуры, может инспектировать и модифицировать её входные и выходные данные, не меняя при этом их интерфейса. Как правило, промежуточные обработчики формируют цепочку, вызывая друг друга по порядку, пока последний в цепочке не выполнит процедуру. Такой подход позволяет сохранять обработчики однозадачными и переиспользуемыми. Классические сценарии использования промежуточных обработчиков включают в себя добавление логирования или инструментации, проверки доступа (в этом случае цепочка вызова может быть прервана). Данный паттерн проектирования популярен в Ruby-фреймворках, и помимо Rails вы можете встретить его в таких библиотеках, как Sidekiq, Faraday и AnyCable. Вне Ruby самым популярным примером использования паттерна Middleware можно назвать фреймворк Express.js.

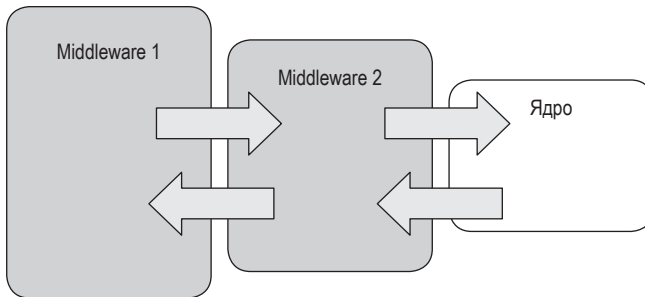


Рис. 1.2 ❖ Схема паттерна Middleware

Rack как Ruby-фреймворк позволяет расширять логику процесса обработки запросов путём добавления промежуточных слоёв обработки. Каждый такой слой *перехватывает* данные запроса до попадания непосредственно в приложение, может изменять их, выполнять второстепенные действия (например, логировать запрос), делегировать дальнейшую обработку ниже по цепочке промежуточных слоёв вплоть до непосредственно приложения, а на выходе обогащать данные ответа (например, добавлять заголовки типа «X-Runtime»).

Rails подключает более двадцати промежуточных обработчиков по умолчанию. Вы можете посмотреть весь список, или *стек промежуточной обработки* («middleware stack»), выполнив команду `bin/rails middleware`:

```

$ bin/rails middleware
use ActionDispatch::HostAuthorization
use Rack::Sendfile
use ActionDispatch::Static
use ActionDispatch::Executor
use ActionDispatch::ServerTiming
use Rack::Runtime
  
```

```
...
use Rack::Head
use Rack::ConditionalGet
use Rack::ETag
use Rack::TempfileReaper
run MyProject::Application.routes
```

Rails позволяет вам управлять этим списком: добавлять, удалять или переставлять местами промежуточные обработчики. Стек промежуточной обработки должен рассматриваться только как *слой пост- и препроцессинга HTTP-данных*. Он должен работать с приложением как с чёрным ящиком, не зная ничего о его бизнес-логике. Таким образом, промежуточные обработчики должны выступать лишь в роли посредников между внешним миром и Rails-приложением, не изменяя поведение последнего.

Маршрутизация в Rails

В конце цепочки промежуточных обработчиков (см. выше) мы можем видеть уже знакомую нам команду `run` с объектом `Application.routes` в качестве аргумента. Это объект класса `ActionDispatch::Routing::RouteSet`, также являющийся Rack-приложением и отвечающий за маршрутизацию запросов. Он использует ваш файл `routes.rb` для сопоставления пути (и других параметров запроса) паре контроллер–действие или другому Rack-приложению. Рассмотрим пример файла маршрутизации с различными вариантами определения целевых объектов:

```
Rails.application.routes.draw do
  # Так мы определяем набор путей для ресурса, которые будут обрабатываться одним
  # контроллером (PostsController)
  resources :posts, only: [:show]

  # redirect() создаёт специальное Rack-приложение для переадресаций запросов
  get "docs/:article",
    to: redirect("/wiki/#{article}")

  # Можно прямо здесь указать для обратки лямбду - ведь это тоже может быть полноценное
  # Rack-приложение
  get "/_health", to: -> _env {
    [200, { "Content-Type" => "text/html" }, ["??? ???"]]
  }

  # Наконец, мы можем смонтировать Rack-приложение и проксировать через него все подходящие
  # запросы
  mount ActionCable.server, at: "/cable"
end
```

