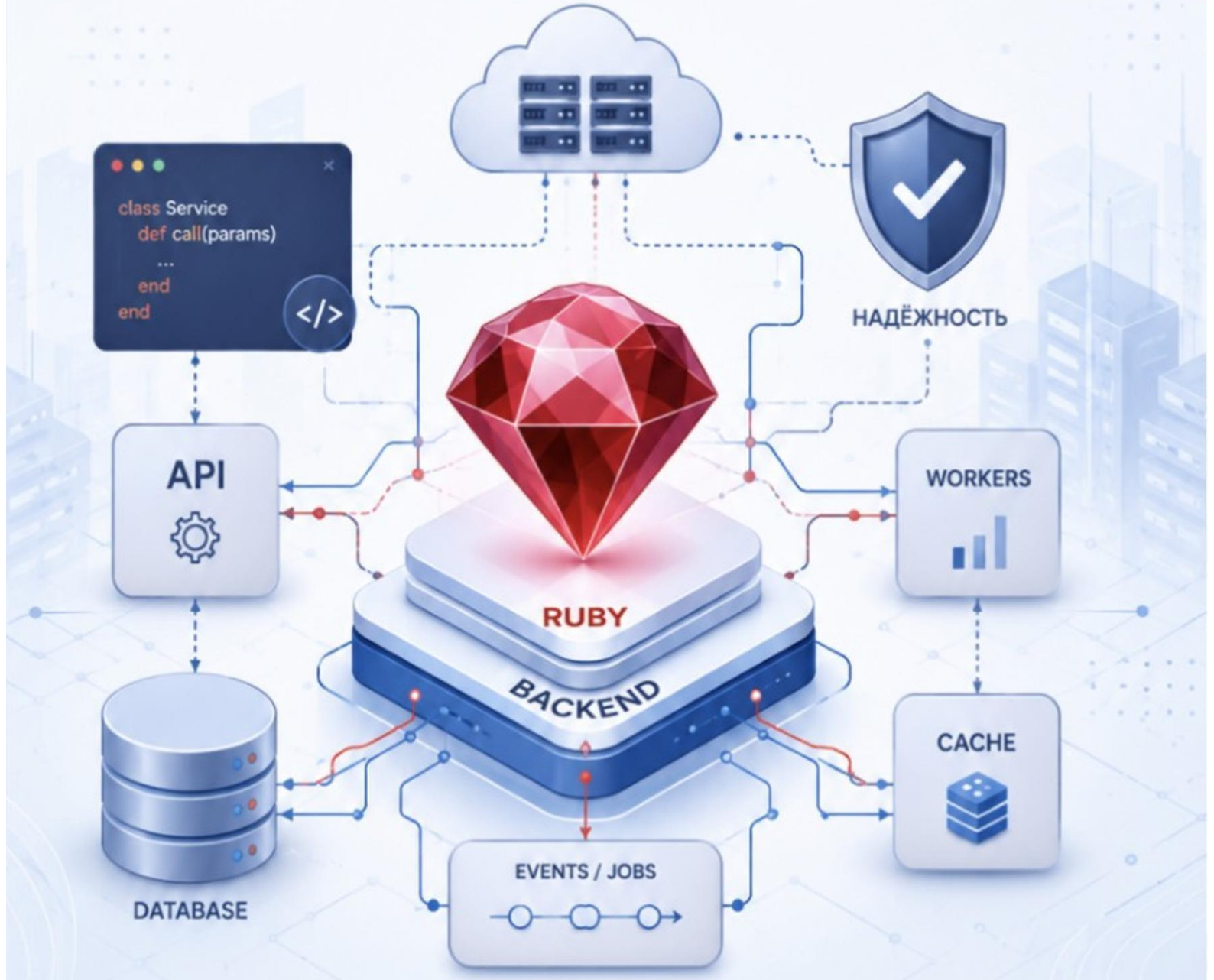
Ruby Backend Engineering: как строят системы, которым доверяют



БЕЗОПАСНОСТЬ
защита данных и доступов

ОТКАЗОУСТОЙЧИВОСТЬ
стабильность даже под нагрузкой

МАСШТАБИРУЕМОСТЬ
рост без потери производительности

ДОВЕРИЕ
проверено временем и практикой

2025

Евгений Сташковьян

Ruby Backend Engineering: как строят системы, которым доверяют

«Автор»
2026

Сташковьян Е. С.

Ruby Backend Engineering: как строят системы, которым доверяют / Е. С. Сташковьян — «Автор», 2026

Книга посвящена практическим и архитектурным аспектам backend-разработки на Ruby on Rails. Автор рассматривает Rails не как инструмент быстрого создания CRUD-приложений, а как зрелую инженерную платформу для построения сложных, нагруженных и юридически значимых систем. В центре внимания находятся проектирование доменной модели, API-first архитектура, тестирование, работа с PostgreSQL, ActiveRecord, очередями, Redis, Elasticsearch, файлами, Docker, AWS-инфраструктурой, наблюдаемостью, безопасностью и модернизацией монолита. Особую ценность книге придаёт опора на реальный производственный опыт автора: участие в разработке федеральных цифровых платформ, систем электронных торгов, закупок и арбитражных процедур. Издание адресовано разработчикам, уже знакомым с Ruby on Rails, стремящимся перейти от написания работающего кода к созданию надёжных production-систем. Книга формирует инженерное мышление, необходимое для senior-уровня, архитектурной ответственности.

Содержание

Евгений Сташковьян
Ruby Backend Engineering: как строят системы, которым доверяют

Ruby Backend Engineering: как строят системы, которым доверяют

2025

ПРЕДИСЛОВИЕ

Я пишу эту книгу из конкретного места — из опыта разработчика, который прошёл путь от первых HTML-страниц в двенадцать лет в Бийске Алтайского края до руководителя группы разработки на федеральных цифровых системах России. Это не биографическая деталь ради биографии. Это важно потому, что каждая глава книги основана на реальных задачах, ошибках и решениях, не на синтетических примерах из учебников.

Я работал над системами, которым люди доверяли кое-что серьёзное. Ошибка в статусе торговой процедуры здесь — это не баг, который исправят в следующем спринте. Это юридические последствия. Центральная цифровая платформа: миллионы заявок, вся страна, данные о жизненно важной инфраструктуре. Система для арбитражных процедур, где каждый документ имеет юридическую силу.

Именно работа с такими системами сформировала инженерное мышление, которое эта книга пытается передать. Не «как написать код, который работает», а «как строить системы, которым доверяют».

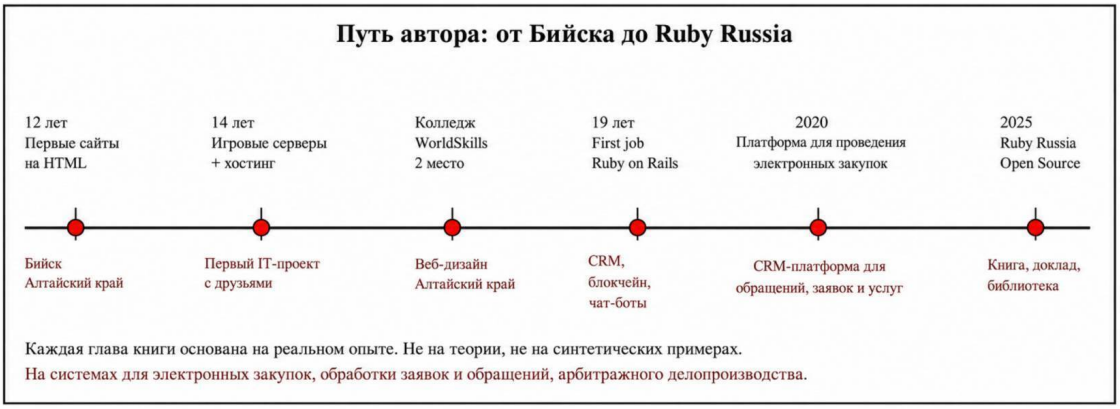
О чём эта книга

Книга посвящена backend-инженерии на Ruby on Rails, не как языку синтаксиса или фреймворку для CRUD, а как инженерной платформе для построения сложных, нагруженных, юридически и финансово значимых систем. Двадцать три главы охватывают путь от устройства языка Ruby до open source и публичной экспертизы.

