PROJECTS

Website Rebuild with Ruby on Rails

Convention Over Configuration, Verified Honestly

A Fourth Arrival at the Same Flexible URL Model

ActiveRecord, ERB & Rails' Own routes.rb DSL

A Reversed Philosophy for Secrets Management

Capstone: The Website Rebuild Series, Complete

Philip Osztromok

Generated with Claude

Table of Contents

Website Rebuild with Ruby on Rails — 12 Chapters

This is the fourth and final course in the Website Rebuild series, alongside the already-complete Next.js, Django, and Laravel rebuilds. Rails is famous for one specific idea — **convention over configuration** — and it's worth being honest from the very first chapter about what that phrase actually means once Laravel is already sitting in the comparison, rather than treating it as a clean, unique selling point.

Generating the Application

```
gem install rails
rails new website-rebuild-with-rails --database=mysql
cd website-rebuild-with-rails
bundle install
```

`--database=mysql` adds the `mysql2` gem and pre-fills `config/database.yml` for MySQL, matching the site's own real production database rather than defaulting to Rails' own out-of-the-box SQLite. `rails new` generates a full directory structure in one step — `app/models`, `app/controllers`, `app/views`, `config/routes.rb`, `db/` — all present before a single line of application code exists.

Convention Over Configuration — the Honest Version

Rails coined this phrase, and it's a real, defining design choice: a model class named `Page` maps to a `pages` table with zero configuration, timestamp columns are managed automatically, and a single line in `routes.rb` can generate a full set of RESTful routes. But Laravel Rebuild's own chapters already showed Eloquent doing several of these exact same things — automatic pluralized table names, no redeclaring columns in the model, its own one-line `Route::resource()`. That's not a coincidence.

> **LARAVEL'S OWN ACKNOWLEDGED DEBT TO RAILS**
>
> Laravel was explicitly designed drawing on Rails' own ideas — the two frameworks are close philosophical cousins, sharing far more DNA with each other than either shares with Django's more explicit style or with Next.js/Prisma's fully separate schema DSL. Framing this chapter as "Rails infers, Laravel configures" would be inaccurate; the real, honest distinction has to be found somewhere more specific than the phrase "convention over configuration" itself.

What's Actually Rails-Specific: Ruby's Own Open Classes

```
# ActiveSupport extends Ruby's own core classes directly
3.days.ago
"CamelCase".underscore    # => "camel_case"
"page".pluralize          # => "pages"
```

`3.days.ago` isn't a helper function wrapping the number `3` — it's a real method call on Ruby's own `Integer` class, added directly to the language itself by ActiveSupport. Ruby permits this style, called **monkey-patching** or reopening a class, as an ordinary, idiomatic part of the language. Neither Python nor PHP's own ecosystems lean on this technique the way Rails does: Python's own community explicitly discourages patching built-in types (part of the same "explicit is better than implicit" philosophy visible in Django's own more spelled-out models), and PHP doesn't offer an equally natural mechanism for extending its built-in scalar types at all. This — not table-name inference, which Laravel already shares — is the genuinely distinguishing trait worth naming here.

Four Frameworks, Compared Honestly

	NEXT.JS	DJANGO	LARAVEL	RAILS
Model-to-table mapping	Explicit — a separate `schema.prisma` DSL	Explicit — fields declared in the model class	Inferred — no field redeclaration	Inferred — no field redeclaration
URL-to-handler wiring	Implicit — folder structure alone	Explicit — `urlpatterns`	Explicit — `routes/web.php`	Explicit — `routes.rb`
Core-language extension	Not idiomatic in JS/TS	Discouraged in Python	Not natural in PHP	Idiomatic — ActiveSupport monkey-patches core classes

> **NOT A UNIQUELY RAILS ADVANTAGE**
>
> Laravel and Rails are genuinely the closest pair in this comparison on data-layer conventions specifically. The real fourth-way split shows up elsewhere in this course — Chapter 5's asset-bundling philosophy and Chapter 9's authentication chapter both land on genuinely distinct positions, not shared ones.

Hands-On Exercises

EXERCISE 1

Explain the real, historical relationship between Rails' convention-over-configuration philosophy and Laravel's own design, and name two specific conventions the two frameworks genuinely share.

View solution

EXERCISE 2

Explain what `3.days.ago` actually is — not what it does, but what kind of language operation it represents — and explain why this specific technique isn't something Django's or Laravel's own ecosystems lean on the same way.

View solution

EXERCISE 3

Explain why `rails new` was run with `--database=mysql` for this specific rebuild rather than Rails' own SQLite default.

View solution

CHAPTER 1 QUICK REFERENCE

- `rails new --database=mysql` — generates the full application structure with MySQL pre-configured
- **Convention over configuration** — Rails coined it, but Laravel shares much of it (pluralized table names, no field redeclaration, one-line resourceful routing)
- **Genuine Rails-specific trait** — Ruby's own open classes, leaned on pervasively by ActiveSupport (`3.days.ago`, `.pluralize`, `.underscore`)
- **Honesty check** — the real four-way split shows up in later chapters (asset bundling, authentication), not in the data layer, where Laravel is genuinely closest to Rails
- **Next chapter:** Designing a Flexible URL & Content Model

Every sibling course in this series reached the same conclusion independently: a plain adjacency list (just a `parent_id`) makes reads slow at depth, a pure materialized path with no `parent_id` makes reparenting awkward, a nested-set model makes every insert expensive, and a closure table adds a whole second table for a problem this site's own read-heavy, write-rare traffic doesn't need solved that heavily. The adjacency-list-plus-materialized-path hybrid — both a `parent_id` foreign key and a precomputed `full_path` string, kept in sync — wins a fourth independent time here, this time via ActiveRecord.

The Migration

```ruby
# db/migrate/XXXXXX_create_pages.rb
class CreatePages < ActiveRecord::Migration[7.1]
  def change
    create_table :pages do |t|
      t.string :slug, null: false
      t.string :full_path, null: false
      t.string :title, null: false
      t.text :body
      t.references :parent, foreign_key: { to_table: :pages }, null: true

      t.timestamps
    end

    add_index :pages, :full_path, unique: true
end
end
```

`t.references :parent, foreign_key: { to_table: :pages }` is a single line that both creates the `parent_id` column and adds a real database foreign-key constraint pointing back at the same table — a self-relationship, in one terse call.

The Model: `belongs_to`, `has_many`, and the Deferred Cascade Cost

```ruby
# app/models/page.rb
class Page < ApplicationRecord
  belongs_to :parent, class_name: 'Page', optional: true
  has_many :children, class_name: 'Page', foreign_key: :parent_id,
           dependent: :restrict_with_error

  before_save :compute_full_path
private
def compute_full_path
    self.full_path = parent ? "#{parent.full_path}/#{slug}" : slug
  end
end
```

The `before_save` callback recomputes `full_path` from the current `parent` and `slug` every time a page is saved — but only for the page being saved. If a page's parent changes, every one of its own descendants' `full_path` values goes stale, since nothing here walks the tree. Fixing that is deliberately deferred — the real cost gets paid in Chapter 10, exactly where every sibling course paid the identical cost.