Файл `routes.rb` описывает *слой маршрутизации* Rails-приложения. Вы можете задаться вопросом: зачем добавлять Rack-приложения в настройки

маршрутизации приложения, а не реализовывать их через промежуточные обработчики на уровне Rack? Переадресация и веб-сокеты (Action Cable) являются неотъемлемой частью функционала и бизнес-логики приложения; перенеся их на уровень Rack, мы бы нарушили соглашение о чёрном ящике. Проверка работоспособности («health-check») может быть рассмотрена и как часть слоя Rack, и как часть приложения (если, например, для формирования ответа так же анализируется внутреннее состояние приложения).

Размышление о том, в каком слое обработки запроса реализовывать тот или иной функционал, можно развернуть и в другую сторону. Например, можно задаться вопросом: а почему бы нам не реализовать переадресацию в контроллере?

С означает «Controller»

Контроллеры формируют следующий слой, через который проходит обработка сетевых запросов. Это первый полноценный *абстрактный слой* на нашем пути. Само понятие «контроллер» подразумевает предоставление универсального, стандартного интерфейса для обработки *входящих запросов*. В теории контроллеры могут использоваться не только для HTTP, но и для других видов запросов, так как контроллер является всего лишь абстракцией на уровне кода. На практике же мы почти не встречаем использования контроллеров за пределами HTTP/Rack. Отчасти это обусловлено тем, что контроллеры в Rails, хоть и являются абстракцией, спроектированы именно для работы с Rack. У нас даже есть специальный API для преобразования метода контроллера в Rack-приложение:

```
Rails.application.routes.draw do
  # Код ниже эквивалентен:
  # resources :posts, only: %i[index]
  get "/posts", to: PostsController.action(:index)
end
```

MVC

Model-view-controller («модель–представление–контроллер») – это один из старейших и фундаментальных подходов к архитектуре ПО, разработанный ещё в 70-х годах XX в. для первых программ с графическим интерфейсом. Данный подход подразумевает разбиение системы на три концептуальных компонента: модель, представление и контроллер. Контроллер отвечает за обработку пользовательского ввода и изменение модели; модель, в свою очередь, вызывает обновление представления, которое отвечает за непосредственно отрисовку интерфейса для пользователя. И хотя Rails часто называют MVC-фреймворком, потоки данных и управления в нём отличаются от оригинальной схемы: контроллеры отвечают за обновление представления, а представлениям не запрещено напрямую взаимодействовать или даже изменять модели.

Слой контроллеров отвечает за **преобразование веб-запросов в действия** и операции на уровне бизнес-логики, а также за инициацию обновлений интерфейса. Тут мы как раз наблюдаем пример единственной ответственности широкого спектра, так как это самое «преобразование в действие» включает в себя: аутентификацию и авторизацию пользователя, проверку входных параметров и т. п. Аналогичным свойством широкой ответственности обладают и другие *входные слои абстракции* в Rails, такие как каналы в Action Cable или почтовые ящики в Action Mailbox.

Теперь, познакомившись с HTTP-слоями абстракции Rails, мы можем ответить на вопрос о месте переадресации в стеке HTTP: так как переадресация сама по себе не является операцией на уровне бизнес-логики, реализовывать её в контроллере было бы нецелевым использованием абстракции.

Мы вернёмся к более детальному обсуждению контроллеров в последующих главах.

За пределами HTTP: фоновые задачи

Несмотря на то что ключевой задачей веб-приложений мы определили обработку сетевых запросов, это не единственная *работа*, которую выполняют Rails-приложения. Большое количество операций выполняется в фоне.