Книга состоит из четырёх частей. Первая часть закладывает фундамент: Ruby как язык инженерного мышления, Rails за пределами CRUD, доменная модель в монолите, API-first архитектура, тестирование как архитектурный инструмент. Вторая часть посвящена данным и производительности: PostgreSQL в production, ActiveRecord и его подводные камни, очереди и асинхронность, Redis, Elasticsearch, управление файлами. Третья часть — о production: Docker, AWS-инфраструктура, наблюдаемость, инциденты, безопасность. Четвёртая часть — о зрелой инженерии: модернизация монолита, архитектурная декомпозиция, критические бизнес-операции, инженерные стандарты, наставничество, open source.

Каждую часть можно читать независимо, но книга выстроена как путь: от языка через данные к production и к инженерной зрелости.

Откуда взялась эта книга



Интерес к программированию появился рано. В двенадцать лет я начал делать первые сайты. В четырнадцать вместе с другом мы подняли игровые серверы на домашних компьютерах, создали сайт и продавали услуги хостинга. После девятого класса я поступил в Бийский государственный колледж на специальность «Информационные системы» и занял второе место на региональном чемпионате WorldSkills Russia в компетенции «Веб-дизайн».

В девятнадцать лет я устроился на первую работу как Ruby on Rails разработчик. Первые проекты: CRM-система для инвестиционно-финансовой компании, конструктор чат-ботов для бизнеса, блокчейн-приложения для хранения данных. Четыре года реального кода, реальных пользователей, реальных ошибок.

В 2020 году я перешёл в компанию, которая управляет системой электронных торгов и закупок. За шесть лет я вырос от разработчика до руководителя группы, каждый год получал премии по итогам оценки, получил благодарность от генерального директора. За это время я участвовал в модернизации платформы, переводил монолит с архитектуры Rails back + Rails front на Rails API + Nuxt, оптимизировал критические сервисы торгов, разработал систему управления жизненным циклом файлов — решение, которое снизило инфраструктурные расходы без ухудшения доступности для пользователей.

Параллельно я строил систему обучения: подготовил корпоративный курс по Ruby, проводил менторство, помог нескольким разработчикам вырасти до уровня middle+. Из этого опыта и родилась эта книга.

Для кого написана эта книга

Книга написана для разработчиков, которые уже знают Ruby on Rails и хотят двигаться дальше: понять, как строятся системы, а не просто как писать код. Для тех, кто хочет перейти от «работает у меня» к «надёжно работает в production». Для тех, кто думает о переходе в senior-позицию и хочет понять, чем отличается системное мышление от навыка написания кода.

Книга также будет полезна тем, кто уже работает на уровне senior и хочет структурировать свой опыт. Я обнаружил, что многое из того, что я делал интуитивно годами, стало гораздо яснее, когда я попробовал это сформулировать.

Это не учебник по синтаксису Ruby. Предполагается, что читатель умеет писать на Ruby и Rails. Это книга о том, как думать об инженерных задачах.

Как читать эту книгу

Если вы изучаете тему последовательно — читайте главы по порядку. Первая часть закладывает понятийный фундамент, без которого некоторые главы второй и третьей части будут менее понятны.

Если вы решаете конкретную задачу прямо сейчас — идите к нужной главе напрямую. Каждая глава самодостаточна в том смысле, что её можно читать отдельно и получить из неё практическую ценность.

Примеры кода в книге — реальные или основанные на реальных решениях. Я намеренно не упрощал их до «академических» примеров. Production-код сложнее учебного — и именно это делает его полезным.

Часть I.
Ruby и Rails как основа зрелого backend

ГЛАВА 1. RUBY КАК ЯЗЫК ИНЖЕНЕРНОГО МЫШЛЕНИЯ

Когда я впервые открыл редактор и написал первую строку на Ruby, мне было девятнадцать лет. За плечами был колледж, региональный чемпионат WorldSkills, пара самодельных сайтов и горячее желание понять, как устроены системы, которым люди доверяют деньги, данные и юридически значимые решения. Тогда я не думал о философии языка. Я просто хотел, чтобы код работал.

Но со временем понимаешь: выбор языка программирования — это не только вопрос синтаксиса или скорости выполнения. Это вопрос того, как ты думаешь о задаче. Ruby формирует способ рассуждать об объектах, поведении, ответственности и границах. И именно поэтому я начинаю книгу не с фреймворка, не с базы данных и не с архитектурных паттернов, а с самого языка.

Эта глава — попытка объяснить, почему Ruby располагает к определённому инженерному мышлению. Не потому что он лучший язык (такой постановки вопроса я принципиально избегаю), а потому что его устройство подталкивает разработчика к конкретным решениям. Читая её, вы, возможно, узнаете подходы, которые уже используете интуитивно. Или откроете для себя инструменты, которые давно лежали рядом, но казались лишними.

1.1. Почему Ruby — это не только синтаксис, а способ моделировать поведение

Среди разработчиков есть расхожее мнение: Ruby выбирают за красивый синтаксис, а потом страдают от производительности. Это мнение неточно по обеим частям. Синтаксис Ruby действительно выразительный, но это следствие, а не цель. Цель — позволить программисту описывать поведение системы так, как он описывает его в голове.

Мац (Юкихиро Мацумото, создатель Ruby) в своих интервью неоднократно говорил о том, что проектировал язык для людей, а не для машин. Эта фраза кажется маркетинговой, пока не начнёшь работать с языками, спроектированными в обратную сторону. Когда вы пишете на C или Java, вы часто думаете о том, как объяснить компилятору, что вы имеете в виду. В Ruby большую часть времени вы просто описываете, что должно происходить.

Возьмём простой пример. Представьте, что вам нужно найти все активные заявки пользователя, созданные за последние тридцать дней. На большинстве языков вы напишете запрос, пройдётесь по результату циклом и применимте условие. В Ruby запись выглядит иначе:

```
Application.where(user: user).active.created_after(30.days.ago)
```

Это не магия и не синтаксический сахар ради красоты. Это отражение того, как работает цепочка областей видимости (scopes) в ActiveRecord, построенная на принципах композиции объектов. Каждый вызов возвращает объект того же типа с дополненными условиями. Конечный запрос строится лениво, только когда данные действительно нужны.

Этот паттерн формирует привычку: описывать поведение через интерфейс объекта, а не через набор инструкций. Это не просто удобство. Когда кодовая база растёт, умение отделить «что» от «как» становится принципиально важным для поддерживаемости системы.

На проекте цифровой платформы для тендерных процедур, эта привычка спасала нас регулярно. Система обрабатывает тысячи торговых процедур, каждая из которых имеет сложный жизненный цикл: создание, подача заявок, проведение торгов, подведение итогов, подписание договора. Вся эта логика строилась на объектах с чётко определённым поведением, и это позволяло добавлять новые типы процедур, не трогая уже работающий код.

Ruby позволяет строить объекты так, что каждый из них несёт ответственность за свою область. Это звучит банально — принцип единственной ответственности описан в каждой второй книге по проектированию. Но именно в Ruby этот принцип встроен в идиомы языка. Блоки, лямбды, методы как объекты первого класса — всё это инструменты для создания систем, где поведение можно передавать, комбинировать и переопределять без перестройки архитектуры.

Посмотрите на схему ниже. Она наглядно показывает разницу между описанием задачи через инструкции и через объектный интерфейс.

Ruby — это язык, в котором идиоматичный код описывает намерение, а не механику. Это не означает, что он скрывает от вас устройство системы. Это означает, что язык не заставляет вас описывать механику, когда вам нужно описать намерение. Для большого проекта разница накапливается в сотнях строк кода, которые либо читаются как объяснение задачи, либо требуют дешифровки.

Переход от «написать код, который работает» к «написать код, который описывает поведение системы» — это сдвиг, который происходит не сразу. Я сам прошёл через него, работая над первыми проектами: CRM-системой для инвестиционно-финансовой компании, конструктором чат-ботов, проектами для хранения данных на базе блокчейн. Сейчас, когда я обучаю стажёров и провожу менторство, именно это различие пытаюсь донести в первую очередь.

1.2. Объектная модель Ruby: классы, модули, примеси и границы ответственности

В Ruby всё является объектом. Это утверждение слышат на первом занятии по языку, кивают и идут дальше. Но стоит разобраться, что именно это означает на практике, потому что последствия для архитектуры системы весьма существенны.

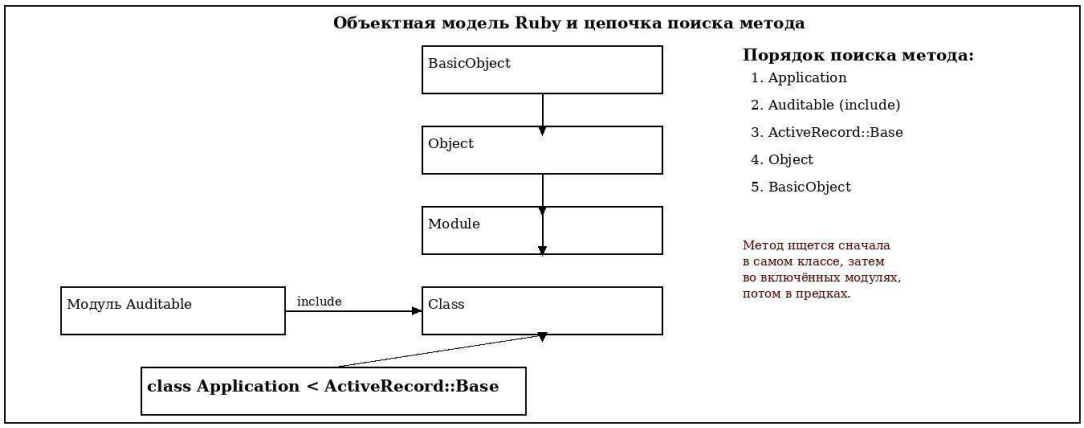
Число 42 в Ruby — это объект класса Integer. Строка «hello» — объект класса String. Класс MyService — тоже объект, только класса Class. Это означает, что класс можно передать в метод как аргумент, сохранить в переменную, вернуть из другого метода. Это не академическая деталь: паттерны, которые в других языках требуют фабрик, шаблонов или отражения, в Ruby реализуются напрямую.

Что из этого следует для реальной системы? Разные типы заявок обрабатываются по-разному: физические лица, юридические лица, заявки на догазификацию. Каждый тип имеет свою логику проверки данных, свой набор документов, свой процесс согласования. Один из подходов — большой блок условий по типу. Другой — передавать класс-обработчик как параметр и вызывать его методы полиморфно. Ruby не просто поддерживает второй подход — он делает его естественным.

Теперь о модулях. Модуль в Ruby — это пространство имён и инструмент для подмешивания (mixin) поведения в класс. Разница между модулем и классом проста: от модуля нельзя создать экземпляр, и у него нет единственного предка. Зато его можно включить в любой класс с помощью include, и все методы модуля станут методами экземпляра класса.

Это мощный инструмент для совместного использования поведения без наследования. Классическая проблема объектно-ориентированного проектирования: что делать, когда поведение нужно разделить между классами, которые не имеют смысла в одной иерархии? В Ruby для этого существуют примеси.

Пример из реальной работы: на платформе системы электронных торгов и закупок у нас было несколько сущностей (процедуры, заявки, договоры), каждая из которых должна была поддерживать аудит изменений. Логика аудита одинакова: при каждом изменении фиксировать, что изменилось, кто изменил, когда. Вместо того чтобы дублировать этот код или выстраивать искусственную иерархию, мы создали модуль Auditable и включили его в нужные классы. Чистое, тестируемое решение, которое не усложняет иерархию объектов.



Объектная модель Ruby подталкивает к тому, чтобы каждая единица кода (класс, модуль, метод) делала что-то одно и делала это хорошо. Но это подталкивание, а не принуждение. Можно написать класс на тысячу строк, который делает всё что угодно — Ruby не запретит. Именно поэтому важно понимать, какие конструкции языка поддерживают правильное разделение, и использовать их осознанно.

Несколько ориентиров, которые я выработал за годы работы с большими приложениями. Если метод делает больше одного логического шага — это сигнал: либо метод слишком боль-

шой, либо он знает о деталях, которые не должен знать. Если класс включает больше двух-трёх модулей и каждый из них большой — это почти наверняка признак того, что класс взял на себя слишком много. Если вы не можете в одном предложении описать ответственность класса — класс либо делает несколько вещей, либо его назначение размыто.

Понимание объектной модели — это фундамент. Но фундамент имеет смысл только тогда, когда на нём что-то строится. Следующий вопрос: как отличить Ruby-код, написанный с пониманием этой модели, от кода «на любом языке с Ruby-синтаксисом»?

1.3. Идиоматичный Ruby против кода на любом языке с Ruby-синтаксисом

Когда команда нанимает нового разработчика с опытом в других языках, и тот начинает писать на Ruby, первое время код выглядит странно. Технически правильно, синтаксис верный, тесты проходят — но что-то не так. Обычно это называют «неидиоматичным кодом».

Что такое идиоматичность? Это использование возможностей языка так, как их задумал автор и как принято в сообществе. Неидиоматичный код — это Ruby-синтаксис поверх мышления в другой парадигме. Такой код работает, но его сложнее читать тому, кто знает Ruby, и он обычно упускает возможности языка, которые сделали бы код короче и выразительнее.

Конкретный пример. Разработчик с опытом Java пишет метод для получения имени пользователя или значения по умолчанию:

Разница не только в длине. Первый вариант говорит: «если пользователь не пустой, вернуть имя, иначе вернуть гость». Второй говорит: «имя пользователя, если пользователь есть, иначе гость». Это чуть ближе к тому, как мы думаем о задаче, а не к тому, как процессор её выполняет.

На большом проекте это накапливается. Кодовая база платформы для проведения электронных закупок насчитывает сотни тысяч строк. Если каждый разработчик пишет в своём стиле, новый человек (или тот же разработчик через полгода) тратит значительно больше времени на понимание чужого кода. Единый идиоматичный стиль — это форма документации.

Важная оговорка: идиоматичность не самоцель и не религия. Иногда явный цикл читается лучше, особенно если тело цикла сложное или выполняет несколько действий. Всегда нужно спрашивать: какая запись лучше всего передаёт намерение в конкретном контексте? Если идиоматичная запись требует от читателя знания пяти специфичных методов Ruby, возможно, более явный вариант предпочтительнее.

Идиоматичный код — это не конечная цель, а условие, при котором следующая тема имеет смысл. Метапрограммирование строится именно на глубоком понимании объектной модели и идиом языка. Без этого фундамента оно становится источником магии, а не силы.

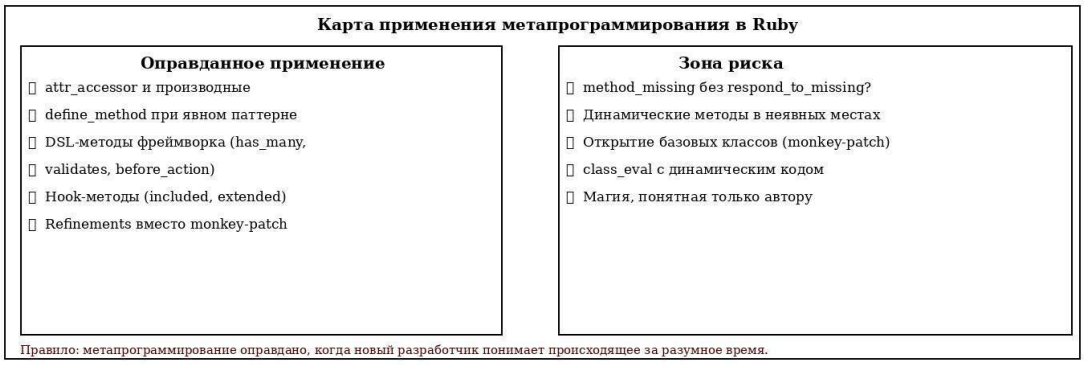
1.4. Метапрограммирование: где оно усиливает систему, а где разрушает читаемость

Метапрограммирование — одна из тех тем, которые вызывают диаметрально противоположные реакции у разработчиков. Одни смотрят на возможности define_method, method_missing, class_eval как на то, что делает Ruby уникальным. Другие считают это источником нечитаемого кода и инцидентов в production. Правы и те и другие. Вопрос в том, где граница.

Что такое метапрограммирование в Ruby? Это способность программы изучать и изменять свою структуру во время выполнения. Классы можно открывать и добавлять методы. Методы можно определять динамически по имени. Можно перехватить вызов несуществующего метода и обработать его. Это не экзотика: именно на этих возможностях построены некоторые из самых часто используемых частей Rails.

Рассмотрим ActiveRecord. Когда вы определяете класс User < ApplicationRecord, вы получаете методы find, where, create, update, destroy — и при этом не пишете ни строки их реализации. ActiveRecord читает схему базы данных и динамически создаёт методы-атрибуты для каждого столбца. Это метапрограммирование на службе у разработчика: удобный интерфейс без дублирования кода для каждой таблицы.

Теперь о том, где метапрограммирование разрушает читаемость. Главная опасность — когда оно скрывает связи, которые должны быть видны. С одной стороны, это убирает дублирование. С другой — попробуйте найти метод render_details в кодовой базе с помощью обычного поиска. Вы не найдёте его. Редактор не сможет предложить автодополнение. Если в логике метода есть ошибка, трассировка стека покажет его имя, но перейти к «определению» стандартными средствами будет невозможно.



Правило, которое я выработал: метапрограммирование оправдано, когда оно устраняет значительное дублирование и когда его присутствие явно обозначено в архитектуре системы. Хороший признак правильного применения: разработчик, не знакомый с этим местом кода, может понять, что происходит, за разумное время. Плохой признак: «магия», которую нужно долго объяснять новому человеку в команде.

Отдельно стоит сказать о monkey-patching — открытии существующих классов и добавлении в них методов. В Rails это используется повсеместно: 5.days.ago, 'hello'.titleize, [1,2,3].sum работают именно потому, что Rails открывает классы Integer, String, Array и добавляет в них методы. Это удобно и делает Ruby-код выразительным. Но в прикладном коде

monkey-patching опасен: вы можете случайно затронуть поведение, которое используется в других частях системы. Безопасная альтернатива — Refinements, механизм, позволяющий расширять классы в ограниченной области видимости.

Понимание метапрограммирования — это понимание того, где возможности языка заканчиваются и начинается ответственность разработчика. Следующий вопрос, который из этого вырастает: как написать код, который можно поддерживать не только сегодня, но и через три года?

1.5. Как писать Ruby-код, который можно поддерживать годами

За шесть лет работы на одном проекте я видел код, написанный до меня, и код, написанный мной три года назад. Это разный опыт. Код, написанный до тебя, вызывает вопросы: почему именно так? Что здесь происходит? Можно ли это изменить? Свой старый код вызывает другие чувства: я помню контекст, но не всегда помню детали. И если код хорошо написан, контекст восстанавливается по нему самому.

Поддерживаемый код — это не красивый код и не короткий код. Это код, который объясняет свои намерения и делает очевидными свои ограничения. Несколько конкретных принципов, выработанных на практике.

Первый принцип: имена важнее комментариев. Если метод называется process_data, вы ничего не знаете о том, что он делает. Если он называется validate_and_persist_application, вы знаете многое. Хорошее имя устраняет необходимость в комментарии. Плохое имя требует комментария, который со временем устаревает, пока код меняется, а комментарий остаётся нетронутым.

Второй принцип: явное лучше неявного. Когда метод принимает хэш опций без явного описания, что именно в этом хэше ожидается, — это источник проблем. Когда метод принимает именованные параметры (keyword arguments) с явными именами — читатель видит контракт сразу.

Третий принцип: мутации должны быть очевидны. Ruby позволяет методам изменять объект на месте (bang-методы: sort! вместо sort, upcase! вместо upcase). Когда метод изменяет состояние объекта, это должно быть отражено либо в имени (persist!, complete!), либо в документации. Скрытые побочные эффекты — один из самых сложных источников ошибок в больших системах.

Четвёртый принцип, который я особенно ценю: fail fast. Если метод получает некорректные данные, он должен сразу сигнализировать об этом, а не пытаться продолжить работу с неверными предположениями. В системах, где ошибка означает некорректный статус торгов или неверно принятую заявку на газификацию, «продолжить работу как-нибудь» означает создать данные, которые потом невозможно восстановить без ручного вмешательства.

На практике это выражается в явных проверках в начале метода:

def submit_application(user, params) raise ArgumentError, 'Пользователь не передан' unless user raise InvalidStateError unless user.verified? raise DuplicateApplicationError if user.has_pending_application? # основная логика только здесь end

Пятый принцип, который часто недооценивают: думайте о контексте чтения, а не о контексте написания. Когда вы пишете код, у вас в голове полный контекст задачи. Когда кто-то другой (или вы сами через год) читает этот код, этого контекста нет. Хорошо написанный код должен воссоздать этот контекст через имена, структуру и, где необходимо, через коммента-

рии. Комментарий уместен не для объяснения того, что делает код, а для объяснения того, почему именно так.

1.6. Баланс выразительности, простоты и предсказуемости

Ruby соблазняет выразительностью. Язык богатый: у него много способов сделать одно и то же, много конструкций, которые «работают как по-английски», много синтаксических удобств. Это приятно при написании кода, но может создать проблемы в большой команде или долгоживущем проекте.

Покажу на примере. В Ruby есть несколько способов написать условие: if / unless, тернарный оператор, постфиксный if, конструкция case/when. Все они правильны. Но если в кодовой базе они используются вперемешку без ясного принципа, код становится труднее читать — не потому что конструкции сложны, а потому что читатель должен каждый раз переключаться между стилями.

Простота в Ruby — это не упрощение задачи. Это устранение лишних уровней абстракции там, где задача не требует их. Рефакторинг к простоте часто выглядит как удаление кода, а не добавление.

Предсказуемость — другой аспект. Код предсказуем, когда его поведение понятно из интерфейса без необходимости читать реализацию. Метод save_order должен сохранять заказ. Если он кроме этого отправляет уведомление, обновляет статистику и записывает в лог — это непредсказуемое поведение, которое рано или поздно приведёт к ошибке.

Именно из этого баланса — выразительность там, где она помогает, простота там, где можно, предсказуемость всегда — вырастает то, что называют хорошим Ruby-кодом. Это не правило из книги. Это культура, которая формируется в команде через ревью, обсуждения и общие примеры.

Ruby — язык, который предоставляет разработчику высокую степень свободы. Эта свобода работает на вас, если вы используете её осознанно, и против вас, если злоупотребляете ею. Баланс выразительности, простоты и предсказуемости — это навык, который приходит с опытом и рефлексией.

Разобрав объектную модель, идиомы и принципы поддерживаемого кода, можно сформулировать наблюдение, которое редко звучит явно: Ruby — это язык, в котором инженер

постоянно принимает решения о выразительности. В отличие от языков с более жёсткими ограничениями, Ruby не принуждает к единственному способу решить задачу. Это делает его мощным инструментом для опытных разработчиков и потенциально трудным для тех, кто ещё не выработал внутренние критерии выбора.

Именно поэтому изучение Ruby стоит начинать не с синтаксиса, а с вопроса: что я хочу выразить этим кодом? Когда разработчик может ответить на этот вопрос применительно к каждому методу и каждому классу — он пишет код, который понятен и через три года. Когда не может — пишет код, который работает сегодня и становится проблемой завтра.

Следующий уровень этого мышления — понять, как эти принципы применяются на уровне фреймворка. Rails предоставляет богатую экосистему инструментов, и большинство из них следуют той же логике: дать разработчику выразить намерение, а не механику. Но только при условии, что разработчик понимает, где заканчиваются возможности фреймворка и начинается ответственность за архитектурные решения.

ГЛАВА 2. RAILS ЗА ПРЕДЕЛАМИ CRUD

Когда я рассказываю коллегам из других технологических стеков, что работаю с Ruby on Rails, часто слышу одну и ту же реакцию: «а, это тот фреймворк для быстрого прототипирования». Иногда добавляют: «он же не подходит для серьёзных систем». Обе фразы объясняются просто: люди знакомы с Rails по его репутации начала 2000-х годов и по статьям, написанным людьми, которые использовали его ровно на том уровне, которым он известен — быстрое создание небольших приложений.

Реальность другая. За шесть лет работы на платформе для проведения электронных закупок я работал с Rails-приложением, которое обрабатывало тендеры и закупочные процедуры. Это система, где изменение статуса процедуры имеет юридические последствия, где одновременно могут работать тысячи участников торгов, и где ошибка в логике заявки может стоить компании контракта. Это не прототип.

Цель этой главы, показать, что Rails за пределами контроллеров и ActiveRecord — это богатая архитектурная экосистема. Не быстрый старт, а зрелая платформа для сложных бизнес-систем. Но только если вы понимаете, где проходят архитектурные границы и какие инструменты для их проведения у вас есть.

2.1. Почему Rails не заканчивается контроллерами и ActiveRecord

Стандартная структура Rails-приложения, которую видит разработчик в первые дни знакомства с фреймворком: папки app/controllers, app/models, app/views. Это MVC (Model-View-Controller) в явном виде. Фреймворк сделал всё, чтобы создать ресурс за несколько минут. И это отличное начало.

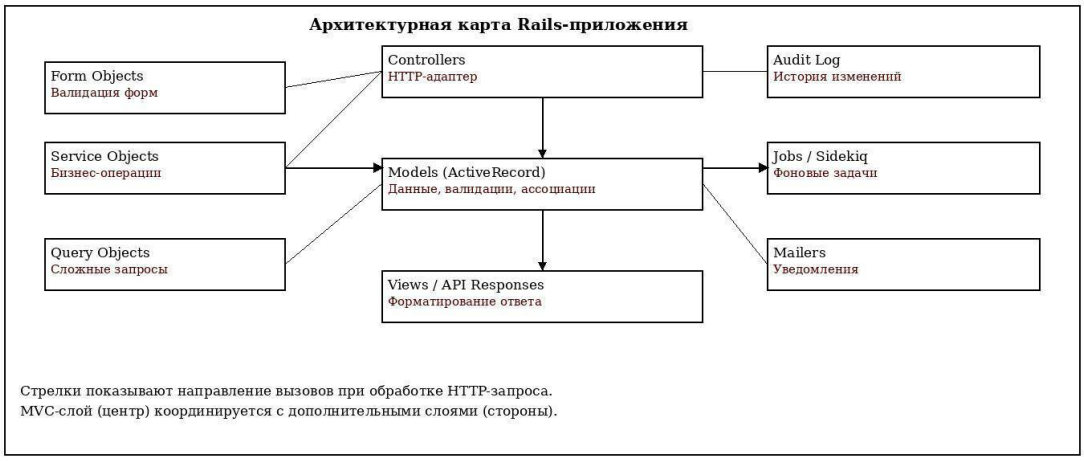
Проблема начинается тогда, когда разработчик продолжает использовать только эти три папки на протяжении всего роста приложения. Контроллеры начинают содержать бизнес-логику. Модели ActiveRecord обрастают методами, которые не имеют отношения к данным. Виды получают логику, которой там быть не должно. Система начинает «течь»: логика проникает не в те слои.

Контроллер в Rails отвечает за приём HTTP-запроса, извлечение параметров, вызов бизнес-логики и формирование ответа. Это три чётких задачи. Если в контроллере появляются многострочные условия, вызовы нескольких моделей, сложная подготовка данных для ответа — это сигнал о нарушении ответственности.

Аналогично с моделью. ActiveRecord-модель обеспечивает взаимодействие с базой данных: атрибуты, проверки данных, связи, области видимости. Это её зона ответственности. Когда модель начинает отправлять письма, интегрироваться с внешними API, выполнять сложные бизнес-правила — она перестаёт быть моделью данных и становится чем-то неопределённым.

Rails не запрещает это. Он даже немного поощряет через обратные вызовы (before_save, after_create и другие). Они кажутся удобными: в одном месте описано всё, что происходит при сохранении объекта. Но за этим удобством скрывается опасная связность: теперь вы не можете сохранить объект в тесте без того, чтобы не запустить цепочку побочных эффектов.

Что Rails предоставляет за пределами MVC? Начнём с того, что уже встроено в фреймворк, но часто игнорируется: jobs, mailers, channels, validators, formatters. Кроме этого, сообщество Rails выработало ряд паттернов, которые закрывают пробелы стандартной архитектуры.



Понимание того, что Rails — это платформа, а не только MVC-каркас, открывает путь к рассмотрению конкретных паттернов. Каждый из них решает конкретную проблему, которая возникает при росте приложения.

2.2. MVC в больших приложениях: где проходит реальная граница слоёв

В учебных примерах MVC выглядит просто: модель хранит данные, представление отображает, контроллер координирует. Но в реальном приложении вы быстро обнаруживаете, что многие задачи не укладываются в эту схему однозначно.

Возьмём типичную ситуацию: пользователь подаёт заявку на участие в торговой процедуре. Что происходит? Нужно проверить данные заявки (некоторые поля обязательны только в определённых типах процедур). Убедиться, что пользователь имеет право подавать заявку (роль, статус аккредитации). Сохранить заявку с правильным начальным статусом. Уведомить организатора торгов. Записать событие в аудит. Возможно, обновить счётчик в кэше.

Где это всё должно жить? Если засунуть в контроллер — получим контроллер на сто строк с семью вызовами разных моделей. Если в модель Application — получим модель, которая знает о ролях пользователей, уведомлениях, кэше и аудите. Ни то ни другое не масштабируется.

Реальная граница слоёв в большом Rails-приложении определяется вопросом: что вы тестируете и как легко это тестировать? Если тест контроллера требует создания сложного набора объектов и проверки побочных эффектов в базе — логика в неправильном месте. Если тест модели не может обойтись без подмены внешних сервисов — то же самое.

Отдельно скажу о представлениях и помощниках. В традиционном Rails с ERB-шаблонами они часто становятся ещё одним местом для бизнес-логики. Паттерн Presenter или Decorator решает эту проблему: создаётся объект-обёртка над моделью, который содержит методы форматирования и вычисляемые свойства для представления. Это позволяет держать шаблон простым, а модель — не знающей о том, как она отображается.

Когда речь идёт о реальной границе слоёв — это прежде всего тестируемость. Код хорошо разложен по слоям, если каждый слой можно протестировать изолированно, не поднимая весь стек. Это признак того, что зависимости между слоями разумны.

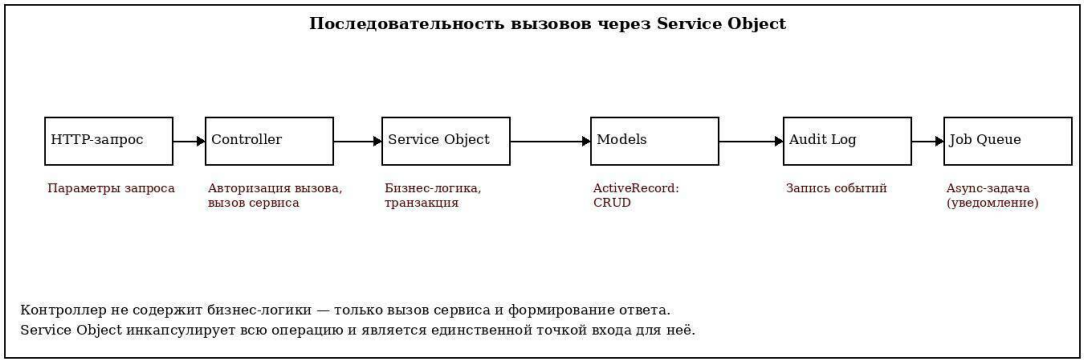
2.3. Service objects, form objects, query objects и другие способы разгрузить модель

Service object — пожалуй, самый распространённый паттерн в зрелых Rails-приложениях. Идея проста: если у вас есть бизнес-операция (зарегистрировать пользователя, обработать платёж, опубликовать объявление), она заслуживает собственного класса, который инкапсулирует всю логику этой операции.

Типичный service object в Rails выглядит так:

Обратите внимание на несколько вещей. Во-первых, один входной метод call. Это соглашение, которое делает service objects взаимозаменяемыми и предсказуемыми. Во-вторых, явные предусловия в виде bang-методов, которые выбрасывают исключение при нарушении. В-третьих, транзакция, которая гарантирует атомарность: либо всё произошло, либо ничего.

Этот паттерн я использовал для операций с торговыми процедурами. Создание процедуры, изменение её статуса, подведение итогов торгов — каждая из этих операций имела собственный service object. Это позволяло тестировать их изолированно и видеть в коде, какие именно операции с точки зрения бизнеса существуют в системе.



Form object решает другую задачу. В Rails по умолчанию проверка живёт в модели. Но что если одна и та же модель сохраняется через разные формы с разными наборами обязательных полей? Или если форма агрегирует данные из нескольких моделей? Здесь помогает form object, отдельный объект, который моделирует конкретную форму ввода со своим набором проверок.

Query object инкапсулирует сложный запрос к базе данных. Если у вас есть отчёт, который требует нескольких объединений, подзапросов и сложных условий — это хороший кандидат для выделения в query object. Запрос именован, его можно тестировать изолированно с реальной базой данных, и он не засоряет модель.

Важная оговорка: не каждому приложению нужны все эти паттерны. Преждевременная абстракция так же вредна, как отсутствие абстракции. Service objects стоит вводить тогда, когда контроллер стал делать более чем одну бизнес-операцию, или когда одна и та же операция нужна из нескольких мест. Это хорошее практическое правило.

Все перечисленные паттерны объединяет одно: они делают архитектуру явной. Глядя на папку app/services, новый разработчик сразу видит, какие бизнес-операции существуют в системе. Это само по себе форма документации.

2.4. Когда обратный вызов помогает, а когда превращает систему в ловушку

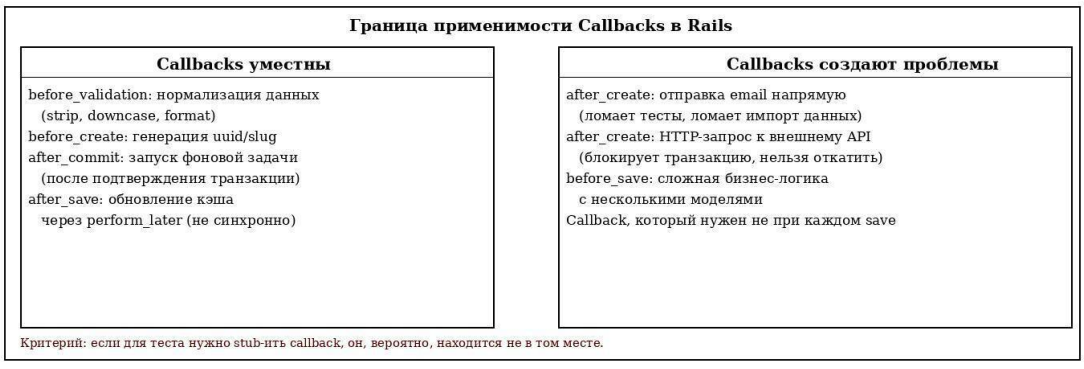
Обратные вызовы (callbacks) в Rails — пожалуй, самая спорная тема в сообществе. Фреймворк предоставляет богатый набор: before_validation, after_validation, before_save, after_save, before_create, after_create, after_commit, after_rollback — и это неполный список. Соблазн велик: можно описать всё, что должно произойти при сохранении объекта, в одном месте, в самом классе.

Рассмотрим конкретный пример. В государственной CRM-системе для работы с заявками, при создании нужно отправить уведомление организатору и записать событие в аудит. Самое простое решение:

Что здесь может пойти не так? Рассмотрим три сценария. Первый: в тестах вы создаёте Application.create! для проверки какой-то бизнес-логики, не связанной с уведомлениями. Но каждый такой вызов запускает отправку письма и запись в аудит. Тесты замедляются, вы вынуждены подменять побочные эффекты.

Второй сценарий: вам нужно импортировать исторические данные из старой системы. Вы создаёте тысячи заявок через Application.create!, и на каждую уходит письмо тысячам пользователей. Чтобы этого избежать, нужно временно отключать обратные вызовы — это не очевидно и легко забыть.

Третий сценарий: ваш обратный вызов зависит от внешнего сервиса. Если этот сервис недоступен, он падает с исключением — и транзакция откатывается. Заявка не сохраняется, хотя данные были корректны.



Означает ли это, что обратные вызовы нужно полностью избегать? Нет. Они оправданы для операций, неразрывно связанных с жизненным циклом объекта данных: before_validation для нормализации данных (нижний регистр, удаление лишних пробелов), after_commit для отложенных задач (именно after_commit, а не after_create — это важно), before_create для генерации идентификаторов и служебных полей.

Практическое правило: если обратный вызов можно запустить в тесте без дополнительных подмен и это не нарушит тест — он, вероятно, уместен. Если для теста нужно подменять обратный вызов — это признак того, что логика в неправильном месте.

Именно это правило подводит нас к следующей теме: Rails API как архитектурная точка, где разделение контроллеров и бизнес-логики становится не просто желательным, а необходимым.

2.5. Rails API как основа современной архитектуры

Одним из значимых этапов в работе стала модернизация платформы: перевод с монолитной архитектуры Rails back + Rails front (ERB-шаблоны) на схему Rails API + Nuxt. Это была полная перестройка слоя взаимодействия с пользователем при полностью сохранённой бизнес-логике на backend.

Что меняется, когда Rails переходит в режим API? Прежде всего, убирается слой представлений из Rails. Ответы формируются в формате JSON. Это упрощает backend-приложение и фокусирует его на том, что оно делает хорошо: обработка бизнес-логики, работа с данными, авторизация.

Но это не только техническое изменение. Rails API — это публичный контракт. Каждый эндпоинт — это соглашение между backend и frontend. Это соглашение должно быть предсказуемым: одинаковые статусы для одинаковых ситуаций, одинаковая структура ошибок. Должно быть версионированным: изменение контракта без версионирования ломает клиентов. И должно быть задокументированным: это не бонус, а часть работы.

Переход на Rails API + Nuxt потребовал нескольких месяцев работы. Основные вызовы: сессионная аутентификация (куки работают иначе при cross-origin запросах), согласование форматов ответов между командами, миграция существующих форм и интерактивных элементов. Но результат оправдал вложения: backend стал чище, frontend получил свободу в выборе UI-паттернов, зоны ответственности двух команд были разделены.

Rails API — это не просто технический выбор. Это решение о том, как организована система в долгосрочной перспективе. Прежде чем переходить на этот путь, стоит честно ответить: какие клиенты будут потреблять API? Насколько сложен UI? Есть ли смысл делить команды?

2.6. Как сохранять скорость разработки без архитектурного хаоса

Rails прославился именно скоростью: от идеи до первой версии приложения за часы, а не недели. Это реальное преимущество. Но по мере роста приложения скорость падает, если архитектура не поддерживается. Это проблема не только Rails, но именно здесь она проявляется особенно остро — из-за того, как легко писать «удобный» код, который создаёт проблемы позже.

Что именно замедляет разработку в большом Rails-приложении? Медленные тесты (приходится запускать весь набор, чтобы убедиться, что ничего не сломано). Неочевидные зависимости (изменение в одном месте ломает что-то совсем в другом). Сложное понимание контекста (чтобы понять, что делает один метод, нужно прочитать несколько файлов). Конфликты при параллельной работе над одними и теми же файлами.

Архитектурные решения, описанные в этой главе, напрямую влияют на эти факторы. Service objects снижают неочевидные зависимости. Form objects ускоряют тесты проверки данных. Query objects делают сложные запросы именованными и изолированно тестируемыми.

Есть и более тонкий фактор: соглашения. Rails силён соглашениями (convention over configuration). Когда команда придерживается единых соглашений — как именовать service objects, как строить ответы API, где жить бизнес-логике — разработчик, открывающий новый файл, уже примерно знает, что его ждёт. Это резко снижает когнитивную нагрузку.

Скорость разработки — это не только скорость написания новой функциональности. Это скорость безопасного изменения существующей. Именно вторая метрика важна для долгоживущего продукта. Архитектура, которая позволяет добавить новую фичу за два дня без страха сломать то, что уже работает — это хорошая архитектура.

Эта глава описала инструменты и принципы архитектуры Rails-приложения за пределами стандартного MVC. Следующий шаг — понять, как эти инструменты применяются в контексте сложной бизнес-логики, которую нужно моделировать внутри монолита.

Rails как фреймворк предоставляет инструменты для построения архитектуры — но не диктует, какой она должна быть. Это одновременно сила и ответственность. Команды, которые воспринимают Rails только как инструмент для быстрого создания приложений, со временем создают системы, которые стагнируют под весом связного неструктурированного кода.

Команды, которые видят в Rails платформу с богатой экосистемой паттернов — service objects, query objects, form objects, явное проектирование API — создают системы, которые остаются управляемыми при масштабировании. Это не сложнее. Это просто требует принятия явных архитектурных решений вместо того, чтобы позволить коду расти самому по себе.

Принципиальный вывод этой главы: архитектурные границы в Rails нужно проводить активно. Не потому что фреймворк плохо устроен, а потому что хороший фреймворк предоставляет свободу — а свобода требует дисциплины.

ГЛАВА 3. ДОМЕННАЯ МОДЕЛЬ В RAILS-МОНОЛИТЕ

Слово «монолит» в инженерной среде приобрело странный оттенок. Его произносят с извинительной интонацией, как будто признаются в чём-то постыдном. «У нас монолит, но мы планируем перейти на микросервисы». Это ожидание перехода, как будто монолит — промежуточная стадия, а не полноценная архитектурная форма.

Я хочу поспорить с этим представлением. Не потому что микросервисы плохи, они решают реальные проблемы в определённых контекстах. А потому что монолит при правильном проектировании может быть зрелой, поддерживаемой, высоконагруженной системой на протяжении многих лет.

Электронная торговая площадка, через которую проходят закупочные процедуры, работает на Rails-монолите. Это не legacy в смысле «устаревшего и страшного». Это production-система, которую мы развивали, модернизировали и масштабировали. Я занимался переводом части этой системы с монолитной структуры на API, но именно потому что понял: проблема была не в монолите как таковом, а в отсутствии чётких доменных границ внутри него.

Эта глава о том, как строить доменную модель внутри Rails-монолита так, чтобы он оставался управляемым при росте. Это разговор о статусах, переходах, транзакциях и о том, где заканчивается ActiveRecord-модель и начинается бизнес-логика.

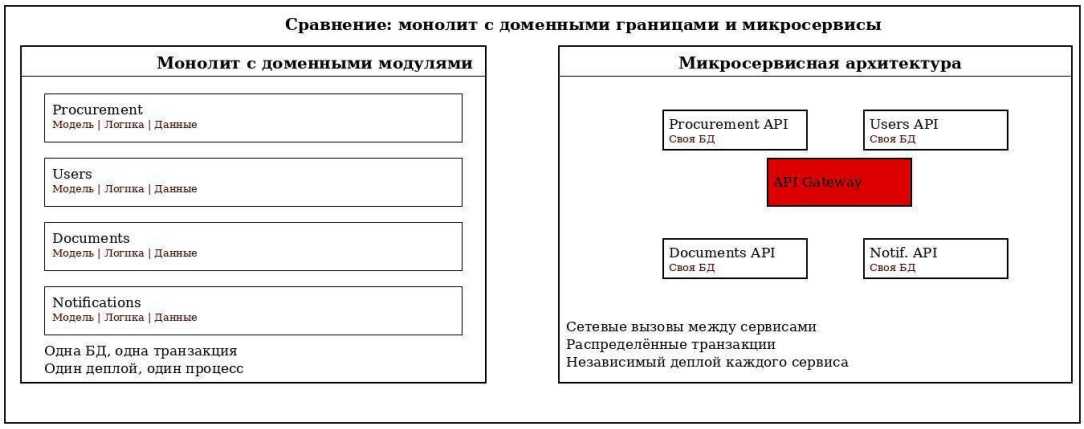
3.1. Почему монолит может быть зрелой архитектурой, а не техническим долгом

Технический долг — это не синоним монолита. Технический долг — это решения, которые вы приняли сознательно или нет и которые создают работу в будущем. Монолит с хорошей структурой и тестовым покрытием не является техническим долгом. Монолит с запутанной логикой, без тестов и с циклическими зависимостями между модулями — да.

У монолита есть реальные преимущества, которые редко признают явно. Транзакционность: когда вся система работает в одном процессе с одной базой данных, вы можете обернуть любую сложную операцию в транзакцию и получить гарантию атомарности. В микросервисах для этого нужны распределённые транзакции или паттерн Saga — значительно сложнее.

Простота отладки: когда что-то ломается, вы смотрите в один стектрейс, один лог, один процесс. Нет необходимости сопоставлять трейсы между несколькими сервисами, чтобы найти, где именно произошла ошибка.

Развёртывание: один артефакт, один процесс деплоя. Для команд среднего размера это существенная разница в сложности инфраструктуры. Возможность рефакторинга: внутри монолита вы можете переименовывать классы, перемещать методы, менять интерфейсы — и всё это проверяется через тесты. В микросервисах изменение интерфейса одного сервиса требует координации с командами всех потребителей.



Это не означает, что монолит подходит всегда. Когда разные части системы требуют принципиально разных характеристик масштабирования, выделение в отдельный сервис имеет смысл. Когда команды разные и работают автономно, микросервисы позволяют деплоить независимо. Но это специфические случаи, а не общее правило.

Позиция, которую я отстаиваю: начинать с хорошо структурированного монолита и выделять сервисы только тогда, когда есть конкретная причина. Не потому что «так все делают», а потому что конкретная задача требует выделения.

С этим пониманием перейдём к тому, что делает монолит управляемым: доменная модель с чёткими правилами и защитой бизнес-логики.

3.2. Критические правила бизнеса: какие ошибки система обязана блокировать

Каждая бизнес-система имеет набор инвариантов — состояний и переходов, которые никогда не должны нарушаться. Для торговой площадки: ставка в аукционе не может быть выше предыдущей ставки, заявка не может быть подана после окончания срока приёма, победитель не может быть назначен, если нет ни одной заявки. Нарушение этих инвариантов — это не просто ошибка. Это юридическая проблема, финансовая ответственность, потеря доверия пользователей.

Вопрос, который я всегда задаю при проектировании: на каком уровне эти инварианты защищены? Если ответ «только в бизнес-логике», то любой прямой запрос к базе данных (из скрипта миграции, из задачи Sidekiq, из тестовых данных) может нарушить инвариант. Правильный ответ: защита на нескольких уровнях. Бизнес-логика в приложении проверяет инварианты и возвращает понятные ошибки пользователю. База данных поддерживает целостность через constraints, которые являются последним рубежом защиты.

Рассмотрим конкретный пример. Заявка на газификацию может быть в одном из нескольких статусов: черновик, подана, на рассмотрении, одобрена, отклонена. Некоторые переходы допустимы (черновик -> подана, на рассмотрении -> одобрена), некоторые нет (одобрена -> черновик). Это правило должно быть отражено в коде явно:

Этот код явно документирует, какие переходы разрешены. Любая попытка выполнить недопустимый переход вызовет исключение с понятным сообщением. Для самых важных инвариантов стоит добавить ограничение на уровне базы данных — PostgreSQL поддерживает CHECK constraints, которые проверяются при каждой записи.

Критические правила бизнеса должны быть в коде явными и названными. Не просто условием в методе, а именованной концепцией с понятным исключением при нарушении. Это облегчает отладку (вы сразу видите, какое правило нарушено) и документирует доменные ограничения в самом коде.

3.3. Статусы, переходы и жизненные циклы сущностей

Жизненный цикл сущности — один из самых богатых источников бизнес-логики в любой системе. Заявка проходит через статусы. Процедура закупки имеет этапы. Договор подписывается, исполняется, закрывается или расторгается. Каждый переход имеет условия, которые должны выполняться, и действия, которые должны произойти.

В Rails для управления жизненными циклами часто используют гем AASM (Acts As State Machine) или Statesman. AASM — более простой, встраивается прямо в ActiveRecord-модель. Statesman — более строгий, хранит историю переходов отдельно. Вот как выглядит управление жизненным циклом торговой процедуры с AASM:

Что даёт этот подход? Граф переходов явно описан в одном месте. Недопустимые переходы автоматически вызывают исключение. Guards проверяют предусловия перед переходом. After-хуки запускают действия после успешного перехода.

Работая над оптимизацией критических сервисов, связанных с проведением торгов, я столкнулся с реальной проблемой: after-хуки запускались синхронно внутри транзакции, включая медленные операции (уведомления, обновление кэша). Если уведомление занимало секунду, транзакция держала блокировку на секунду. При высоком параллелизме это быстро превращалось в проблему производительности. Решение: выносить медленные операции в фоновые задачи через perform_later, а не выполнять их синхронно в хуке.

3.4. Транзакции как инструмент защиты доменной целостности

Транзакция в базе данных — одна из мощнейших гарантий при работе с данными. Она гарантирует: либо все операции внутри выполнены успешно и зафиксированы, либо ни одна из них не применена. Это свойство называется атомарностью.

В Rails транзакции доступны через ApplicationRecord.transaction {}. Если любая из операций внутри блока вызывает исключение, все изменения откатываются. Договор не будет создан, если изменение статуса процедуры завершилось ошибкой. Запись в аудит не появится, если договор не был создан.

Но есть несколько тонкостей, важных для понимания. Первое: perform_later внутри транзакции ставит задачу в очередь, но задача будет запущена независимо от результата транзакции. Если транзакция откатится после того, как задача уже поставлена, задача выполнится для данных, которых теперь нет. Правильное решение — использовать after_commit хук.

Второе: Rails автоматически открывает транзакцию при вызове save! или create!. Если вы явно открываете ещё одну транзакцию внутри — используются вложенные транзакции. Поведение вложенных транзакций в PostgreSQL специфично: внутренняя транзакция создаёт savepoint, но откат внутренней транзакции не откатывает внешнюю автоматически.

Третья тонкость: не все операции участвуют в транзакции. Запись в лог-файл, HTTP-запрос к внешнему API, отправка письма напрямую — всё это происходит немедленно и не откатывается при откате транзакции. Это нужно учитывать при проектировании операций: действия с внешними побочными эффектами должны выполняться либо после подтверждения транзакции, либо быть идемпотентными.

Разобравшись с транзакционной защитой, подойдём к вопросу, который часто остаётся без явного ответа: где именно в архитектуре Rails-приложения должна жить бизнес-логика?

3.5. Где заканчивается ActiveRecord-модель и начинается бизнес-логика

Это один из самых практически важных вопросов в Rails-архитектуре. Граница между «данными» (ActiveRecord) и «бизнесом» размыта, и Rails не принуждает её соблюдать. Разработчики часто добавляют бизнес-методы прямо в модель, потому что это удобно: модель уже знает о своих атрибутах, об ассоциациях, о базе данных.

Проблема возникает, когда модель начинает знать о вещах, выходящих за пределы её собственных данных. Модель Application не должна знать о том, как работает очередь уведомлений. Модель Procedure не должна знать, какой формат ответа ожидает JSON API. Модель User не должна знать о деталях платёжной системы.

Рабочая граница, которую я использую: в ActiveRecord-модели живут операции, которые полностью определяются данными самой модели. Области видимости, проверки данных (формат, наличие обязательных полей), атрибуты и вычисляемые свойства (full_name = first_name

+ last_name), простые ассоциации. Всё остальное — в service objects, domain objects или другой структуре за пределами модели.

Конкретные сигналы о нарушении этой границы: модель имеет инициализацию зависимостей (например, инициализацию HTTP-клиента). Модель вызывает другую модель не через ассоциацию. Метод модели принимает параметры, которые имеют смысл только в контексте HTTP-запроса. Метод модели отправляет уведомления или пишет в лог. Как только хотя бы один из этих сигналов появляется — метод нужно выносить.

Один из показательных примеров из практики: в цифровой платформе для тендерных процедур была модель Bid (ставка на торгах). Она содержала метод для проверки, является ли ставка ведущей. Этот метод запрашивал текущую лучшую ставку по процедуре. Казалось бы, логично: ставка знает о своей процедуре через ассоциацию. Но этот метод постепенно оброс дополнительной логикой: учёт ценовых шагов, проверка времени подачи, особые правила для отдельных типов процедур. В итоге получился метод в модели Bid, который неявно зависел от всей бизнес-логики проведения торгов. Рефакторинг показал: эта логика должна жить в BidEvaluationService.

3.6. Как моделировать бизнес-процессы в Rails-приложении

Бизнес-процесс в отличие от отдельной операции имеет продолжительность и состояние. Процесс газификации дома длится месяцами: заявка подана, документы проверены, технические условия выданы, работы выполнены, подключение произведено. Процесс закупки: от публикации до подписания договора. Арбитражный процесс: от подачи иска до решения.

Как это моделировать в Rails? Несколько подходов, которые я использовал.

Первый подход: машина состояний на модели (рассматривалась выше). Хорошо работает для линейных или относительно простых нелинейных жизненных циклов. Проблема: при росте сложности граф состояний становится трудночитаемым.

Второй подход: явный объект процесса, который знает о текущем шаге и умеет переходить к следующему:

Мы использовали именно второй подход — явный объект процесса с записью шагов. Это позволяло при обращении пользователя точно показывать, на каком этапе находится его заявка, сколько дней прошло на каждом этапе и когда ожидается следующий шаг. Пользователи, которые следят за статусом газификации своего дома, ценят такую прозрачность.

Правильно смоделированный бизнес-процесс в коде — это не только техническое решение. Это способ зафиксировать договорённости между разработчиками и бизнесом о том, как работает система. Когда бизнес-аналитик читает код и видит шаги, которые он сам описывал в спецификации — это признак хорошей абстракции.

Монолит не является синонимом технического долга. Это архитектурная форма с реальными преимуществами, особенно для команд среднего размера, работающих над сложной предметной областью. Условие успеха, не размер системы, а наличие чётких доменных границ, явных инвариантов и правильно организованных слоёв ответственности.

Критические правила бизнеса должны защищаться на нескольких уровнях. Жизненный цикл сущностей должен быть смоделирован явно, а не разбросан по условным конструкциям. Транзакции — инструмент защиты, а не деталь реализации. Граница между ActiveRecord и бизнес-логикой требует активного соблюдения, а не пассивного ожидания.

Нетривиальный вывод, к которому я пришёл после лет работы с производственными системами: качество доменной модели напрямую влияет на то, как быстро команда реагирует на изменения в бизнесе. Плохо смоделированный домен означает, что любое изменение требования превращается в рефакторинг нескольких слоёв. Хорошо смоделированный домен означает, что изменение бизнес-правила отражается в коде именно там, где оно логически находится.

Следующая глава посвящена API-first архитектуре — тому, как выстроить контракт между backend и его потребителями так, чтобы он оставался стабильным, задокументированным и предсказуемым при изменениях системы.

ГЛАВА 4. API-FIRST АРХИТЕКТУРА НА RAILS

Когда разработчик говорит «я написал API», он часто имеет в виду набор эндпоинтов, которые возвращают JSON. Это необходимое условие, но не достаточное. API — это прежде всего контракт. Публичное соглашение о том, как выглядят запросы, что означают ответы, как обрабатываются ошибки и что происходит при изменениях.

Именно к этому пониманию я пришёл после того, как мы переводили систему электронных торгов и закупок с монолитной структуры Rails back + Rails front на схему Rails API + Nuxt. Пока API обслуживает только один фронтенд в рамках одной команды — вы можете позволить себе некоторую небрежность: поменяли поле, предупредили коллегу. Но когда API становится интерфейсом между несколькими системами, изменения без версионирования и документации начинают стоить реальных денег и нервов.

Конец ознакомительного фрагмента.