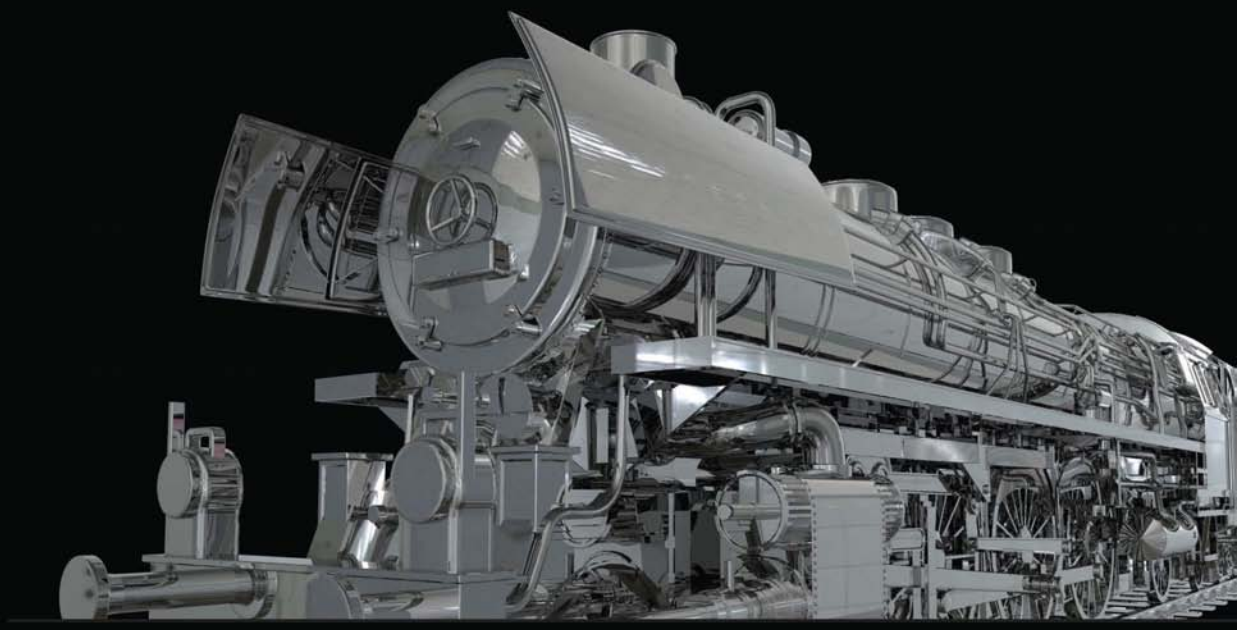


Владимир Дементьев

Проектирование приложений **Ruby on Rails** слой за слоем

ДМК Пресс
Издательство

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 / пер. с англ. В. М. Дементьева. – М.: ДМК Пресс, 2024. – 300 с.: ил.

ISBN 978-5-93700-260-0

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. 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 – (9781801813785)’.

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 978-1-80181-378-5 (англ.)
ISBN 978-5-93700-260-0 (рус.)

© 2023 Packt Publishing
© Перевод, оформление, издание, ДМК Пресс, 2024

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 31
- Абстракции на уровне базы данных 33

Итоги 34

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 46
- 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 54

В поисках всемогущества 55

Итоги 57

Проверь себя 57

Упражнение 57

Глава 3. Больше адаптеров, меньше связи с реализацией 58
Active Job как универсальный интерфейс очереди задач 58
Адаптеризация очередей 60
Сериализуй это 62
Адаптеры и плагины в Active Storage 66
Адаптеры и плагины 68
Адаптеры в вашем коде 70
Итоги 73
Проверь себя 73
Упражнение 73

Глава 4. Антипаттерны в Rails? 74
Колбэки, колбэки повсюду 74
Колбэки под контролем (в контроллерах) 75
Колбэки Active Record выходят из-под контроля 79
Озабоченность консёрнами в Rails 86
Разделяем поведение, а не код 88
Консёрны остаются модулями со всеми их недостатками 91
Композиция объектов 92
О глобальном и текущем состоянии 97
Текущее «всё подряд» 97
Итоги 102
Проверь себя 103
Упражнение 103