О необходимости фоновых задач

Один из ключевых показателей любого веб-приложения – это его **пропускная способность**, то есть число запросов, которые приложение может обработать в секунду (RPS, «requests-per-second») или минуту (RPM, «requests-per-minute»).

Современные веб-приложения на Ruby, как правило, обрабатывают запросы, используя конечное число *потоков обработки*, представленных либо отдельными процессами, либо *тредами* (Thread), либо их комбинацией. Независимо от реализации каждый поток обработки одновременно может заниматься только одним запросом. Таким образом, пропускная способность прямо зависит от числа потоков обработки, запущенных приложением. Почему бы нам просто не увеличивать их число по мере необходимости?

Такая специфика виртуальной машины Ruby (в случае CRuby), как GVL¹ (Global Virtual machine Lock), накладывает ограничение сверху на число тредов, которые могут эффективно работать в одном процессе. Обычно их число невелико, от 3 до 5.

¹ Автор предполагает, что читатель в общих чертах знаком с феноменом GVL; подробное же описание того, как и почему GVL работает, а также чем это грозит Ruby-приложениям, вы можете найти в статье: <https://www.speedshop.co/2020/05/11/the-ruby-gvl-and-scaling.html>.

Выбор оптимального числа тредов

С версии 3.2 в Ruby появился функционал, позволяющий замерять, сколько времени тред проводит в фазе ожидания ввода-вывода. Используя такие библиотеки, как `gvltools` (<https://github.com/Shopify/gvltools>) и `gvl-tracing` (<https://github.com/ivoanjo/gvl-tracing>), вы можете проанализировать эффективность использования тредов вашим приложением и подобрать их оптимальное количество.

Масштабирование путём увеличения числа процессов ОС, в свою очередь, может приводить к кратному потреблению оперативной памяти, а значит, тоже не может являться универсальным способом увеличения пропускной способности. Какие ещё есть варианты?

Fibers

Ruby предлагает ещё один примитив для конкурентной работы – **файберы** (Fibers, <https://rubyapi.org/3.4/o/fiber>). Файберы могут рассматриваться как более легковесная альтернатива тредам, которая к тому же использует кооперативную модель для конкурентного выполнения кода. В этой модели за переключение контекста выполнения отвечает только сам файбер, а не виртуальная машина. В современных версиях Ruby файберы автоматически уступают контекст исполнения при выполнении операций ввода-вывода (например, сетевое или файловое взаимодействие). Именно поэтому файберы всё чаще рассматриваются для реализации потоков обработки в веб-приложениях на Ruby. Долгое время было невозможно полноценно использовать файберы с Ruby on Rails, однако с версии 7.2 все известные проблемы совместимости были устранены, и теперь вы можете попробовать файберы в деле, используя веб-сервер Falcon (<https://github.com/socketry/falcon>) вместо Puma или Unicorn.

На протяжении многих лет Rails-приложения решали проблему пропускной способности следующим образом: для уменьшения времени выполнения запросов как можно большее количество вычислений выносятся в фон, за пределы цикла «запрос–ответ». За фоновые вычисления, как правило, отвечает отдельный процесс приложения, который получает *задачи* от веб-сервера. Данная идея обрела огромную популярность во многом благодаря таким инструментам, как Sidekiq и Delayed Job, которые упростили работы с фоновыми задачами в Rails. И только в Rails 4 данный подход стал *официальным* с появлением Active Job.

Библиотека: Sidekiq

Sidekiq (<https://github.com/mperham/sidekiq>) является одной из самых популярных библиотек на Ruby в принципе и самой популярной для работы с фоновыми задачами. Опираясь на предположение о том, что большинство фоновых задач много работают с вводом-выводом, Sidekiq эффективно использует треды для их выполнения. В качестве шины данных Sidekiq использует Redis, что также положительно сказывается на производительности и функциональных возможностях.