Verifying `dependent: :restrict_with_error` Against Laravel's `restrictOnDelete()`

SAME CATEGORY AS DJANGO, GENUINELY DIFFERENT FROM LARAVEL

`dependent: :restrict_with_error` is implemented as an ActiveRecord *callback* — when `page.destroy` is called, Rails checks whether `children` exist and, if so, adds an error and aborts, entirely in application code, before any `DELETE` statement reaches the database. That puts it in the same general category as Django's own ORM-level `PROTECT` check. Laravel's `restrictOnDelete()`, by contrast, is a genuine database schema constraint — enforced by the database engine itself, regardless of how the delete is attempted. A raw SQL `DELETE` bypassing ActiveRecord entirely would skip Rails' own check; it could never bypass Laravel's.

A REAL BACKSTOP, VERIFIED HONESTLY

`t.references :parent, foreign_key: ...` still adds a genuine foreign-key constraint at the database level. MySQL's own default behavior for a foreign key with no explicit `ON DELETE` clause is effectively `RESTRICT` — so even if Rails' own application-level check were somehow bypassed, the database itself would still refuse the delete, just with a raw MySQL error instead of a friendly ActiveRecord validation message. The friendly check and the database's own default behavior work as two real layers, not one.

Four Tree Strategies, One Recurring Winner

STRATEGY	REAL COST AT THIS SITE'S SCALE
Plain adjacency list	Reading a full path at depth needs one query per level
Pure materialized path	Reparenting means no structured foreign key to lean on
Nested set	Every insert renumbers a wide range of sibling rows
Closure table	An entire second table, for a scale this site doesn't have
Adjacency list + materialized path (chosen)	One extra column kept in sync; the cascade cost is real but deferred to Chapter 10

Hands-On Exercises

EXERCISE 1

Explain the adjacency-list-plus-materialized-path hybrid this course arrives at a fourth independent time, and name the two pieces of information stored on each `Page` row that make it work.

View solution

EXERCISE 2

Explain the real technical difference between Rails' `dependent: :restrict_with_error` and Laravel's `restrictOnDelete()`, specifically which layer — application or database — each one operates at.

View solution

EXERCISE 3

Explain why `full_path` is computed inside a `before_save` callback rather than being set directly by whatever code creates or updates a `Page`.

View solution

CHAPTER 2 QUICK REFERENCE

- `t.references :parent, foreign_key: { to_table: :pages }` — a self-relationship in one line, with a real FK constraint
- `belongs_to` / `has_many ... dependent: :restrict_with_error` — the self-referencing relationship and its delete guard
- `before_save :compute_full_path` — recomputes the moved/created page's own path, not its descendants'
- **Deferred cascade cost** — descendant paths go stale on reparent; paid for real in Chapter 10
- **Verified nuance** — Rails' delete guard is an application-level callback (like Django's `PROTECT`), not a database constraint (like Laravel's `restrictOnDelete()`) — though MySQL's own default FK behavior still backstops it
- **Next chapter:** Routing — Rails' Own `routes.rb` DSL

Next.js infers routes purely from the filesystem. Django's `urlpatterns` is a plain Python list assigned at module load. Laravel's `routes/web.php` is explicit PHP code using a fluent, chainable API. Rails' `routes.rb` is explicit Ruby code too — but it leans on the language's block syntax more heavily than any of the three, in a way worth showing concretely rather than just asserting.

`routes.rb` as an Executed Block-Based DSL

```ruby
# config/routes.rb
Rails.application.routes.draw do
  root to: 'pages#show', defaults: { path: '' }

  get '*path', to: 'pages#show', as: :page, constraints: { path: /.+/ }
end
```

`Rails.application.routes.draw do ... end` passes an entire block of route declarations to a single method call, and nested structures — namespaces, resource groups, member/collection routes — are expressed as further nested blocks inside it. Laravel's own routes file is genuinely explicit code too, using a similarly fluent style (`Route::get(...)->middleware(...)->name(...)`), so the real distinction isn't "explicit vs. implicit" a second time — it's that Rails leans specifically on Ruby's block syntax, a language feature PHP has no direct equivalent of, for its own nesting.

The Glob Wildcard Segment

`get '*path', to: 'pages#show'` uses Rails' own globbing syntax — a segment prefixed with `*` in the route pattern string itself — to capture every remaining path segment, slashes included, into `params[:path]`. No separate regex constraint is required the way Laravel's own wildcard needed one.

A Dedicated Keyword for the Homepage: `root`

A GENUINE, SMALL RAILS-SPECIFIC TOUCH

`root to: 'pages#show', defaults: { path: '' }` is Rails' own dedicated DSL keyword specifically for the homepage route — not a generic `get '/'` call reused for the purpose, the way Laravel and Django both effectively handle their own empty-path case. It's a small thing, but it's a genuinely distinct, semantically named piece of syntax none of the three siblings have a direct equivalent for.

Order Still Matters — the Same Gotcha, Repeated Honestly

NOT A NEW PROBLEM — THE SAME ONE, IN RUBY

Rails matches routes top-to-bottom, first match wins — the identical behavior already flagged in both the Django and Laravel rebuild courses' own Chapter 3. A catch-all `get '*path'` declared before a more specific route (an admin route, for instance) would swallow it. This isn't a new Rails-specific trap; it's the same trap every one of these routing systems shares, worth repeating accurately rather than dressed up as something novel.

Four Wildcard Mechanisms, Compared Honestly

	NEXT.JS	DJANGO	LARAVEL	RAILS
Where it lives	The filesystem itself — `[...path]`	`urlpatterns` — `<path:full_path>`	`routes/web.php` — `{path?}`	`routes.rb` — `*path`
Needs a separate regex	No	No — the converter itself is slash-aware	Yes — `->where('path', '.*')`	No — the `*` glob is slash-aware itself
Named route helper	N/A — file-based	`reverse()` / `{% url %}`	`route('name')`	`page_path`

Hands-On Exercises

EXERCISE 1

Explain what makes `Rails.application.routes.draw do ... end` a block-based DSL, and why this is a real but modest distinction from Laravel's own fluent `Route::` calls rather than a difference in explicitness.

View solution

EXERCISE 2

Explain what Rails' glob segment (`*path`) does, and why it needs no separate regex constraint the way Laravel's own wildcard route did.

View solution

EXERCISE 3

Explain why a catch-all route like `get '*path'` has to be declared carefully relative to more specific routes, and name the two sibling courses that already established this same gotcha in their own frameworks.

View solution

CHAPTER 3 QUICK REFERENCE

- `Rails.application.routes.draw do ... end` — a block-based DSL, not a plain data structure
- `get '*path', to: 'pages#show'` — the glob wildcard, slash-aware with no separate regex needed
- `root to: ..., defaults: { path: '' }` — a dedicated, semantically named keyword just for the homepage
- **Route order still matters** — the same top-to-bottom precedence gotcha already flagged for Django and Laravel
- **Honesty check** — named route helpers and resourceful routing are genuinely shared with Laravel, not unique to Rails
- **Next chapter:** Views: ERB Templates & the Rails MVC

A Rails Controller action renders a view — the same MVC shape as every sibling course. The template layer, ERB, is where this chapter finds its own genuinely distinctive material: not in layouts and partials, which every sibling already has an equivalent of, but in exactly how much of the host language ERB lets through, and in a dedicated architectural layer — Helpers — the other three siblings don't offer quite the same way.

Layouts: The Same Wrapping Shell, ERB's Own Syntax

```
<!-- app/views/layouts/application.html.erb -->
<!DOCTYPE html>
<html>
  <head><title><%= @page&.title || 'Untitled' %></title></head>
  <body>
    <%= yield %>
  </body>
</html>
```

`<%= yield %>` is Rails' own equivalent of Blade's `@yield` and Django's `{% block %}` — the point where the current page's own template content is dropped into the shared shell. `<%= %>` outputs the expression's result into the page; plain `<% %>` executes Ruby without outputting anything — used for control flow like loops and conditionals.

Partials: Reusable View Fragments

```
<%# app/views/pages/show.html.erb -->
<%= render partial: 'breadcrumb', locals: { page: @page } %>
<h1><%= @page.title %></h1>
<%== @page.body %>
```

`<%== %>` outputs without HTML-escaping — ERB's own direct equivalent of Blade's `{!! !!}` and Django's `|safe` filter, needed here for the same reason: `@page.body` is trusted, already-formatted HTML content, not user input that needs escaping.

A Third Place for View Logic: Helpers

```ruby
# app/helpers/pages_helper.rb
module PagesHelper
  def breadcrumb_for(page)
    ancestors = []
    current = page
    while current
      ancestors.unshift(current)
      current = current.parent
    end
    ancestors
  end
end
```

Called from the view as `breadcrumb_for(@page).each do |crumb| ... end`. Rails ships `app/helpers/` by default, and every method defined there is automatically available in every view — a dedicated architectural layer specifically for view-supporting logic that's too presentational for a model method but too structured to sprawl across the template itself.

A REAL THIRD OPTION, NOT JUST A FOURTH NAME FOR THE SAME IDEA

Django's own Chapter 4 built its breadcrumb walk directly inside the view function — there was nowhere else idiomatic to put it. Laravel's own Chapter 4 built it inside the Controller, by convention rather than necessity, since Blade's `@php` would technically have allowed it inline. Rails offers a genuinely separate, named layer for exactly this kind of logic — Helpers aren't a workaround; they're a first-class, intended part of the MVC structure.

SET UP DELIBERATELY, NOT ACCIDENTALLY

`breadcrumb_for` walks `current.parent` in a loop — one query per level, exactly the same N+1 pattern both sibling courses' own Chapter 4/6 built into their breadcrumb code on purpose. It's left as written here, to be caught and fixed live in Chapter 6, matching the running motif across the whole series.

ERB's Permissiveness, Verified Honestly

ERB has no separate template language layered on top of Ruby the way Django's DTL is a genuinely separate, deliberately restricted language — `<% %>` tags contain real, unrestricted Ruby, full stop, with no curated allow-list of tags/filters standing between the template and the language. That puts ERB closer in spirit to Blade's `@php` escape hatch and to Next.js's own JSX (real JavaScript/TypeScript by default) than to Django — with one honest nuance: Blade normally chooses *not* to use raw PHP, reaching for its own sugared directives (`@if`, `@foreach`) that compile down to the same plain PHP `@php` would let through directly, while ERB has no separate sugared layer at all — `<% %>` already *is* the whole language, by design, not an escape hatch from something more restricted.

	NEXT.JS (JSX)	DJANGO (DTL)	LARAVEL (BLADE)	RAILS (ERB)
Full host-language access	Yes — JSX is real JS/TS	No — a deliberately restricted separate language	Yes, via `@php` — though sugared directives are the default style	Yes — no separate language exists at all
Is there a "restricted mode"?	No	Yes — the entire point of DTL	No — `@php` is always available	No

Hands-On Exercises

EXERCISE 1

Explain what a Rails Helper module is, and why it's a genuinely different architectural option from both Django's "build it in the view" approach and Laravel's "build it in the Controller by convention" approach.

View solution

EXERCISE 2

Explain the real difference between ERB's own permissiveness and Blade's `@php` permissiveness, given that both technically allow full host-language code.

View solution

EXERCISE 3

Explain why `breadcrumb_for` is expected to have a performance problem, and where in this course that problem is deliberately meant to be caught and fixed.

View solution

CHAPTER 4 QUICK REFERENCE

- `<%= yield %>` — the layout's own insertion point, Rails' equivalent of `@yield` / `{% block %}`
- `<%= %>` / `<% %>` / `<%== %>` — output, execute-only, and unescaped output
- `app/helpers/` — a genuine third architectural layer for view-supporting logic, distinct from both siblings' own approaches
- **ERB's own permissiveness** — no separate template language exists; `<% %>` already is unrestricted Ruby, by design
- **Deliberate setup** — `breadcrumb_for`'s N+1 pattern is left in place on purpose, caught in Chapter 6
- **Next chapter:** Styling — Dark Theme

Three siblings, three positions on asset bundling: Next.js bundles by mandatory framework design, Django ships nothing at all, Laravel pre-wires an optional bundler (Vite) without being one itself. Rails 7+ reaches a fourth position that isn't just another point on the same spectrum — it's a deliberate attempt to avoid needing a JavaScript bundler in the first place.

Propshaft: Fingerprinting, Not Bundling

Rails 7.1+ defaults to **Propshaft** as its asset pipeline — a genuinely minimal one. Propshaft doesn't compile Sass, doesn't transpile anything, and doesn't bundle multiple files into one. It does exactly one job: it fingerprints static asset files with a content-based digest for cache-busting, and serves them. Any real compilation this site's own assets might need is delegated to separate, opt-in tools — not baked into the default pipeline the way Next.js's bundler is baked into that framework.

```
<!-- app/views/layouts/application.html.erb -->
<%= stylesheet_link_tag "application" %>
```

`app/assets/stylesheets/application.css` holds this site's own dark theme — plain CSS, no preprocessing needed for it, so Propshaft's own "just fingerprint and serve" behavior is already the entire pipeline this chapter requires.

Importmap: No Bundler for JavaScript, Deliberately

```
# config/importmap.rb
pin "application", preload: true
```

```
<!-- app/views/layouts/application.html.erb -->
<%= javascript_importmap_tags %>
```

`javascript_importmap_tags` renders a browser-native `<script type="importmap">` block, mapping bare module names to real file URLs — resolved directly by the browser's own native ES module support, with no bundling step running anywhere, in development or production. `app/javascript/application.js` holds this site's own copy-to-clipboard script, the same small enhancement used throughout every other course on the site, needing nothing more than what the browser already provides.

> **A PHILOSOPHY, NOT JUST A FOURTH FLAVOR**
>
> Next.js's bundler is mandatory — there's no path around it. Laravel's Vite is optional in principle but pre-wired and expected. Django offers no pipeline to opt into at all. Rails 7's own Importmap default is a deliberate design position: for an application that doesn't need a large client-side JavaScript framework, the browser's own native module resolution is treated as sufficient, and reaching for Node/npm/a bundler is an opt-in escalation (via `jsbundling-rails`) rather than a default assumption. That's a genuinely different starting philosophy, not a fourth data point on the same spectrum.

Four Positions on Asset Bundling

	NEXT.JS	DJANGO	LARAVEL	RAILS
Default JS pipeline	A bundler, mandatory	None	Vite, pre-wired but a real bundler	Importmap — no bundler at all
Node/npm required?	Always	Never (nothing built in)	Yes, for `npm run build`	Not for the default case
CSS handling	Bundled with the build	Served as-is via `collectstatic`	Bundled via Vite	Fingerprinted and served by Propshaft, no processing

FORWARD REFERENCE

Since nothing here depends on Node or a build step, Chapter 11's own deployment story stays simple on this front too — `bin/rails assets:precompile` runs Propshaft's own fingerprinting during deploy, with no separate JavaScript build pipeline to coordinate.

Hands-On Exercises

EXERCISE 1

Explain what Propshaft actually does — and, just as importantly, what it deliberately doesn't do — compared to a real bundler like the one built into Next.js.

View solution

EXERCISE 2

Explain how Importmap resolves a JavaScript module without a bundler, and why this is described in this chapter as a philosophical position rather than just a fourth entry on the same spectrum as Next.js, Django, and Laravel.

View solution

EXERCISE 3

Explain why this course's own dark theme CSS needs nothing more than Propshaft's default fingerprint-and-serve behavior, with no preprocessing step involved.

View solution

CHAPTER 5 QUICK REFERENCE

- **Propshaft** — Rails 7.1+'s default asset pipeline: fingerprint and serve, no compilation or bundling
- `stylesheet_link_tag "application"` — renders the fingerprinted CSS link tag
- **Importmap** — browser-native ES module resolution, no bundler required for the default case
- `javascript_importmap_tags` — renders the native `<script type="importmap">` block
- **A genuine fourth philosophy** — not "bundle by default," "nothing built in," or "bundler pre-wired," but "avoid needing a bundler at all"
- **Next chapter:** The Database: ActiveRecord ORM

Migrations were already introduced in Chapter 2. This chapter's own real work is catching Chapter 4's `breadcrumb_for` helper in the act — the same N+1 pattern deliberately planted, and deliberately caught, in every sibling course's own database chapter.

Migrations & the Rails Console, Briefly

```
rails generate migration AddTitleIndexToPages
rails db:migrate
rails console
```

`rails console` drops into a real Ruby REPL with every model already loaded — the fastest way to reproduce the breadcrumb's own query pattern directly and see it happen.

Catching the N+1 Pattern, Live

```
# app/helpers/pages_helper.rb — as written in Chapter 4
def breadcrumb_for(page)
  ancestors = []
  current = page
  while current
    ancestors.unshift(current)
    current = current.parent   # one query per level, every time
end
  ancestors
end
```

Loading the capstone's own five-level-deep page triggers five separate `SELECT` queries just to walk up to the root — exactly the pattern this chapter exists to fix.

`.includes()`: A Mechanism That Chooses Its Own Strategy

```
@page = Page.includes(parent: { parent: { parent: { parent: :parent } }
}).find(params[:id])
```

VERIFIED: A GENUINELY DIFFERENT MECHANISM FROM BOTH SIBLINGS

Eloquent's `with()` always runs two separate, batched queries — one for the base rows, one `WHERE IN` query for everything related. Django's `select_related` always runs a single `JOIN` — one query, full stop. ActiveRecord's `includes()` is the only one of the three that dynamically *chooses*: by default it behaves like Eloquent's own two-query strategy, but if the association is also referenced inside a `where` clause (via `.references(:parent)`), Rails switches automatically to a single `LEFT OUTER JOIN` instead. The strategy is a property of how the query is used, not a fixed property of the method itself.

The Same Depth Limitation, Shared Honestly

NOT A RAILS-SPECIFIC SHORTCOMING

The nested `includes(parent: { parent: ... })` call above pre-loads exactly five levels — no more. A page six levels deep would still trigger one extra query past what was pre-loaded. This is the identical limitation Eloquent's own nested `with('parent.parent.parent.parent')` and Django's own `select_related('parent__parent__parent__parent')` already share: none of the three ORMs' built-in eager-loading mechanisms natively express "load the entire ancestor chain, however deep it happens to be." A truly unbounded-depth tree would need a raw recursive SQL query (`WITH RECURSIVE`) via `find_by_sql`, genuinely out of scope for this course's own realistic, bounded page hierarchy.

Three ORMs' Eager-Loading, Compared

	DJANGO (`SELECT_RELATED`)	LARAVEL (`WITH()`)	RAILS (`INCLUDES()`)
Default strategy	Always a single `JOIN`	Always two batched queries	Two batched queries by default
Can it switch to a JOIN?	N/A — always a JOIN	No	Yes — when the association is referenced in a filter
Fixed-depth limitation	Yes — same nested-chain syntax	Yes — same nested-chain syntax	Yes — same nested-hash syntax

Hands-On Exercises

EXERCISE 1

Explain why `breadcrumb_for` triggers exactly five queries for the capstone's own five-level-deep page, and identify which line of the method is responsible.

View solution

EXERCISE 2

Explain the real, verified difference between ActiveRecord's `includes()` and both Eloquent's `with()` and Django's `select_related` — specifically, what makes `includes()`'s own behavior dynamic rather than fixed.

View solution

EXERCISE 3

Explain why a nested `includes(parent: { parent: ... })` call has a real, honest limitation for an arbitrary-depth tree, and name the two sibling ORMs that share the identical limitation.

View solution

CHAPTER 6 QUICK REFERENCE

- `rails console` — a real Ruby REPL with every model loaded, fastest way to reproduce a query pattern
- **N+1 caught** — `breadcrumb_for`'s `current.parent` loop, one query per tree level
- `.includes(parent: { parent: ... })` — nested eager-loading, fixed-depth by necessity
- **Verified nuance** — `includes()` dynamically chooses two-query vs. single-JOIN, unlike Eloquent's always-two-query `with()` or Django's always-one-JOIN `select_related`
- **Shared honesty** — all three ORMs' eager-loading is fixed-depth; a truly unbounded tree needs a raw recursive query
- **Next chapter:** Rendering Content & the Kanji Edge Case

Kanji's two original constraints — a fixed-depth hierarchy with no natural slot for individual characters, and no self-contained-fragment convention for a page with inline stroke-animation markup — are already resolved the same way every sibling course resolved them: Chapter 2's arbitrary-depth `Page` model has no special-cased depth limit, and Chapter 4's `<%== @page.body %>` unescaped rendering lets a kanji page's own markup render through the exact same path as any other page. There's nothing new to solve there. What's worth checking carefully is whether Ruby introduces any new risk of its own.

Ruby's Strings: Closer to Python Than to PHP

```
title = "水のページ"
title.length        # => 5 — five characters, not the byte count
title[0, 3]          # => "水のペ" — sliced by character position
```

Ruby's `String` class carries its own encoding metadata (UTF-8 by default since Ruby 2.0), and every core method — `length`, `[]`, `slice`, `each_char` — operates on characters in that encoding, never on raw bytes. There's no `substr()`-vs-`mb_substr()`-style split in Ruby at all, because there's no byte-oriented string family to accidentally reach for in the first place.

> **NOT A NEW PROBLEM SOLVED — A PROBLEM THAT WAS NEVER INTRODUCED**
>
> This puts Ruby in the same category as Python 3, not PHP. Django's own Chapter 7 never needed to discuss string-slicing safety at all, because Python 3's `str` type is already Unicode-code-point-aware by design — the exact same reason Ruby doesn't need one here either. Laravel's Chapter 7 genuinely needed `mb_substr()`, because PHP's own core string functions operate on raw bytes by default. Rails doesn't inherit that risk, the same way Django didn't.

The Real Rails-Specific Detail: `utf8mb4` by Default

```
# config/database.yml — generated automatically by
# `rails new --database=mysql` since Rails 5.2
default: &default
  adapter: mysql2
  encoding: utf8mb4
  ...
```

MySQL's own legacy `utf8` charset alias only supports up to 3 bytes per character — enough for most common kanji, but not for every 4-byte Unicode character (rarer CJK extension ideographs, most emoji). Rails has defaulted its own generated `config/database.yml` to the genuinely complete `utf8mb4` encoding automatically since Rails 5.2, specifically because of this exact class of problem. Chapter 1's own `rails new --database=mysql` already produced a configuration with no manual fix needed here — a small, honest, positive finding, not a dramatic one, but a real one.

String Safety, Compared Across Four Languages

	JAVASCRIPT	PYTHON (DJANGO)	PHP (LARAVEL)	RUBY (RAILS)
Default string unit	UTF-16 code units	Unicode code points	Raw bytes	Characters, per the string's own encoding
Slicing risk for common kanji	Low — most CJK ideographs are one UTF-16 unit	None	Real — `substr()` can split a multi-byte character	None
Needed a safe alternative function?	No, for typical CJK content	No	Yes — `mb_substr()`	No

Hands-On Exercises

EXERCISE 1

Explain why kanji's two original constraints from earlier in the series are already resolved for Rails without any new work in this chapter, and name the two specific earlier chapters responsible.

View solution

EXERCISE 2

Explain why Ruby doesn't have a `substr()`-vs-`mb_substr()`-style split the way PHP does, and explain why this chapter compares Ruby to Python rather than treating it as a uniquely Rails-specific resolution.

View solution

EXERCISE 3

Explain what MySQL's legacy `utf8` charset alias actually fails to support, and explain why this course's own `config/database.yml` never needed a manual fix for it.

View solution

CHAPTER 7 QUICK REFERENCE

- **Same resolution as every sibling** — Chapter 2's arbitrary depth, Chapter 4's unescaped rendering
- **Ruby's** `String` — encoding-aware by design; `length` / `[]` / `slice` operate on characters, never raw bytes
- **Closer to Python than PHP** — no `substr()`-vs-`mb_substr()`-style split exists in Ruby, matching Django's own Python 3 strings
- `utf8mb4` **by default** — Rails' own generated `database.yml` has avoided MySQL's legacy 3-byte `utf8` trap automatically since Rails 5.2
- **Next chapter:** Dynamic Content & Forms

Next.js answers this with an implicit Server Action. Django answers it with an explicit view function plus a separate `ModelForm` class. Laravel answers it with a Controller method plus a separate `FormRequest` class. Rails reaches a fourth, genuinely leaner shape: a single Controller action, with Strong Parameters written directly inline — no second class file at all.

The Route

```
# config/routes.rb
patch '/admin/pages/:id/title', to: 'pages#update_title', as: :update_page_title
```

The Controller Action: Strong Parameters, Inline

```ruby
# app/controllers/pages_controller.rb
class PagesController < ApplicationController
  def update_title
    page = Page.find(params[:id])
    page.update!(page_params)
    redirect_to page_path(page.full_path)
  end
private
def page_params
    params.require(:page).permit(:title)
  end
end
```

`params.require(:page).permit(:title)` is Rails' own mass-assignment safety net — it raises if the expected `:page` key is missing, and returns a filtered hash containing only `:title`, silently dropping everything else. This is Strong Parameters, and there's no separate class involved at all: it's a private method on the Controller itself, by convention rather than framework requirement.

> **THE LEANEST OF THE FOUR STRUCTURAL ANSWERS**
>
> Django needed a whole separate `ModelForm` class. Laravel needed a whole separate `FormRequest` class. Rails needs neither — Strong Parameters lives as an ordinary private method on the same Controller class handling the request, the leanest structural shape of the four, not because Rails lacks the capability to separate concerns into classes elsewhere, but because this specific job doesn't call for one by Rails' own convention.

The Deliberate Gap — and Why It Looks Different Here

AN ABSENCE, NOT A VISIBLE PLACEHOLDER

`update_title` has no `before_action :authenticate_admin!` yet — deliberately, matching the same gap every sibling course left open in its own Chapter 8, to be closed in Chapter 9. But Rails' own version of this gap looks structurally different from Laravel's: Laravel's `FormRequest` has a dedicated `authorize()` method that currently returns a visible `true`. Rails' Strong Parameters has no equivalent authorization method at all — `permit` only ever answers "which fields are allowed," never "who is allowed to submit them." Rails' own idiom treats those as genuinely separate concerns, normally joined by an entirely separate `before_action` filter — which is exactly the line that's currently just missing, not present as a placeholder.

CSRF Protection

```
<%= form_with url: update_page_title_path(@page), method: :patch do |f| %>
  <%= f.text_field :title, value: @page.title %>
  <%= f.submit "Save" %>
<% end %>
```

`ApplicationController` includes `protect_from_forgery with: :exception` by default, and Rails' own `form_with` helper automatically embeds a hidden `authenticity_token` field in every generated form — the combination is Rails' direct equivalent of Blade's `@csrf` and Django's `{% csrf_token %}`, requiring no manual token handling in the view at all.

Four Structural Answers, Compared

	NEXT.JS	DJANGO	LARAVEL	RAILS
Mechanism	Implicit Server Action	View function + `ModelForm`	Controller method + `FormRequest`	Controller action + inline Strong Parameters
Separate validation class needed?	No — but no built-in structure either	Yes	Yes	No
Auth/permission concern	Bundled into the action itself	A missing decorator	A visible `authorize()` placeholder	A missing `before_action` filter

Hands-On Exercises

EXERCISE 1

Explain what `params.require(:page).permit(:title)` does, and explain why `page_params` lives as a private method directly on `PagesController` rather than in a separate class file the way Django's and Laravel's own equivalents do.

View solution

EXERCISE 2

Explain what's currently missing from `update_title`, and explain why Rails' own version of this gap is structurally different from Laravel's `authorize() { return true; }` placeholder.

View solution

EXERCISE 3

Explain the two mechanisms responsible for this chapter's CSRF protection, and how they work together without any manual token handling written in the view.

View solution

CHAPTER 8 QUICK REFERENCE

- `params.require(:page).permit(:title)` — Strong Parameters, Rails' own mass-assignment safety net
- **No separate validation class** — the leanest of the four structural answers in this series
- **The gap looks different here** — a missing `before_action` filter, not a visible placeholder method
- `protect_from_forgery` + `form_with` — Rails' own automatic CSRF protection, no manual token handling
- **Next chapter:** Admin Authentication

Laravel's own Chapter 9 found a clean, zero-configuration win: its default hasher already matched the legacy site's bcrypt hash exactly. Rails' `has_secure_password` also defaults to bcrypt — but the honest version of this chapter isn't a repeat of that same clean finding. There's a real, worth-verifying nuance underneath it.

`has_secure_password`

```
# Gemfile
gem "bcrypt", "~> 3.1.7"
```

```
# app/models/admin_user.rb
class AdminUser < ApplicationRecord
  has_secure_password
end
```

`has_secure_password` adds a virtual `password` / `password_confirmation` pair and an `authenticate(password)` method to the model, backed by a real `password_digest` column — expected to already hold the legacy site's existing bcrypt hash for its own admin row.

A Real, Verified Nuance: `$2a$` vs. `$2y$`

NOT ASSUMED AUTOMATICALLY SEAMLESS

Ruby's `bcrypt` gem, when it *generates* a new hash, tags it with the `$2a$` version prefix. PHP's `password_hash()` tags its own bcrypt output `$2y$` — a distinct prefix PHP introduced to mark a historical fix for how certain implementations handled 8-bit characters in a password. In practice, modern bcrypt implementations — including Ruby's own `bcrypt` gem — treat `$2a$`, `$2b$`, and `$2y$` as interchangeable for verification, and an ordinary alphanumeric admin password will authenticate correctly either way. But this is a real point worth checking deliberately when a hash crosses from one language's ecosystem into another — not something to assume automatically seamless the way Laravel's same-ecosystem match was.

Normalizing the Hash on Login

```
if admin&.authenticate(params[:password])
  admin.update(password: params[:password])   # regenerates password_digest, tagged
$2a$
  session[:admin_user_id] = admin.id
end
```

On a successful login, re-assigning `password` triggers `has_secure_password`'s own setter, regenerating `password_digest` using Ruby's own `bcrypt` gem — tagged `$2a$` going forward. This quietly normalizes the legacy PHP-generated hash to a Rails-native one over time, the same spirit as Django Rebuild 9's own automatic hash-upgrade-on-login, closing the version-tag question naturally rather than needing a permanent resolution.

Closing Chapter 8's Gap

```
# app/controllers/pages_controller.rb
class PagesController < ApplicationController
  before_action :require_admin, only: [:update_title]

  private
def require_admin
    redirect_to new_session_path, alert: "Please log in." unless
session[:admin_user_id]
  end
end
```

`before_action :require_admin` is exactly the missing piece Chapter 8 flagged — a genuinely separate concern from Strong Parameters, added the way Rails' own idiom expects: as a callback, not a change to `page_params` itself.

Password Hashing, Four Frameworks

	NEXT.JS	DJANGO	LARAVEL	RAILS
Default algorithm	None — hand-rolled	PBKDF2	Bcrypt	Bcrypt
Matches the legacy hash?	Only via manual code	No — needed reconfiguration	Yes — zero configuration	Yes, functionally — with a real version-tag nuance worth verifying
Session model	Signed JWT cookie	Server-side session	Server-side session	Server-side session

Hands-On Exercises

EXERCISE 1

Explain what `has_secure_password` provides, and name the exact database column it expects to already exist.

View solution

EXERCISE 2

Explain the real, verified difference between the `$2a$` and `$2y$` bcrypt version tags, and explain why this chapter treats Rails' own bcrypt match differently from Laravel's clean zero-configuration finding.

View solution

EXERCISE 3

Explain what `before_action :require_admin` does, and explain exactly which gap from Chapter 8 it closes.

View solution

CHAPTER 9 QUICK REFERENCE

- `has_secure_password` — adds `authenticate`, backed by a `password_digest` column
- **Verified nuance** — Ruby's `bcrypt` gem tags new hashes `$2a$`, PHP tags them `$2y$`; functionally compatible, but a real cross-ecosystem detail worth checking
- **Hash normalization on login** — re-assigning `password` quietly upgrades the legacy hash to Rails' own tag over time
- `before_action :require_admin` — closes Chapter 8's own missing-filter gap
- **Next chapter:** Admin CRUD Interface

Django's admin is a live, framework-owned interface that reads the model at runtime — always current, never a file you'd open and edit. Laravel has nothing free by default at all. Rails reaches a genuine third position: `rails generate scaffold` produces real, ordinary source files — a model, a migration, a Controller with all seven RESTful actions, ERB views — in one command, at generation time. It's "free" the way a starting template is free, not the way a live, self-updating admin panel is free.

What Scaffold Would Generate

```
# illustrative — not run against Page itself, see below
rails generate scaffold AdminNote title:string body:text
```

For a plain, flat resource like this hypothetical `AdminNote`, that one command would generate a working index/show/new/edit/create/update/destroy interface immediately — genuinely useful, genuinely "free" in the one-time sense. But it's not what builds this chapter's own admin interface.

> **WHY `PAGE` ITSELF ISN'T SCAFFOLDED**
>
> Scaffold's generated views assume a flat, non-hierarchical resource. `Page`'s own self-referencing tree — a parent picker, reparenting logic, a delete guard tied to children — has no equivalent in what scaffold produces by default. This is the same reason Django's and Laravel's own admin interfaces both needed hand-written code too: a generic starting point, whether live-introspected or one-time-generated, still runs out exactly where this specific model's own real complexity begins.

The Parent Picker: Same Searchable List

Matching every sibling course's own approach — a plain, client-side-filterable flat list of every other page's `full_path`, rather than a nested tree widget, for the same scale-justified reason established since the Next.js rebuild's own Chapter 10.

Paying Off Chapter 2's Deferred Cost

```ruby
# app/models/page.rb
def move_to(new_parent)
  update!(
    parent: new_parent,
    full_path: new_parent ? "#{new_parent.full_path}/#{slug}" : slug
  )
  update_descendant_paths!
end
def update_descendant_paths!
  children.each do |child|
    child.update!(full_path: "#{full_path}/#{child.slug}")
    child.update_descendant_paths!
  end
end
```

Rails' own version of the same explicit recursive walk both Laravel's `updateDescendantPaths()` and this course's own three predecessors ultimately needed — `move_to` updates the moved page first, then recurses through `update_descendant_paths!` to fix every descendant's own stale `full_path`.

Delete Refusal — a Callback to Chapter 2's Own Finding

```rb
# app/controllers/admin/pages_controller.rb
def destroy
  page = Page.find(params[:id])
  if page.destroy
    redirect_to admin_pages_path
  else
    redirect_to admin_pages_path, alert: page.errors.full_messages.to_sentence
  end
end
```

A TWO-SIDED FINDING, NOT A ONE-SIDED VERDICT

Chapter 2 named `dependent: :restrict_with_error` as an application-level check — a real limitation, since a raw SQL delete could bypass it in a way Laravel's database-level `restrictOnDelete()` never could. Here in Chapter 10, that same application-level design pays a genuine ergonomic dividend: `destroy` simply returns `false` and populates `page.errors` when a child still exists — no `begin`/`rescue` block needed at all. Laravel's Chapter 10 needed a `try`/`catch` around a raw `QueryException` specifically *because* its own protection lived one layer deeper, at the database. The same design choice that was a real weakness in Chapter 2 is a real convenience here — worth naming both sides honestly rather than declaring one approach simply better.

Three Admin Interfaces, Compared

	DJANGO	LARAVEL	RAILS
Free admin?	Yes — live, runtime-introspected	No — Nova/Filament are separate packages	Yes, but one-time — `rails generate scaffold` writes real files
Reparent cascade	Implicit — re-triggered `save()`	Explicit — `updateDescendantPaths()`	Explicit — `update_descendant_paths!`
Delete-with-children handling	Refused, surfaced in the admin	Caught `QueryException`, surfaced manually	Returns `false`, errors already populated — no exception handling needed

Hands-On Exercises

EXERCISE 1

Explain what `rails generate scaffold` produces, and explain why calling it a "one-time free" is a meaningful distinction from Django's own live, runtime-introspected admin.

View solution

EXERCISE 2

Explain why `Page`'s own admin interface isn't built by running `rails generate scaffold` directly against it.

View solution

EXERCISE 3

Explain the two-sided finding about `dependent: :restrict_with_error` — a real limitation back in Chapter 2, but a genuine advantage here — and explain specifically why no `begin`/`rescue` block is needed in this chapter's own `destroy` action.

View solution

CHAPTER 10 QUICK REFERENCE

- `rails generate scaffold` — a genuine third kind of "free," real editable code, generated once
- **Not run against** `Page` **itself** — its tree logic needs hand-written code, matching both siblings' own reasons
- `move_to` **/** `update_descendant_paths!` — the explicit recursive cascade paying off Chapter 2's deferred cost
- **Two-sided finding** — Chapter 2's application-level delete guard was a real limitation there, a real ergonomic win here
- **Next chapter:** Deployment

This chapter covers two genuinely separate stories: Rails' own distinct approach to secrets management, and a real, mechanically different version of this series' own "modernize in place" throughline.

Secrets: A Reversed Philosophy

```
rails credentials:edit
```

> **THE OPPOSITE OF THE .ENV PATTERN**
>
> Every sibling course kept its own secrets in a `.env` file, deliberately excluded from the repository via `.gitignore`. Rails takes the reverse approach: `config/credentials.yml.enc` holds the actual secrets, *encrypted*, and is committed to the repository as ordinary source. `config/master.key` — the one file that can decrypt it — is what's excluded from the repo instead. Rather than keeping the secrets themselves out of version control, Rails keeps them in, encrypted, and protects only the single key that unlocks them.

`RAILS_MASTER_KEY` is set as a real environment variable on the production server, letting the running application decrypt `credentials.yml.enc` at boot — the deployment's own equivalent of the legacy site's admin credential needing to be present somewhere, just moved to protecting one small key file instead of an entire secrets file.

Allowed Hosts — A Genuine Parallel, Not a Difference

```
# config/environments/production.rb
config.hosts << "osztromok.com"
```

Rails' own `config.hosts` is a genuine, direct parallel to Django's `ALLOWED_HOSTS` — both protect against HTTP Host header attacks by rejecting requests for domains the application doesn't recognize as its own. Worth naming honestly as a real similarity, not manufactured as a difference.

Deployment: Phusion Passenger's Apache Module

```
# Apache VirtualHost — the site's own already-running Apache
<VirtualHost *:443>
    ServerName osztromok.com
    DocumentRoot /var/www/rails-rebuild/public

    PassengerEnabled on
    PassengerRuby /usr/bin/ruby3.2

    SSLEngine on
    SSLCertificateFile /etc/letsencrypt/live/osztromok.com/fullchain.pem
    SSLCertificateKeyFile /etc/letsencrypt/live/osztromok.com/privkey.pem
</VirtualHost>
```

`PassengerEnabled on` and `PassengerRuby` are essentially the entire configuration. There's no separate reverse-proxy directive to write.

A DIFFERENT MECHANISM, NOT JUST A DIFFERENT CONFIG FILE

Laravel's own `mod_proxy_fcgi` reuse still *proxies* — Apache forwards PHP requests over a socket to an independently-running PHP-FPM process pool, a real separate process Apache talks to. Phusion Passenger's own Apache module (`mod_passenger`) works differently: it embeds Ruby process spawning and supervision directly into Apache's own module system, with Apache itself managing the Rails application processes as a native extension of itself, no separate proxy hop involved. Both reuse the site's own already-live Apache, giving this course its own honest version of the "modernize in place" story — but the two are genuinely different mechanisms, not the same idea in a different config syntax.

NAMED HONESTLY: NOT THE MORE COMMON MODERN CHOICE

The more common production setup for a modern Rails application is standalone Puma (Rails' own default app server, bundled in the Gemfile already) running behind nginx as a reverse proxy — genuinely the more typical path most current Rails deployment guides recommend. Passenger is chosen here specifically because it lets this chapter continue the same "reuse what's already live" throughline as Laravel's own deployment chapter, not because it's the field's dominant default.

Assets & Migrations

```
RAILS_ENV=production rails assets:precompile
RAILS_ENV=production rails db:migrate
```

`assets:precompile` runs Propshaft's own fingerprinting step, established in Chapter 5 — no separate Node/npm build pipeline to coordinate, exactly as that chapter's own forward reference promised.

Four Frameworks' Secrets & Deployment, Compared

	NEXT.JS / DJANGO / LARAVEL	RAILS
Secrets storage	Plain `.env`, excluded from the repo	Encrypted `credentials.yml.enc`, committed to the repo; only the key is excluded
Reused Apache reverse-proxy mechanism	N/A / N/A / `mod_proxy_fcgi`, proxying to PHP-FPM	`mod_passenger`, embedding process management directly into Apache

Hands-On Exercises

EXERCISE 1

Explain what `config/credentials.yml.enc` and `config/master.key` are, and explain why Rails' own secrets-management approach is the reverse of the `.env`-file pattern every sibling course used.

View solution

EXERCISE 2

Explain the real mechanical difference between Phusion Passenger's Apache module and Laravel's own `mod_proxy_fcgi` reuse of the site's already-live Apache.

View solution

EXERCISE 3

Explain why this chapter names Puma-behind-nginx honestly as the more common modern Rails deployment choice, rather than presenting Passenger as simply the correct or superior option.

View solution

CHAPTER 11 QUICK REFERENCE

- `config/credentials.yml.enc` + `config/master.key` — encrypted secrets committed to the repo; only the key stays out
- `config.hosts` — a genuine parallel to Django's `ALLOWED_HOSTS`, not a difference
- `PassengerEnabled on` — embeds Ruby process management directly into Apache, mechanically different from proxying to PHP-FPM
- **Honest alternative** — Puma-behind-nginx is the more common modern choice; Passenger is used here for the "modernize in place" throughline specifically
- `rails assets:precompile` — Chapter 5's Propshaft payoff, no separate JS build step needed
- **Next chapter:** Capstone — A Complete, Working Flexible-Routing Site

Website Rebuild with Ruby on Rails

Chapter 12 · Capstone: A Complete, Working Flexible-Routing Site

Sam — the same site admin persona from every capstone in this series, reused a fourth and final time — logs into this course's own deployed Rails version. Every step below draws on a specific earlier chapter, closing with an attribution table, an honest scope note, and the close of the entire four-framework Website Rebuild series.

STEP 1 — VISITING THE LIVE DEPLOYMENT

Sam opens the site. Phusion Passenger's Apache module is handling the request directly, embedded in the same Apache process that's always run this domain. `RAILS_MASTER_KEY` on the server decrypts `config/credentials.yml.enc` at boot.

STEP 2 — LOGGING IN

At the login form, Sam enters the legacy admin credential. `AdminUser#authenticate` — from `has_secure_password` — verifies it against the stored bcrypt hash, functionally compatible across the `$2a$` / `$2y$` version-tag difference for an ordinary password. A successful login quietly regenerates the hash under Ruby's own `$2a$` tag.

STEP 3 — BUILDING THE FIVE-LEVEL CHAIN

Through the hand-built admin interface, Sam creates `programming`, then `general-purpose-languages`, then `java`, then `fundamentals`, then `chapter-1` — five real levels deep. Each save triggers the `before_save :compute_full_path` callback from Chapter 2, recomputing that page's own path from its new parent.

STEP 4 — VIEWING THE PAGE

Visiting the full five-segment path, Chapter 3's `get '*path'` glob route resolves it in one line, Chapter 4's layout and `breadcrumb_for` helper render the page with its ancestry, and Chapter 5's Propshaft-served dark theme applies with no build step involved.

STEP 5 — EDITING THE TITLE, NOW ACTUALLY PROTECTED

Sam edits the Chapter 1 page's title. Chapter 8's `update_title` action and Strong Parameters handle the submission — but this time, Chapter 9's `before_action :require_admin` is in place, checking `session[:admin_user_id]` before the action runs at all. Logged out, the identical request would now redirect to the login page instead.

STEP 6 — REORGANIZING: PAYING THE REAL CASCADE COST

Sam decides `fundamentals` belongs under a different parent. Chapter 10's `move_to` updates `fundamentals`' own `full_path` first, then `update_descendant_paths!` recurses down and updates `chapter-1`'s path too — the real payment of the cascade cost Chapter 2 deliberately deferred at the very start of the course.

STEP 7 — ATTEMPTING TO DELETE A PAGE WITH CHILDREN

Sam tries to delete `java`, which still has `fundamentals` beneath it. Chapter 2's `dependent: :restrict_with_error` refuses the operation, and Chapter 10's `destroy` action reads the resulting `page.errors` directly — no exception-handling code needed, the genuine ergonomic payoff of the same application-level design that was named as a real limitation back in Chapter 2.

STEP 8 — CONFIRMING THE KANJI MIGRATION HELD UP

Sam visits the kanji page. Chapter 7's resolution holds: Ruby's own character-aware `String` methods never risked corrupting the multi-byte character, and the database's `utf8mb4` encoding — defaulted automatically by `rails new --database=mysql` since Rails 5.2 — stores it correctly with no manual configuration ever needed.

STEP 9 — A QUICK N+1 SANITY CHECK

Sam checks the server log while loading the breadcrumb-heavy Chapter 1 page. Chapter 6's `.includes()` is doing its job — batched queries instead of one per tree level, the same fix every sibling course's own database chapter applied to the identical deliberately-planted problem.

STEP 10 — CONFIRMING THE DEPLOY ITSELF IS HEALTHY

Sam confirms `assets:precompile`'s output is being served correctly, and that Passenger is spawning Rails processes cleanly under Apache — the last confirmation in a series that's now rebuilt the same real site four separate times.

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1. Visiting the deployment	11 — Deployment
2. Logging in	9 — Admin Authentication
3. Building the five-level chain	2 — URL & Content Model, 10 — Admin CRUD
4. Viewing the page	3 — Routing, 4 — Views: ERB, 5 — Styling
5. Editing the title	8 — Dynamic Content & Forms, 9 — Admin Authentication
6. Reorganizing	2 — deferred cascade cost, 10 — `move_to`
7. Delete refusal	2 — `dependent: :restrict_with_error`, 10 — Admin CRUD
8. Kanji migration	7 — Rendering Content & the Kanji Edge Case
9. N+1 sanity check	6 — ActiveRecord ORM
10. Deploy health check	5 — Styling, 11 — Deployment

Honest Scope Note

WHAT THIS COURSE DELIBERATELY DOESN'T COVER

- No multi-admin roles or permission levels — a single legacy admin credential only
- No revision history or audit log of content edits
- No media/image upload handling
- No automated tests or CI pipeline
- No internationalization beyond the kanji character-storage edge case itself
- No rate limiting or lockout policy on the login form

RAILS' OWN HONEST CONTRIBUTIONS TO THIS SERIES

Two genuinely Rails-specific findings, grounded in real, verified facts rather than manufactured uniqueness: Ruby's own open-class design underneath ActiveSupport (Chapter 1), and the reversed `credentials.yml.enc` / `master.key` secrets philosophy (Chapter 11) — both real, checkable technical facts, not restatements of what Laravel already shared with Rails elsewhere in this course.

The Website Rebuild Series Is Now Complete

Four frameworks, one real site, rebuilt four separate times: Next.js, Django, Laravel, and now Ruby on Rails. Each course found its own genuine advantages and honest limits rather than declaring any single architecture simply "the best" — the same standard the Food Tracker Quartet set for comparative courses on this site.

Hands-On Exercises

EXERCISE 1

Trace Step 6's reorganization through the code: what does `move_to` update first, and what does `update_descendant_paths!` update afterward?

View solution

EXERCISE 2

Explain what changed between Step 5 in this capstone and the identical-looking action back in Chapter 8, and name the exact line of code responsible for the difference.

View solution

EXERCISE 3

Name two genuinely Rails-specific findings from this course — not shared with Laravel or Django — and explain the real technical fact each one is grounded in.

View solution

COURSE COMPLETE

- **12/12 chapters** — Website Rebuild with Ruby on Rails is now complete
- **Standout findings named honestly throughout** — Ruby's open classes (Ch1), `includes()`'s dynamic strategy (Ch6), reversed secrets philosophy (Ch11)
- **Real limits named honestly too** — application-level (not database-level) delete protection (Ch2), Passenger as the honest minority choice (Ch11)
- **Fourth and final course in the Website Rebuild series** — alongside Next.js, Django, and Laravel
- **The entire Website Rebuild series is now complete**