Глава 5. Когда абстракций Rails уже недостаточно 104
Проклятие толстых (тонких) контроллеров и тонких (толстых) моделей 104
От толстых контроллеров к толстым моделям 105
Пример толстого контроллера 106
Рефакторинг в соответствии с принципом тонких контроллеров и толстых моделей 108
От толстых моделей к сервисам 110
Сервисы общего назначения и специализированные абстракции 114
Связь между многоуровневой архитектурой и слоями абстракции 116
Итоги 119
Проверь себя 119

Часть II. ВЫДЕЛЕНИЕ АБСТРАКЦИЙ ИЗ МОДЕЛЕЙ 120

Глава 6. Абстракции слоя данных 121
Использование объектов запросов для вынесения (сложных) запросов из моделей 122
Выделение объектов запросов 124
Скоупы и объекты запросов 128
Объекты запросов общего пользования и Arel 130
Место объектов запросов в многоуровневой архитектуре 135

Отделение моделей от хранилища данных с помощью репозиториев 136
Итоги 139
Проверь себя 140

Глава 7. Обработка пользовательского ввода за пределами моделей 141

Объекты форм: ближе к интерфейсу, дальше от схемы данных 142
Формы ввода и модели 142
Использование Active Model для абстракции объектов форм 151
Объекты фильтров, или Построение запросов на основе пользовательского ввода 163
Фильтрация в контроллерах 164
Перенос фильтрации на уровень модели 165
Выделение объектов фильтров 166
Сравнение объектов фильтров, объектов форм и объектов запросов 169
Итоги 170
Проверь себя 170
Упражнение 170

Глава 8. Выделение презентационной логики из моделей 171

Использование презентеров для отделения моделей от представлений 171
Оставьте хелперы библиотекам 173
Презентеры и декораторы 174
Презентеры как слой абстракции 179
Сериализаторы как презентеры для вашего API 183
Преобразование модели в JSON 184
Сериализаторы как презентеры для API 185
Итоги 188
Проверь себя 189

Часть III. СЛОИ АБСТРАКЦИЙ НА КАЖДЫЙ ДЕНЬ 190

Глава 9. Модели и слои авторизации 191

Авторизация, аутентификация и другие аспекты безопасности 191
Разница между аутентификацией и авторизацией 192
Линии обороны веб-приложения 193
Модели авторизации 194
Безмодельная авторизация 195
Классические модели авторизации 195
Обеспечение контроля доступа, или Необходимость абстракций авторизации 200
Внедрение политик 201
Формирование авторизационного слоя абстракции 202
Авторизация в шаблонах представления 207
Влияние авторизации на производительность 211
Проблема ***N***+1 авторизации на уровне представления 211

Авторизация на основе выгрузки данных ... 213
Итоги ... 214
Проверь себя ... 214
Упражнение ... 215

Глава 10. Формирование абстрактного слоя уведомлений ... 216

От Action Mailer к многоканальной связи с пользователем ... 216
Action Mailer в действии ... 217
Место рассыльщиков почты в многоуровневой архитектуре ... 218
Не почтой единой, или Добавление других каналов связи ... 220
Выделение абстрактного слоя для работы с уведомлениями ... 222
Самодельная абстракция ... 222
Использование сторонних библиотек для организации работы с уведомлениями ... 225
Моделирование пользовательских настроек уведомлений ... 232
Битовые атрибуты и объекты-значения ... 233
Хранилище настроек уведомлений ... 235
Использование отдельной таблицы для настроек уведомлений ... 236
Итоги ... 236
Проверь себя ... 237
Упражнения ... 237

Глава 11. HTML под контролем абстракций ... 238

V в MVC Rails: шаблоны и хелперы ... 238
Пользовательский интерфейс без программного интерфейса ... 240
Переиспользование и дизайн-системы ... 245
Компонентный подход ... 248
Превращаем фрагменты и хелперы в компоненты ... 248
Компоненты интерфейса как слой абстракции ... 251
Компоненты интерфейса без HTML ... 256
Компоненты как связующее звено между командами ... 257
Итоги ... 257
Проверь себя ... 258