Что же из себя представляет фоновая задача? Это некоторая операция, которая выполняется вне цикла жизни веб-сервера, то есть за пределами обработки сетевого запроса.

Простейшим примером фоновой задачи может быть отправка e-mail сообщения. Чтобы отправить сообщение на электронную почту, нам нужно выполнить сетевой запрос (SMTP или HTTP). Нужно ли нам ждать, пока почтовый сервер обработает его, и тем самым задерживать возвращение HTTP-ответа пользователю? Чаще всего нет. Тогда как нам *вырваться* из цикла «запрос–ответ» и отправить почту асинхронно? В Ruby мы можем это сделать, например, используя треды:

```
class PasswordResetController < ApplicationController
  def create
    user = User.find_by!(email: params[:email])
    Thread.new do
      UserMailer.with(user:).reset_password.deliver_now
    end
  end
end
```

И хотя формально задача решена, в этом решении много «дыр»: мы не контролируем число тредов (а их создание не бесплатно), мы не отлавливаем в них исключения и никак не реагируем на возможные ошибки. Именно для учёта всех этих нюансов нам и требуется полноценный абстрактный слой для работы с фоновыми задачами. Например, такой как Active Job.

Фоновые задачи как единицы работы

Слой фоновых задач оперирует двумя основными понятиями: **задача** и **очередь**.

Задача как Ruby-объект описывает, какие операции бизнес-логики необходимо выполнить, а также содержит параметры обработки, используемые менеджером фоновых задач: приоритет выполнения, логика перезапуска в случае возникновения ошибок и т. д. Именно поэтому нам и нужна отдельная абстракция; простой Ruby-объект не подходит, так как не имеет контекста фонового выполнения.

Очереди возникают в фоновых задачах естественным образом: обычно мы хотим выполнять наши отложенные операции в порядке постановки, **first in, first out**. Менеджеры фоновых задач могут использовать разные структуры данных и системы хранения для управления задачами; мы лишь требуем от них соблюдения интерфейса очереди.

Мы предполагаем, что фоновые задачи могут выполняться независимо друг от друга¹, а значит, и параллельно (или конкурентно). Таким образом,

¹ Выполнение фоновых процессов («workflows»), состоящих из нескольких задач, запускаемых в определённом порядке, пока оставим за скобками.

как и сетевые запросы, фоновые задачи являются **единицами работы** приложения.

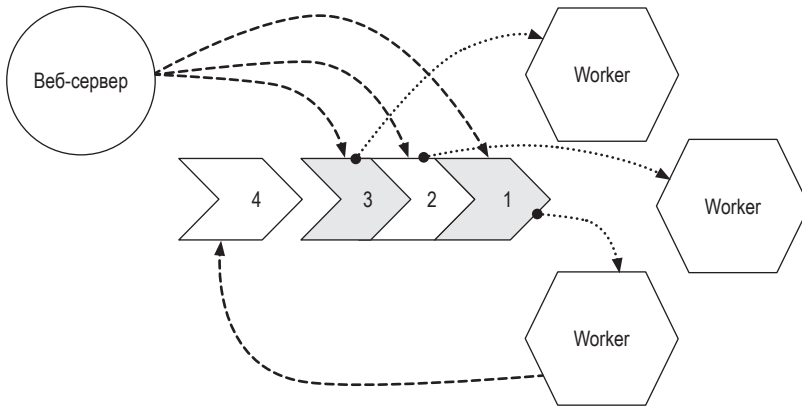


Рис. 1.3 ❖ Верхнеуровневая схема архитектуры процесса обработки фоновых задач

С каждой единицей работы в Rails-приложении мы можем связать соответствующий **контекст выполнения**. Контекст выполнения описывает состояние окружения, сформированного инициатором единицы работы. Например, для HTTP-запросов окружение естественным образом содержит сессию пользователя, а также характеристики самого запроса и его производные (например, параметры локализации). Для фоновых задач у нас нет аналогичного естественного контекста, мы должны формировать его сами – это также является утилитарной функцией задачи как абстракции.

Фоновые задачи являются примером *внутреннего входного слоя абстракции*. В отличие от внешних входных слоёв (например, контроллеров), у нас тут нет пользовательского ввода, а значит, отсутствует необходимость в таких операциях, как аутентификация, авторизация, проверка параметров. В остальном же фоновые задачи близки по своей роли и дизайну к контроллерам.

Задачи по расписанию

Задачи по расписанию являются специальным подмножеством фоновых задач. Основное (и, по сути, единственное) их отличие заключается в том, что инициатором постановки задачи выступает не действие пользователя, а время.

В контексте Rails, однако, есть и ещё одно отличие: фреймворк (конкретно Active Job) не предоставляет функционала для конфигурации задач по рас-

писанию из коробки. А значит, каждая команда стоит перед проблемой выбора (или изобретения велосипеда), когда вопрос – о добавлении подобного функционала. Кроме того, есть риск *сломать абстракцию*. Многие библиотеки, например `whenever` (<https://github.com/javan/whenever>) или `rufus-scheduler` (<https://github.com/jmettraux/rufus-scheduler>), позволяют выполнять произвольный Ruby-код или системную команду по расписанию, не только ставить фоновые задачи в очередь. Такие *псевдозадачи* теряют все преимущества использования абстракции (обработку ошибок, логирование и т. д.) и увеличивают сложность кодовой базы.

Задачи по расписанию должны быть частью того же слоя абстракции, что и остальные фоновые задачи, а не существовать отдельно от него.

Библиотека: `solid_queue`

В Rails 8 менеджером фоновых задач по умолчанию станет библиотека `Solid Queue` (https://github.com/rails/solid_queue). В качестве хранилища она использует любую базу данных, поддерживаемую `Active Record`, а значит, не требует для работы дополнительных инфраструктурных компонентов. Кроме того, в числе её многочисленных возможностей есть запуск задач по расписанию. И хотя этот функционал не является частью `Active Job`, его наличие в `Solid Queue` – важный шаг на пути к стандартизации работы с задачами по расписанию.

Сердце веб-приложения – база данных

Классическое веб-приложение можно представить как интерфейс для доступа к данным. Разрабатываете ли вы блог-платформу, интернет-магазин или платформу для онлайн-обучения, большинство действий пользователя так или иначе связаны с чтением или записью информации. Конечно, есть классы веб-приложений, которые не работают с данными, например прокси-серверы, но для их написания вы вряд ли будете использовать `Ruby on Rails`.

Данные зачастую являются главной ценностью вашего продукта или сервиса. Чтобы понять, а так ли это, представьте, что вы случайно удалили базу данных, а магнитные плёнки с резервными копиями погрызли крысы – будете ли после такого спать спокойно?

База данных, как правило, является и самым нагруженным компонентом вашего приложения. А так как практически все операции взаимодействуют с данными, от состояния и здоровья БД зависит и производительность всего приложения.

Таким образом, при проектировании приложения нам нужно всегда держать в уме возможную нагрузку на базу данных.

Влияние абстракций на производительность базы данных

В начале данной главы мы обсудили, что одной из главных задач абстракции является инкапсуляция деталей реализации: пользователь не должен знать, что происходит внутри предоставленного ему метода API. Рассмотрим следующий пример:

```
class User
  def self.create(name:)
    DB.exec "INSERT INTO users (name) values (%)", name
  end
end
names = %w[lacey josh]
names.each { |name| User.create(name:) }
```

Класс User является абстракцией над базой данных. Мы добавили простой интерфейс для создания пользователей, который планируем использовать везде в приложении.

К сожалению, чрезмерное использование этой абстракции может привести к повышенной нагрузке на базу данных: как только мы захотим создать N пользователей, нам придётся выполнить N запросов. Если бы мы не использовали никаких абстракций, мы бы написали SQL-выражение («INSERT INTO...») для массового создания записей – гораздо более производительный вариант, не так ли?

Это лишь небольшой пример, демонстрирующий следующее утверждение: **степень сокрытия деталей реализации должна выбираться с учётом возможных сценариев и стоимости (с точки зрения производительности и безопасности) использования.** Проектирование абстракций и интерфейсов должно в том числе ограждать пользователей от выстрелов себе в ногу.

Одним из распространённых источников проблем производительности в контексте баз данных являются *предметно-ориентированные языки* («Domain-specific language», или DSL) для формирования запросов. Использование DSL бывает очень удобным, но за такое удобство часто приходится платить.

Рассмотрим на примере из реальной жизни: использование библиотеки CanCanCan (<https://github.com/CanCanCommunity/cancancan>). Данная библиотека позволяет описывать права доступа с помощью *симпатичного* DSL. Вот небольшой кусочек файла с описанием прав:

```
can :read, Article do |article|
  article.published_at <= Time.now
end
```

Данное правило говорит, что пользователям для чтения доступны только уже опубликованные статьи. Далее, для отображения доступных пользова-

тею статей на странице мы используем следующее выражение: `Article.all.accessible_by(user)`. Как вы думаете, какой или какие запросы в базу данных будут выполнены? Возможно, вы предположили, что будет выполнен один запрос, `"SELECT * FROM articles WHERE published_at < now()"`, – именно так вы бы и сделали, да? Но хитрая библиотека с красивым DSL сделает по-другому: будут выгружены все записи из базы данных, а фильтрация произойдёт уже на стороне Ruby. Разницы с точки зрения результата никакой, но нагрузка на систему может разительно отличаться (расход памяти на инициализацию *лишних* объектов и циклов CPU на выполнение Ruby-кода). К счастью, `CanCanCan` позволяет вам добавить специальную инструкцию в DSL для оптимизации данного сценария:

```
can :read, Article, "published_at < now()" do |article|
  article.published_at <= Time.now
end
```

Да, исходная проблема решена, но появилась новая: теперь мы имеем дело с **протекающей абстракцией**. Сейчас наш файл конфигурации прав, использующий специальный DSL содержит вкрапления SQL – деталей реализации слоя хранения данных. В данном случае это необходимое зло, так называемый «костыль»; но наличие этого костыля указывает на то, что выбранный уровень абстракции оказался не самым удачным.

При проектировании абстракций мы всегда должны думать о потенциальном воздействии на производительность и учитывать его сразу же, чтобы избежать протекания абстракции в будущем.

Абстракции на уровне базы данных

Абстракции не обязательно существуют в коде вашего приложения. В некоторых случаях использование абстракций в самой базе данных бывает оправдано.

Основным аргументом в пользу внедрения абстракций на уровне базы данных, конечно, является производительность. Другой причиной может быть стремление к максимальной согласованности данных: база данных является главным источником правды, а ещё базы (по крайней мере реляционные) очень хороши в гарантировании согласованности данных. Всё это может привести вас на мысль о том, чтобы часть логики перенести из приложения в базу.

Но почему часть? Раз базы данных такие классные, почему бы не реализовать всю логику через хранимые процедуры, функции, триггеры? Производительность приложения не единственная, а иногда и не главная характеристика приложения. Зачастую гораздо важнее продуктивность команды, которая достигается использованием продуманной архитектуры кода и эффективных инструментов разработки. Вы ведь не случайно сейчас читаете книгу по Ruby on Rails?

Рассмотрим пример вынесения части логики на уровень базы данных, который значительно улучшает производительность без потери продуктивности.

Часто в веб-приложениях перед нами стоит задача отслеживания изменений, сделанных пользователями в данных (задача аудита). Мы можем реализовать данный функционал в Rails, используя колбэки («callback») в моделях (`after_create` и т. д.). Так, например, работает популярная библиотека Paper Trail (https://github.com/paper-trail-gem/paper_trail). Данный подход прост в использовании, но оказывает заметную нагрузку на производительность, а также не даёт гарантий согласованности данных (есть ситуации, в которых Active Record не вызывает колбэки). Мы могли бы подойти к решению этой задачи с другой стороны и использовать триггеры в базе данных: полная согласованность гарантирована, производительность на порядки выше. Да, написание и поддержка триггеров и хранимых процедур несравнимо сложнее пары строчек кода на Ruby. К счастью, есть готовые решения (например, библиотека Logidze), которые предоставляют интерфейс, сравнимый с решениями на уровне Active Record.

Библиотека: Logidze

Logidze (<https://github.com/palkan/logidze>) предоставляет набор расширений для базы данных (PostgreSQL и MySQL), а также Ruby API для работы с логами изменений отдельных записей. Логи хранятся в базе данных в компактном формате (инкрементально), позволяя не только отслеживать изменения записей, но и откатывать их состояние к определённой точке назад во времени.

Другой потенциальный кандидат на миграции в базу данных – это функционал «мягкого» удаления («soft deletion»). «Мягкое» удаление подразумевает пометку записи как удалённой вместо её реального удаления из таблицы. Этот подход часто используется для реализации функционала отмены предыдущего действия («undo»), а также для целей аудита.

Наконец, мы можем добавлять абстракции на уровне базы данных исключительно ради согласованности данных. Например, в PostgreSQL вы можете создавать доменные и составные типы данных. Для удобной работы с ними в Rails можно использовать библиотеку `pg_trunk` (https://github.com/nepalez/pg_trunk).

В общем случае использование абстракции на уровне базы данных имеет смысл, если данная абстракция выступает в роли *промежуточного преобразователя данных* («data middleware»), то есть рассматривает данные в изоляции и без привязки к бизнес-логике приложения. Технически такая изоляция означает, что подобная абстракция подключается и настраивается единожды, и не требует постоянной поддержки. Я специально использую терминологию, схожую с промежуточными обработчиками в Rack: они связаны концептуально через рассмотрение через чисто утилитарную функцию работы с данными.

Кроме того, мне хотелось закольцевать эту главу и вернуться наверх, к HTTP, ведь именно так и заканчивается путешествие «клика» – пройдя запросом до центра, *сердца*, он возвращается *наверх* ответом.

Итоги

В этой главе вы узнали об основных функциях и компонентах веб-приложений. Вы узнали об уровнях абстракции в приложениях Rails и о том, как они соотносятся с веб-природой Rails и его философией MVC. Вы узнали о концепциях единицы работы и контекста выполнения и их связи с входными уровнями абстракции. Вы также узнали о возможных компромиссах абстракций в сочетании с производительностью приложения и, в частности, с базой данных. В этой главе были продемонстрированы основные идеи, лежащие в основе многоуровневой архитектуры программного обеспечения, к которым мы будем часто обращаться на протяжении всей книги.

В следующей главе вы углубитесь в M-часть архитектуры MVC и узнаете о дизайнерских идеях, на которых построен Active Record – самый главный строительный блок фреймворка Ruby on Rails.

Проверь себя

1. Какие абстракции из предоставляемых Rails (контроллеры, модели, представления) соответствуют требованиям *хорошей* абстракции, сформулированным в данной главе?
2. Сколько слоёв абстракции необходимо иметь в Rails-приложении?
3. Влияет ли число слоёв абстракции на производительность Rails-приложения?
4. Какую общую проблему решают фоновые задачи?
5. Что такое протекающая абстракция?

Упражнение

Мы узнали, что для обработки веб-запроса Rails-приложение совершает тысячи вызовов методов и создаёт тысячи объектов в памяти Ruby. Как изменятся полученные значения, если мы пропустим цепочку промежуточных обработчиков в Rack и оставим только маршрутизацию (`Rails.application.routes.call(request)`)? А если рассмотреть лишь вызов контроллера (например, `PostsController.action(:index).call(request)`)? Используйте библиотеку `trace_location` и метод `GC.stats` для проведения замеров.