Глава 12. Конфигурация как первоклассная сущность приложения ... 259

Виды настроек и источников данных конфигурации ... 259
Файлы, секреты, зашифрованные хранилища и многое другое ... 260
Настройки и секреты ... 263
Окружения приложения и провайдеры данных ... 265
Многоуровневая архитектура и конфигурация ... 265
Использование объектов предметной области для упрощения настроек приложения ... 266
Отделение кода приложения от источников конфигурации ... 266
Освобождаем кодовую базу от зависимости от окружения ... 270

Использование классов конфигурации ... 272
Итоги ... 276
Проверь себя ... 277
Упражнение ... 277

Глава 13. Сквозь слои и дальше ... 278

Разнообразие инфраструктурного уровня в Rails ... 278
Инфраструктурные абстракции и реализации ... 279
Сквозь уровни: логирование и мониторинг ... 280
Логирование ... 280
Отслеживание исключений ... 284
Инструментация ... 285
Вынесение низкоуровневой реализации в отдельный сервис ... 290
Отпочковываем веб-сокеты от Action Cable с помощью AnyCable ... 290
Обработка изображений на лету, но не в Rails ... 291
Итоги ... 293
Проверь себя ... 294

Предметный указатель ... 295

Библиотеки и паттерны ... 299

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.

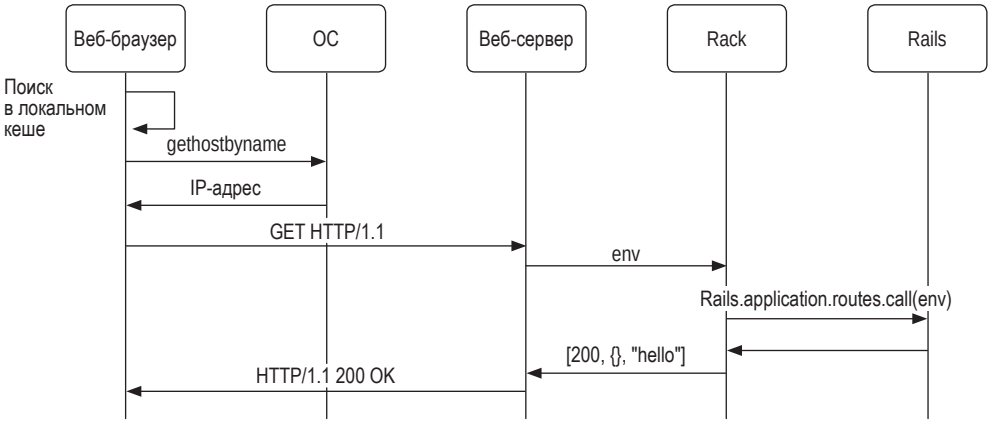


Рис. 1.1 ❖ Упрощённая схема путешествия «клика»

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-

тов: 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 1.7.0
[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 в 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.

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.

---

[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`.

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 числа? Хотя целью данной книги является демонстрация того, как грамотное внедрение и выделение абстракций помогает 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 C – 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.

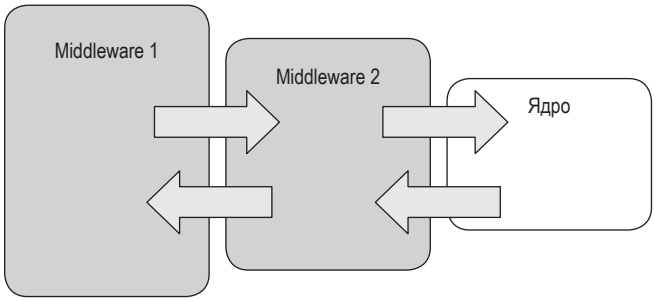


Рис. 1.2 ❖ Схема паттерна Middleware

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: %i[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?

C 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 единовременно 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 GVL[1] (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.

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,

[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.

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 выносится 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: