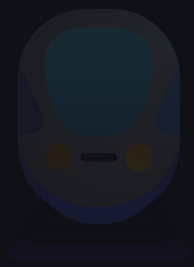

RUBY ON RAILS

Fundamentals

Getting Started · Routing · Controllers & Actions

Views & ERB · Active Record Basics

Active Record Associations · Validations & Callbacks

Forms & Strong Parameters

Philip Osztromok

Generated with Claude

Table of Contents

Ruby on Rails Fundamentals — 8 Chapters

🚂 Getting Started with Rails

Ruby on Rails is a full-featured web framework built directly on top of everything covered in the two Ruby courses — blocks, modules, metaprogramming, and Active Record's heavy use of the block/mixin mechanics from Ruby Course 1 all show up again here. Rails' defining philosophy is **convention over configuration**: an enormous amount of what Express required you to wire up by hand — routing, folder structure, the controller/model split — Rails generates and assumes automatically, as long as you follow its naming conventions.

📦 Installing Rails and Creating a New App

```
$ gem install rails   # Rails is itself just a gem -- Ruby Course 2, Chapter 7
$ rails new blog_app
$ cd blog_app
$ rails server        # starts the dev server at http://localhost:3000
```

`rails new blog_app` does something Express never does on its own: it generates a complete, working application skeleton — folders, a database configuration, a `Gemfile` with sensible defaults, and a functioning (if empty) app you can immediately run. Compare this to Express's `npm init` plus manually installing and wiring together every piece.

📁 The Generated Directory Structure

```
blog_app/
├── app/
│   ├── controllers/
│   ├── models/
│   ├── views/
│   └── assets/
├── config/
│   ├── routes.rb
│   └── database.yml
├── db/
│   ├── migrate/
│   └── schema.rb
├── Gemfile
└── Gemfile.lock
```

Every one of these folders already exists, already has a defined purpose, and Rails' generators (previewed at the end of this chapter) know exactly where new code belongs. `app/controllers`, `app/models`, and `app/views` aren't a structure you invent — they're the framework's built-in **M**odel-**V**iew-**C**ontroller layout, present the moment `rails new` finishes running.

⚖️ Convention Over Configuration

This is Rails' central idea, and it's worth stating plainly: if you follow Rails' naming conventions, an enormous number of decisions get made for you automatically, with no explicit configuration required.

Naming implies wiring

A controller named `PostsController` automatically lives at `app/controllers/posts_controller.rb`, automatically renders views from `app/views/posts/`, and — once Chapter 2 covers `resources :posts` — automatically gets a full set of RESTful routes, with zero additional configuration.

Database tables, inferred

A model named `Post` automatically maps to a database table named `posts` (pluralized, snake_case) — Active Record (Chapter 5) infers this mapping rather than requiring it to be spelled out anywhere.

MVC: Baked In vs Hand-Assembled

Recall Express Course 1's final chapter, which walked through manually organizing a Controller-Service-Model structure: a hand-written `src/controllers/`, `src/services/`, `src/models/`, and `src/routes/`, each layer wired together explicitly by you, one `require` and one router mount at a time. Rails' `app/controllers`, `app/models`, and `app/views` aren't something you build — they exist from the first command.

Project Structure: Express vs Rails

Concern	Express (hand-assembled)	Rails (built in)
Where does this exist?	You design and create it yourself	Generated automatically by rails new
Controllers	src/controllers/ (your convention)	app/controllers/ (Rails' convention)
Models / data access	src/models/ (your convention)	app/models/ (Rails' convention, Active Record built in)
Business logic layer	src/services/ — Express has no opinion, you invented this layer	No built-in equivalent — Rails traditionally puts logic in models ("fat models"); a services layer is a common addon pattern, not a framework default
Route wiring	Manual router.use() mounting per resource	config/routes.rb, often one resources line per resource

⚠ Convention over configuration cuts both ways

Express's explicitness means every piece of wiring is visible somewhere in your own code — nothing happens that you didn't write. Rails' conventions mean a lot happens that you *didn't* write, inferred purely from a file's name and location. This is faster once you know the conventions, and genuinely confusing before you do — "magic" is the word often used, sometimes as praise, sometimes as a complaint. Expect an adjustment period where things work and you're not entirely sure why, before the conventions click.

🏭 A First Taste of Generators

```
$ rails generate controller Welcome index
    create  app/controllers/welcome_controller.rb
    create  app/views/welcome/index.html.erb
    route   get "welcome/index"
```

`rails generate` (often shortened to `rails g`) scaffolds files following Rails' conventions instantly — a controller file, a matching view, and even a route entry, all created and correctly wired together in one command. This is the generator mechanism Chapters 2 through 8 lean on repeatedly for routes, models, and migrations.

📜 Starting a Project: Express vs Rails

Side by Side

Step	Express	Rails
Create the project	npm init, then manually install express	rails new blog_app
Project structure	You design it (express1-8 walked through one approach)	Generated automatically, following Rails conventions
Start the dev server	node server.js (or nodemon)	rails server
Scaffold a new resource	Manually create controller/service/model/route files	rails generate controller/model/scaffold

💻 Coding Challenges

Challenge 1: Tour the Generated App

Run `rails new library_app` (or read through the structure shown in this chapter if you don't have Rails installed locally) and write one sentence each describing the purpose of `app/`, `config/`, `db/`, and the `Gemfile`.

Goal: Build a working mental map of where things live before writing any actual Rails code.

→ Solution

Challenge 2: Map Express's Structure Onto Rails

Using the Express project structure from this chapter's comparison table (`src/controllers`, `src/services`, `src/models`, `src/routes`), write down which Rails folder each one maps to — and explicitly note which one has no direct Rails equivalent, and why.

Goal: Practice translating a structure you already know into Rails' conventions.

→ Solution

Challenge 3: Generate a Controller

Write the exact command to generate a controller called `Products` with two actions, `index` and `show`. List every file (and route entry) you'd expect Rails to create as a result, based on the pattern shown in this chapter's Welcome example.

Goal: Predict a generator's output before running it, based on the naming conventions covered so far.

→ Solution

💎 The Trade This Whole Course Is About

Nearly every chapter from here on is some version of the same trade: Rails gives up some of Express's explicitness in exchange for dramatically less boilerplate, as long as you're building something that fits its conventions. That's a genuinely different bet than Express's philosophy — neither is simply "better," and this course will keep pointing out exactly where that trade shows up, starting with routing in the next chapter.

🎯 What's Next

Next chapter: **Routing** — `config/routes.rb`, and how a single `resources :posts` line generates all seven RESTful routes Express would otherwise need defined one at a time.

Routing

Chapter 1's generator briefly added a single route entry as a side effect. This chapter covers `config/routes.rb` properly — and specifically the single line, `resources :posts`, that replaces an entire file's worth of Express route definitions with one declaration.

config/routes.rb

```
Rails.application.routes.draw do
  get "welcome/index"
# more routes go here
end
```

Every route in a Rails app is declared inside this one file, inside the `draw` block — a single source of truth, unlike Express's routes potentially spread across multiple router files mounted together (as Chapter 1's comparison showed).

⚡ resources :posts — Seven Routes, One Line

```
Rails.application.routes.draw do
  resources :posts
end
```

This single line generates all seven conventional RESTful routes for the `Post` resource. Run `rails routes` to see exactly what got created:

```
$ rails routes -c posts

   Prefix   Verb    URI Pattern            Controller#Action
    posts   GET     /posts                 posts#index
            POST    /posts                 posts#create
 new_post   GET     /posts/new             posts#new
edit_post   GET     /posts/:id/edit        posts#edit
     post   GET     /posts/:id             posts#show
            PATCH   /posts/:id             posts#update
            PUT     /posts/:id             posts#update
            DELETE  /posts/:id             posts#destroy
```

Seven distinct actions, mapped to conventional URLs and HTTP verbs, generated entirely from one word: `:posts`. Rails infers the controller name (`PostsController`, from Chapter 1), the pluralization, and every URL pattern from that single symbol.

📜 The Same Seven Routes — Rails vs Express

One Line vs a Full File

Express (from Chapter 1's example)	Rails

router.route('/').get(controller.list).post(controller.create);
router.route('/:id').get(controller.show).patch(controller.update).delete(controller.remove);
(plus new/edit routes, if the app renders server-side forms)
	resources :posts

Express requires every verb/path/handler combination spelled out explicitly — genuinely more visible and traceable, but genuinely more typing, and easy to have drift out of sync with REST conventions over time. Rails' single line is faster to write and impossible to get inconsistent, at the cost of needing to already know what those seven routes actually are.

🔗 Named Route Helpers

```
posts_path      # => "/posts"
post_path(5)    # => "/posts/5"
new_post_path   # => "/posts/new"
edit_post_path(5) # => "/posts/5/edit"
```

`resources :posts` also generates named path helpers, visible as the `Prefix` column in the `rails routes` output above. Instead of hardcoding `"/posts/#{id}/edit"` as a string throughout the app — the way you'd typically build a URL in Express, string-interpolating a path directly — Rails gives you a real method call. Rename the route later, and every call site using the helper updates automatically; every hardcoded string wouldn't.

🪆 Nested Resources

```
Rails.application.routes.draw do
  resources :posts do
    resources :comments # nested under posts
end
end
# generates, among others:
# GET  /posts/:post_id/comments     comments#index
# POST /posts/:post_id/comments     comments#create
```

Nesting `resources :comments` inside `resources :posts` generates comment routes scoped under a specific post's ID — the Rails equivalent of Express's `router.use('/posts/:postId/comments', commentsRouter)` pattern, again collapsed to a declarative block instead of manual router mounting.

🎯 Limiting and Customizing Routes

```
resources :posts, only: [:index, :show]      # just these two, nothing else
resources :posts, except: [:destroy]       # all seven except deleting

resources :posts do
 member do
  post "publish" # POST /posts/:id/publish -- acts on ONE post
end
 collection do
  get "drafts" # GET  /posts/drafts     -- acts on the whole collection
end
end
```

⚠ Custom, non-RESTful routes need more ceremony, not less

Here's a genuine reversal of this course's usual pattern: adding one extra Express route is a single `router.post(...)` line, no ceremony at all. Adding a non-standard Rails route requires understanding `member` vs `collection` blocks first — `member` for routes acting on one specific resource (needs an `:id`), `collection` for routes acting on the resource as a whole (no `:id`). Rails' conventions make the *common* case (the seven standard actions) dramatically shorter, but the moment you step outside them, you're learning Rails-specific concepts Express simply doesn't require.

💻 Coding Challenges

Challenge 1: Generate and List Routes

Write the single `routes.rb` line for a fully RESTful `articles` resource, then write out, in a table or list, all seven routes you'd expect `rails routes` to show — verb, path, and controller#action for each.

Goal: Memorize the seven standard REST routes well enough to predict them without running the command.

→ Solution

Challenge 2: Convert Express Routes to Rails

Given this Express setup — `router.get('/products', ...)`, `router.get('/products/:id', ...)`, `router.post('/products', ...)` only, with no update or delete routes at all — write the equivalent single-line Rails `routes.rb` entry using `only:` to match exactly this reduced set of actions.

Goal: Practice recognizing when `only:`/`except:` is the right tool rather than a bare `resources` line.

→ Solution

Challenge 3: Nested Resources Plus a Member Route

Write a `routes.rb` that nests `resources :reviews` under `resources :products`, and adds a custom member route, `POST /products/:id/feature`, that maps to a `feature` action on `ProductsController`.

Goal: Combine nested resources and a custom member route in one realistic routes file.

→ Solution

💎 Where the "Magic" Actually Pays Off

Chapter 1 warned that convention over configuration can feel confusing before it clicks. Named route helpers are a clean, concrete example of where it pays off clearly: `edit_post_path(@post)` reads as intent ("the edit URL for this post") rather than a hardcoded, easy-to-typo string, and it stays correct automatically if the underlying URL structure ever changes. This is the kind of small, compounding ergonomic win the rest of this course will keep surfacing.

🎯 What's Next

Next chapter: **Controllers & Actions** — writing the seven conventional action methods routing actually calls, working with `params`, and how Rails' controller layer compares to Express's request handlers.

🎮 Controllers & Actions

Chapter 2's `resources :posts` pointed seven routes at seven controller actions. This chapter writes those actions — and covers the single biggest behavioral surprise coming from Express: a Rails action doesn't have to explicitly send a response at all.

🏛️ ApplicationController

```
class PostsController < ApplicationController
# actions go here
end
```

Every Rails controller inherits from `ApplicationController`, itself a subclass of `ActionController::Base` — real Ruby classes using the inheritance and class syntax from both Ruby courses. Express has no equivalent concept at all: a "controller" there (as Chapter 1's example showed) is just a plain object of exported functions, with no shared base class or built-in framework hooks.

🎬 The Seven Actions

```
class PostsController < ApplicationController
def index
@posts = Post.all            # Active Record -- fully covered in Chapter 5
end
def show
@post = Post.find(params[:id])
 end
def new
@post = Post.new
 end
def create
@post = Post.new(post_params)
  if @post.save
    redirect_to @post
else
    render :new
end
end
def edit
@post = Post.find(params[:id])
 end
def update
@post = Post.find(params[:id])
  if @post.update(post_params)
    redirect_to @post
else
    render :edit
end
end
def destroy
Post.find(params[:id]).destroy
  redirect_to posts_path   # the named route helper from Chapter 2
end
end
```

Every method name matches an action from Chapter 2's route table exactly — that's not a coincidence, it's the same naming convention linking routes to controllers that Chapter 1 introduced.

👻 Implicit Rendering — the Biggest Surprise Coming From Express

⚠ index and show above never call render or res.json — and that's correct

In Express, forgetting to call `res.send`/`res.json`/`res.render` leaves the request hanging with no response at all — the framework never assumes what you meant to send back. Rails does the opposite: if an action doesn't explicitly call `render` or `redirect_to`, Rails **automatically** renders the view matching the action's name — `index` implicitly renders `app/views/posts/index.html.erb`, `show` implicitly renders `app/views/posts/show.html.erb`. This is convention over configuration applied to the controller/view boundary itself, and it's genuinely disorienting the first time a page renders correctly despite the action "doing nothing" to send a response.

📥 params — One Hash for Everything

```ruby
# GET /posts/5?highlight=true
def show
  params[:id]        # => "5"        -- from the route (:id in the URL pattern)
  params[:highlight] # => "true"     -- from the query string
end
# POST /posts, with a form body of { post: { title: "Hi" } }
def create
  params[:post][:title]  # => "Hi"       -- from the request body
end
```

Rails merges route parameters, query string parameters, and request body parameters into a single `params` hash-like object — no need to know or care which source a given value came from. Express keeps these genuinely separate: `req.params` (route), `req.query` (query string), and `req.body` (parsed body, requiring middleware like `express.json()` to populate at all).

🔀 render vs redirect_to

render — same request, different content

Renders a view (or a different one than the action's default) for the *current* request — no new HTTP round trip, the URL in the browser doesn't change. Used above in `create`'s failure branch, to redisplay the form with validation errors still in place.

redirect_to — a brand new request

Sends an HTTP redirect response, telling the browser to make an entirely new GET request to a different URL — the browser's address bar changes. Used above in `create`'s success branch, sending the user to the new post's `show` page.

This maps directly onto Express's `res.render(...)` vs `res.redirect(...)` — same underlying HTTP distinction, just spelled with Rails' own method names.

📜 Handling a Request: Express vs Rails

Side by Side

Concept	Express	Rails
Handler shape	A plain function: (req, res) => { … }	An instance method inside a class inheriting ApplicationController
Route params	req.params	params (merged with query + body)
Query string	req.query	params (same hash, no separate access)
Request body	req.body (needs body-parsing middleware)	params (parsed automatically)
Must you send a response?	Yes, always, explicitly	No — implicit rendering falls back to a matching view

💻 Coding Challenges

Challenge 1: Write a Full Controller

Write a `CommentsController` with all seven actions, following the same pattern as this chapter's `PostsController` (you can use placeholder Active Record calls like `Comment.find(params[:id])` even before Chapter 5 covers Active Record in depth). Comment above `index` and `show` explaining specifically which view file Rails will implicitly render for each.

Goal: Get the shape of all seven action methods, and the implicit-render convention, under your fingers.

→ Solution

Challenge 2: Reading From params

Given a request to `GET /posts/12/comments?sort=newest`, with route `resources :posts { resources :comments }` (Chapter 2), write the `index` action for `CommentsController` and show exactly how you'd read both the post's ID and the `sort` query parameter out of `params`.

Goal: Practice pulling values from Rails' unified params hash regardless of where they originated.

→ Solution

Challenge 3: render vs redirect_to

Write an `update` action for a `ProductsController` that, on a successful save, redirects to the product's `show` page, and on failure, re-renders the `:edit` view so the user sees their validation errors. Add a comment explaining what would go wrong (from the user's perspective) if `redirect_to` were used in the failure branch instead of `render`.

Goal: Internalize exactly when each of the two response methods is the correct choice.

→ Solution

💎 Implicit Rendering Is Convention Over Configuration's Purest Form

No routing table, no folder structure — just a controller action's *name* silently determining what gets rendered if you don't say otherwise. It's the single clearest example in this course so far of Chapter 1's warning made concrete: extremely little code, and genuinely confusing the first time a page renders correctly despite an action that appears to do nothing at all.

🎯 What's Next

Next chapter: **Views & ERB** — the templates those implicitly-rendered actions actually render, embedding Ruby directly in HTML, and how ERB compares to the EJS templating already covered in Express.

📝 Views & ERB

Chapter 3's implicit rendering pointed at a view file by convention — this chapter is what's actually inside it. Rails' templating language, **ERB** (Embedded Ruby), turns out to share a surprising amount of syntax with the EJS templates already covered in Express, which makes this one of the more comfortable transitions in the whole course.

🧬 ERB Tags — <%= %> vs <% %>

```
<!-- app/views/posts/show.html.erb -->
<h1><%= @post.title %></h1>

<% if @post.published? %>
  <p>Published</p>
<% end %>
```

ERB vs EJS — A Genuine Syntax Convergence

Purpose	EJS (Express)	ERB (Rails)
Output a value (escaped)	<%= value %>	<%= value %>
Run logic, no output	<% if (x) { %>	<% if x %>
File extension	.ejs	.html.erb

This is worth pausing on: `<%= %>` for output and `<% %>` for logic-only is **identical syntax** between EJS and ERB — a genuine convergence, similar in spirit to the spaceship operator and safe navigation operator syntax Ruby Course 2 flagged converging into PHP. The main practical difference: ERB's logic tags use Ruby's own `if`/`end` block syntax (Ruby Course 1, Chapter 3) instead of EJS's JavaScript curly braces.

Instance Variables Arrive Automatically

```
# PostsController
def show
@post = Post.find(params[:id])
end
<!-- app/views/posts/show.html.erb -- @post is just... available -->
<h1><%= @post.title %></h1>
```

Recall Chapter 3's `@post` — every instance variable set in a controller action is automatically available inside that action's view, with no explicit data-passing step. Compare this to Express, where `res.render("show", { post })` requires the controller to explicitly build and pass a data object into the template every single time.

🖼️ Layouts

```
<!-- app/views/layouts/application.html.erb -->
<!DOCTYPE html>
<html>
 <head><title>My Blog</title></head>
 <body>
  <nav>...</nav>
  <%= yield %> <!-- the current view's content gets inserted here -->
 </body>
</html>
```

`yield` here is the same keyword from Ruby Course 1 Chapter 4's blocks — the layout is, in effect, a method that yields to whichever view is currently being rendered. Every view in the app is automatically wrapped in `application.html.erb` with zero effort on the individual view's part.

⚠ EJS has no automatic equivalent of this

Express's EJS setup (Chapter 1's comparisons already flagged this pattern) typically requires each template to explicitly `<%- include('partials/header') %>` and `<%- include('partials/footer') %>` itself — shared page structure is opt-in, template by template. Rails' layout wraps *every* view automatically; you'd have to explicitly opt **out** (`layout false`) rather than opt in.

🧩 Partials

```
<!-- app/views/posts/_form.html.erb -- leading underscore names a partial -->
<%= form_with model: post do |f| %>
  <%= f.text_field :title %>
<% end %>

<!-- app/views/posts/new.html.erb -->
<h1>New Post</h1>
<%= render "form", post: @post %> <!-- note: no leading underscore when referencing it -->
```

Partials are reusable view fragments, named with a leading underscore on disk but referenced without it. `render "form", post: @post` passes `post` in explicitly as a **local variable** to the partial — this is the conceptual equivalent of EJS's `include()`, just with Rails' own naming convention (the underscore) marking a file as a partial rather than a full view.

🛠️ View Helpers

```
<%= link_to "Edit", edit_post_path(@post) %>
<!-- generates: <a href="/posts/5/edit">Edit</a> -->
```

`link_to` is a real Ruby method call — using the named route helpers from Chapter 2 — that generates the HTML for you, rather than hand-writing an `<a href="...">` tag with a string-interpolated URL the way an EJS template typically would. Dozens of similar helpers exist for forms, dates, and more.

🛡️ Automatic Escaping

```
@comment = "<script>alert('xss')</script>"

<%= @comment %>        <!-- output is escaped -- renders as inert TEXT, not executed -->
<%= raw(@comment) %>   <!-- explicitly opt out -- renders as real, executable HTML -->
```

`<%= %>` automatically HTML-escapes its output by default — the exact XSS protection covered in depth in the completed XSS security course, here built into the templating layer itself rather than something you have to remember to apply. EJS's `<%= %>` auto-escapes too (a real parallel), while EJS's separate `<%- %>` tag opts out; Rails instead keeps one output tag and requires an explicit `raw(...)` call (or `.html_safe`) to bypass escaping.

📜 Templating, Side by Side

EJS vs ERB

Concept	EJS (Express)	ERB (Rails)
Passing data in	Explicit object in res.render(view, data)	Automatic — controller's instance variables
Shared layout	Opt-in via include() in each template	Automatic — every view wraps in the layout by default
Reusable fragment	include('partials/name')	render "name", locals passed explicitly
Escape by default?	Yes, with <%= %> (opt out via <%- %>)	Yes, with <%= %> (opt out via raw()/.html_safe)

💻 Coding Challenges

Challenge 1: An index View

Write `app/views/posts/index.html.erb` that loops over `@posts` (an array of Post objects, each with `.title` and `.published?`) using `<% %>`, and for each one outputs the title with `<%= %>`, only showing a "Published" label next to posts where `.published?` is true.

Goal: Practice correctly distinguishing output tags from logic-only tags in a realistic loop.

→ Solution

Challenge 2: A Shared Navbar Partial

Write a partial `app/views/shared/_navbar.html.erb` containing a simple nav with two `link_to` calls, then show the single line you'd add to `application.html.erb` (above the `yield`) to render it on every page.

Goal: Practice the underscore-on-disk, no-underscore-when-referenced partial convention.

→ Solution

Challenge 3: Escaping vs raw

Given `@bio = "Bold claim"` set in a controller, write two lines in a view: one using plain `<%= %>` and one using `raw(...)`. Add a comment describing exactly how each would actually render in the browser, and which one you'd use for genuinely untrusted, user-submitted content.

Goal: Be certain, not just aware, of when escaping matters.

→ Solution

💎 Secure by Default, Not by Discipline

The XSS course already covered why unescaped user content is dangerous. What's worth noticing here is the framework-level design choice: Rails makes the *safe* behavior (`<%= %>`, escaped) the default and the *dangerous* behavior (`raw(...)`) something you have to explicitly, visibly opt into. That's a meaningfully different security posture than a templating setup where escaping is something a developer has to remember to apply correctly every time.

🎯 What's Next

Next chapter: **Active Record Basics** — migrations, models, and how `Post.find(params[:id])`, used casually throughout this chapter and the last, actually talks to the database.

🗄️ Active Record Basics

Every chapter since Chapter 3 has used `Post.find(params[:id])` or similar without explaining it. This chapter finally does: **Active Record** is Rails' built-in ORM (Object-Relational Mapper), and it replaces essentially everything Express's `mysql2` integration required you to write by hand.

 Migrations — Versioned Schema Changes

```
$ rails generate migration CreatePosts title:string body:text published:boolean


# db/migrate/20260101000000_create_posts.rb
class CreatePosts < ActiveRecord::Migration[7.1]
 def change
   create_table :posts do |t|
    t.string  :title
    t.text    :body
    t.boolean :published, default: false
    t.timestamps   # adds created_at and updated_at automatically
end
end
end

$ rails db:migrate
```

A **migration** is a single, ordered, timestamped Ruby file describing one schema change — running `rails db:migrate` applies every migration that hasn't run yet, in order. This is the direct Rails equivalent of manually writing and tracking `CREATE TABLE`/`ALTER TABLE` SQL yourself against a `mysql2` connection, with the ordering and "has this already run?" bookkeeping handled automatically instead of by convention you maintain yourself.

📋 schema.rb — The Combined Snapshot

```
# db/schema.rb (auto-generated, never edited by hand)
ActiveRecord::Schema[7.1].define(version: 2026_01_01_000000) do
 create_table "posts" do |t|
  t.string  "title"
  t.text    "body"
  t.boolean "published", default: false
  t.datetime "created_at"
  t.datetime "updated_at"
end
end
```

`schema.rb` is automatically regenerated after every migration — a single, current source of truth for the entire database structure. Express has no direct equivalent at all; without a separate schema-tracking tool, the actual structure of a `mysql2`-backed database lives only in the database itself, discoverable only by connecting and inspecting it directly.

🎭 Models — Almost Suspiciously Empty

```
# app/models/post.rb
class Post < ApplicationRecord
end
```

⚠ Yes, that's the entire model — no column definitions anywhere

This is one of Active Record's biggest "magic" moments: `Post` automatically knows it has `title`, `body`, `published`, `created_at`, and `updated_at` attributes — Rails introspects the actual database schema at runtime and generates matching accessor methods automatically. There's no column list to keep in sync inside the model file itself; the migration (and by extension, `schema.rb`) is the single source of truth. Compare this to `mysql2`, where every column has to be manually pulled out of a raw result row, field by field, every time.

🔧 Basic CRUD — Active Record vs Raw SQL

The Same Five Operations, Both Ways

Operation	mysql2 (Express)	Active Record (Rails)
Get all rows	connection.query("SELECT * FROM posts")	Post.all
Find by ID	connection.query("SELECT * FROM posts WHERE id = ?", [id])	Post.find(id)
Insert	connection.query("INSERT INTO posts (title) VALUES (?)", [title])	Post.create(title: title)
Update	connection.query("UPDATE posts SET title = ? WHERE id = ?", [title, id])	post.update(title: title)
Delete	connection.query("DELETE FROM posts WHERE id = ?", [id])	post.destroy

⛓️ Chained Query Methods

```
Post.where(published: true)
  .order(created_at: :desc)
  .limit(5)
```

Each method returns another chainable query object rather than immediately hitting the database — the actual SQL query is only built and executed once the result is enumerated (printed, looped over, etc.). This reads close to a plain-English sentence — "posts that are published, ordered by newest, limited to five" — and generates a single, efficient SQL query behind the scenes, comparable to method chaining Enumerable chains from both Ruby courses, just building SQL instead of transforming an in-memory array.

🛡️ Automatic SQL Injection Protection

```
# SAFE -- parameterized automatically
Post.where(title: params[:title])

# SAFE -- ? is a placeholder, Active Record escapes the value
Post.where("title = ?", params[:title])

# DANGEROUS -- raw string interpolation, exactly the vulnerability from the SQLi course
Post.where("title = '#{params[:title]}'")
```

Every one of Active Record's normal query methods — `where` with a hash, `find`, `create` — parameterizes values automatically, the same protection the completed SQL Injection course covered as essential. The dangerous pattern above is the exact same mistake covered there: building a query with raw string interpolation instead of a placeholder, whether in `mysql2` or, as shown, even inside Active Record's own string-based `where` form if used carelessly.

📜 Data Access, Side by Side

mysql2 vs Active Record

Concern	mysql2 (Express)	Active Record (Rails)
Schema changes	Hand-written SQL, tracked manually	Migrations, tracked and versioned automatically
Current schema reference	No built-in equivalent	db/schema.rb, auto-generated
Model definition	Manually map every column yourself	Empty class — columns introspected at runtime
Query building	Raw SQL strings with placeholders	Chainable Ruby methods (where/order/limit)
Injection safety	Manual — must use placeholders correctly every time	Automatic through the standard query interface

💻 Coding Challenges

Challenge 1: A Comment Migration

Write the `rails generate migration` command (with column types) and the resulting migration file for a `Comment` model with `post_id` (integer), `body` (text), and `author_name` (string). Then write out what `db/schema.rb` would show for the `comments` table after running `rails db:migrate`.

Goal: Practice the full migration → schema.rb pipeline for a new table.

→ Solution

Challenge 2: Raw SQL to Active Record

Convert this raw SQL into an equivalent chained Active Record query: `SELECT * FROM posts WHERE published = 1 ORDER BY title ASC LIMIT 10`.

Goal: Practice translating a query you already know how to write in SQL into Active Record's chainable method style.

→ Solution

Challenge 3: Spot the SQL Injection Risk

Given two versions of a search action — one using `Post.where("title LIKE ?", "%#{params[:q]}%")` and one using `Post.where("title LIKE '%#{params[:q]}%'")` — identify which one is vulnerable to SQL injection and explain, in a sentence referencing the completed SQLi course, exactly why.

Goal: Be able to spot this specific vulnerability pattern even inside Active Record's own query interface, not just in raw mysql2 code.

→ Solution

💎 Active Record Doesn't Remove SQL — It Automates the Safe Version

It's worth being precise about what actually changed here: Active Record isn't hiding SQL entirely, it's generating well-formed, parameterized SQL for you based on the same relational concepts (tables, columns, WHERE clauses) already familiar from raw SQL or `mysql2`. The genuine win isn't "you never think about SQL again" — it's that the default, easiest-to-write path is also the secure one, the same "secure by default" theme Chapter 4's automatic HTML escaping already introduced.

🎯 What's Next

Next chapter: **Active Record Associations** — `has_many`, `belongs_to`, and how Rails handles relationships between tables that would otherwise require manually writing `JOIN` queries.

🔗 Active Record Associations

Chapter 5's `Comment` migration added a plain `post_id` integer column — just a number, as far as the database is concerned. This chapter turns that raw foreign key into a real, navigable relationship, replacing the manual `JOIN` queries Express + `mysql2` would otherwise require.

👉 belongs_to

```
# app/models/comment.rb
class Comment < ApplicationRecord
  belongs_to :post
end
comment = Comment.find(1)
comment.post        # => the Post this comment belongs to, one method call
comment.post.title  # chain straight through to the post's own attributes
```

`belongs_to :post` relies on exactly the `post_id` column Chapter 5's migration created — Rails expects a foreign key named after the singular association (`post_id` for `belongs_to :post`) by convention, and generates a `comment.post` method that looks that row up automatically.

👉 has_many

```
# app/models/post.rb
class Post < ApplicationRecord
  has_many :comments
end
post = Post.find(1)
post.comments                    # all comments where post_id = post.id
post.comments.where(approved: true)   # chainable, exactly like Chapter 5's query methods
post.comments.create(body: "Nice post!")   # creates a new comment with post_id already set
```

`post.comments` replaces manually writing `SELECT * FROM comments WHERE post_id = ?` against a `mysql2` connection. It's also fully chainable — `post.comments.where(...)` scopes the query further, and `post.comments.create(...)` automatically sets `post_id` on the new row without you passing it explicitly.

🔀 has_many :through — Many-to-Many

Tagging a post with multiple tags (and a tag applying to multiple posts) is a classic many-to-many relationship, requiring a join table:

```
# app/models/post.rb
class Post < ApplicationRecord
  has_many :taggings
  has_many :tags, through: :taggings
end
# app/models/tagging.rb -- the join model, sitting between posts and tags
class Tagging < ApplicationRecord
  belongs_to :post
  belongs_to :tag
end
# app/models/tag.rb
class Tag < ApplicationRecord
  has_many :taggings
  has_many :posts, through: :taggings
end
post.tags          # => all tags on this post, joining through taggings automatically
```

`post.tags` silently performs the three-table join a hand-written SQL query would need — `posts JOIN taggings JOIN tags` — behind one method call. This is the association type worth taking slowest: it requires an actual join model (`Tagging`) with its own `belongs_to` declarations on both sides, not just a single line.

🗑️ dependent: :destroy

```
class Post < ApplicationRecord
  has_many :comments, dependent: :destroy
end
post.destroy   # automatically destroys every associated comment first, no orphaned rows left behind
```

Without `dependent: :destroy`, deleting a post would leave its comments in the database with a `post_id` pointing at nothing — the same orphaned-row problem raw SQL requires either a manual cleanup query or a database-level `ON DELETE CASCADE` constraint to prevent.

⚠️ The N+1 Query Problem

```
# Looks innocent -- silently issues 1 + N queries
Post.all.each do |post|
  puts "#{post.title}: #{post.comments.count} comments"
end
```

⚠ This runs one query to get the posts, then one MORE query per post

With 50 posts, the loop above issues **51 separate queries**: one for `Post.all`, then one more `SELECT COUNT(*) FROM comments WHERE post_id = ?` for every single post inside the loop — the notorious **N+1 query problem**. This is a genuine reversal of this course's usual pattern: writing the equivalent raw SQL `JOIN` query by hand in `mysql2` forces you to think about efficiency upfront, while Active Record's convenient `post.comments` syntax can hide this exact performance problem until it's slow in production. The fix is `.includes`:

```
# Fixed -- 2 queries total, not 51
Post.includes(:comments).each do |post|
  puts "#{post.title}: #{post.comments.count} comments"
end
```

`.includes(:comments)` eager-loads every post's comments in a second, separate query *before* the loop runs, instead of issuing a fresh query on every iteration — bringing the total back down to two queries regardless of how many posts exist.

📜 Relationships, Side by Side

Manual Joins vs Active Record Associations

Concern	mysql2 (Express)	Active Record (Rails)
One-to-many	SELECT * FROM comments WHERE post_id = ?	post.comments
Many-to-many	Manual 3-table JOIN	has_many :tags, through: :taggings
Cascading delete	Manual cleanup query or DB constraint	dependent: :destroy
Query efficiency	Explicit — you write exactly the JOIN you need	Implicit — easy to accidentally trigger N+1 queries

💻 Coding Challenges

Challenge 1: Wire Up Post and Comment

Add the correct association declarations to `Post` and `Comment` (from Chapter 5's migration) so that `@post.comments` and `@comment.post` both work. Write one line demonstrating each direction.

Goal: Practice the paired belongs_to/has_many declarations that make a foreign key column into a real, navigable relationship.

→ Solution

Challenge 2: has_many :through for Tags

Following this chapter's Post/Tag/Tagging example, write out the three model files in full (`Post`, `Tag`, and the `Tagging` join model), then write one line showing how you'd get all tags for a given post.

Goal: Get comfortable with the three-model shape a has_many :through relationship actually requires.

→ Solution

Challenge 3: Fix an N+1 Query

Given `Author.all.each { |author| puts author.books.count }`, explain how many total queries this issues for 20 authors, then rewrite it using `.includes` to fix the problem, and state the new total query count.

Goal: Recognize the N+1 pattern on sight and know the one-word fix.

→ Solution

💎 Convenient Syntax Doesn't Mean Free

The N+1 problem is this chapter's clearest reminder that `post.comments` being one short method call doesn't mean it's one cheap operation — it's still a real database query every time it runs, exactly like the hand-written SQL it replaces. Active Record hides the query, not the cost of running it. Reading generated SQL logs (Rails prints every query it runs in development) is a genuinely useful habit for catching this before it reaches production.

🎯 What's Next

Next chapter: **Active Record Validations & Callbacks** — `validates`, `before_save`, and where Rails expects business rules to actually live inside a model.

✅ Active Record Validations & Callbacks

Chapter 3's `create` and `update` actions branched on `if @post.save` without fully explaining why `.save` could ever return `false`. This chapter answers that: **validations** live directly on the model, run automatically before every save, and replace the manual validation logic Express handles in middleware or a service layer (Chapter 1's reviewed example).

✔️ validates — Declaring Rules on the Model

```
class Post < ApplicationRecord
  validates :title, presence: true, length: { maximum: 100 }
  validates :body, presence: true
  validates :slug, uniqueness: true
  validates :view_count, numericality: { greater_than_or_equal_to: 0 }
end
```

Common built-in validators: `presence` (not blank), `length` (min/max), `uniqueness` (no duplicate value across all rows), `numericality` (must be a number, optionally within a range), and `format` (must match a regex). Every one of these runs automatically before `.save` actually writes to the database.

❓ How Validations Actually Work

```
post = Post.new(title: "", body: "Some content")

post.valid?        # => false  -- runs every validation, returns true/false
post.save          # => false  -- refuses to write to the DB, returns false instead of raising
post.errors.full_messages
# => ["Title can't be blank"]
```

This is exactly the mechanism behind Chapter 3's `if @post.save ... else render :new ... end` pattern: an invalid record makes `.save` return `false`, and `post.errors` holds a real, structured collection of what went wrong — which is what a re-rendered form (Chapter 8) uses to show error messages next to the offending fields.

🏢 Model-Level vs Middleware-Level Validation

⚠ Recall Express's service-layer validation from Chapter 1

The Express example reviewed back in Chapter 1 hand-wrote its validation inside a service function — `if (!data?.name?.trim()) errors.push(...)` — which only runs if that specific service function is actually called. Any other code path that touches the database directly (a script, a different service, a background job) bypasses it entirely, since the validation isn't attached to the data itself. Rails' `validates` lives directly on the `Post` model, so **any** code that calls `.save` on a `Post` — a controller, the Rails console, a background job — automatically runs the same rules, with no risk of one entry point forgetting to re-implement them.

🔧 Custom Validations

```
class Post < ApplicationRecord
  validate :publish_date_not_in_past # note: validate (singular), not validates
private
def publish_date_not_in_past
return if publish_date.blank?
    if publish_date < Date.today
      errors.add(:publish_date, "can't be in the past")
    end
end
end
```

When a rule doesn't fit a built-in validator, `validate :method_name` (singular) runs your own instance method, which manually calls `errors.add` if something's wrong — the equivalent of writing an arbitrary custom check inside an Express middleware or service function, just triggered by the same automatic save lifecycle as the built-in validators.

🔄 Callbacks — Lifecycle Hooks

```
class Post < ApplicationRecord
  before_save :normalize_title
  before_create :generate_slug
  after_create :notify_subscribers
private
def normalize_title
self.title = title.strip
  end
def generate_slug
self.slug = title.downcase.gsub(/\s+/, "-")
  end
def notify_subscribers
SubscriberMailer.new_post(self).deliver_later
  end
end
```

Callbacks run automatically at specific points in a record's save/create/destroy lifecycle — `before_save` before any write, `before_create` only on the first insert, `after_create` after a new row exists, and similarly-named hooks for updates and destruction. This is comparable to Express middleware running before/after a route handler, except scoped to the model's own lifecycle rather than the HTTP request lifecycle.

⚠ Callbacks can make control flow genuinely hard to trace

`notify_subscribers` above means calling `post.save` anywhere in the codebase silently sends an email — a real side effect, triggered from a place that, reading just the calling code, looks like nothing more than "save this record." In Express, that same side effect would be an explicit, visible line inside the route handler or service function. Overusing callbacks for anything beyond simple, self-contained data preparation (like `normalize_title` above) is a well-known Rails anti-pattern precisely because it hides consequential side effects behind an innocent-looking method call.

📜 Validation & Side Effects, Side by Side

Express vs Rails

Concern	Express	Rails
Where validation lives	Middleware or service-layer code, per route	Directly on the model, via validates
Applies to every code path?	Only if every entry point calls the same validation	Yes — automatic on every .save, everywhere
Reporting what went wrong	Manually built error array/object	record.errors, structured and built in
Side effects around a save	Explicit lines in the route/service	Callbacks — implicit, can be hard to trace

💻 Coding Challenges

Challenge 1: Validate a Model

Add validations to a `Product` model: `name` must be present with a maximum length of 50, `price` must be a number greater than 0, and `sku` must be unique. Then write code that builds an invalid `Product` (blank name, negative price), calls `.valid?`, and prints `errors.full_messages`.

Goal: Practice the most common built-in validators together on one model.

→ Solution

Challenge 2: A Custom Validation

Write a custom validation on an `Event` model ensuring `end_time` is always after `start_time` — a rule no single built-in validator can express — using `validate :method_name` and `errors.add`.

Goal: Practice writing a validation for a business rule that spans two fields.

→ Solution

Challenge 3: A Careful Callback

Write a `before_save :downcase_email` callback on a `User` model that normalizes the `email` attribute to lowercase before every save. Then write two sentences: one explaining why this particular callback is safe/appropriate, and one explaining what would make a *different* callback (like sending a welcome email) risky by comparison.

Goal: Practice writing an appropriately-scoped callback and articulate the line between safe and risky callback use.

→ Solution

💎 One Definition, Every Entry Point

The real win this chapter is about centralization, not brevity: a validation rule written once on a model applies everywhere that model is ever saved, for the entire lifetime of the application — no risk of a new controller action, a background job, or a console script quietly bypassing a rule that only lived in one specific Express route's middleware. That guarantee is worth more than the line count it saves.

🎯 What's Next

Next chapter — the final chapter of this course: **Forms & Strong Parameters** — `form_with`, `params.require().permit()`, and Rails' built-in CSRF protection, contrasted with the manual work Express needs for the same protections.

📋 Forms & Strong Parameters

This final chapter of the course closes a loop running since Chapter 3: the forms that actually submit to `create` and `update`, the security mechanism preventing a submitted form from setting fields it shouldn't, and the CSRF protection every one of Rails' forms has been quietly including the entire time.

📝 form_with

```
<!-- app/views/posts/_form.html.erb -->
<%= form_with model: post do |f| %>
  <%= f.label :title %>
  <%= f.text_field :title %>

  <%= f.label :body %>
  <%= f.text_area :body %>

  <%= f.submit %>
<% end %>
```

`form_with model: post` inspects the `post` object and automatically decides everything Express would need spelled out by hand: a new, unsaved `Post` generates a form that `POST`s to `/posts` (the `create` action); an existing, persisted `Post` generates a form that submits as `PATCH` to `/posts/:id` (the `update` action) — the same partial (from Chapter 4) works for both the `new` and `edit` views without any conditional logic inside it.

🏷️ Field Helpers Generate Rails-Shaped Names

```
<%= f.text_field :title %>
<!-- generates: -->
<input type="text" name="post[title]" id="post_title" value="...">
```

`f.text_field :title` generates an input with `name="post[title]"` — matching Chapter 3's nested `params[:post][:title]` structure automatically. In Express + EJS, this nesting has to be hand-built: writing `name="title"` directly, or manually adopting a naming convention like `name="post[title]"` yourself with no framework enforcement or generation.

⚠️ Displaying Validation Errors

```
<% if post.errors.any? %>
  <div class="errors">
    <ul>
      <% post.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
    </ul>
  </div>
<% end %>

<%= form_with model: post do |f| %>
  ...
<% end %>
```

This is Chapter 7's `errors.full_messages` put to work: when `create` fails and re-renders `:new` (Chapter 3), the same `@post` instance — invalid, but still holding the user's submitted values and the validation errors — flows straight into this partial with no extra wiring.

🔒 Strong Parameters

```
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
def create
@post = Post.new(post_params)
  # ...
end
private
def post_params
  params.require(:post).permit(:title, :body, :published)
 end
end
```

`params.require(:post)` raises an error if the `post` key is missing entirely; `.permit(:title, :body, :published)` explicitly whitelists exactly which fields are allowed through — anything else submitted in the form gets silently dropped, never reaching `Post.new`.

⚠ Without this, you have a mass assignment vulnerability

Imagine `Post.new(params[:post])` instead — passing the entire submitted hash straight into the model with no whitelist. If `Post` ever gains an `admin_only` or `featured` boolean column, a malicious user could add a hidden field, `post[featured]=true`, to their form submission (using browser dev tools, or a crafted request bypassing the UI entirely) and set a field the form never intended to expose. This is a **mass assignment vulnerability** — trusting raw user input to determine which attributes get set, the same "never trust user input" principle underlying every completed security course, just applied specifically to model attribute assignment.

🛡️ CSRF Protection, Built In

```
# app/controllers/application_controller.rb -- present by default in every new Rails app
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
end
```

Recall the completed CSRF security course: preventing cross-site request forgery requires a unique, unpredictable token embedded in every form and verified on submission. Rails generates this automatically — `protect_from_forgery` is present in `ApplicationController` from the moment `rails new` runs (Chapter 1), and every `form_with` call silently embeds the token as a hidden field with zero additional code required.

CSRF Protection: Express vs Rails

Step	Express	Rails
Enable protection	Install and configure csurf (or a modern equivalent) manually	On by default — no setup required
Embed the token in a form	Manually add a hidden input with the token, in every form	Automatic — form_with does it for you
Verify on submission	Middleware you wired up yourself	Automatic — built into ApplicationController

💻 Coding Challenges

Challenge 1: A Product Form

Write `app/views/products/_form.html.erb` using `form_with model: product` with fields for `name` (text field), `description` (text area), and `in_stock` (checkbox), plus a submit button.

Goal: Practice the standard form_with + field helper pattern for a realistic set of field types.

→ Solution

Challenge 2: Strong Parameters and a Mass Assignment Attack

Write the `product_params` private method for a `ProductsController`, permitting only `name`, `description`, and `price` — deliberately excluding a `featured` boolean column that exists on the table. Then write two sentences describing exactly what a malicious form submission would need to include to try to set `featured`, and why `product_params` as written stops it.

Goal: Understand strong parameters as an active defense, not just boilerplate.

→ Solution

Challenge 3: Displaying Errors in a Form

Add an error-display block (following this chapter's pattern) above a `form_with model: product` form, then trace through what happens end to end: a user submits an invalid product, `create` fails, and the form re-renders. Write out, step by step, which chapter's concept handles each part of that flow (validation, the branch in the controller, the errors object, the re-rendered view).

Goal: Connect this chapter's form back to the validations (Chapter 7) and controller logic (Chapter 3) it depends on.

→ Solution

💎 Security as a Default, Not a Feature You Add

Look back across this course: automatic HTML escaping (Chapter 4), automatic SQL injection protection through the query interface (Chapter 5), and now strong parameters and built-in CSRF protection — every one of them is the *same* underlying pattern this course keeps returning to. Rails doesn't just make the common case shorter to write; it consistently makes the common case the **secure** case, requiring a deliberate, visible opt-out (`raw()`, string-interpolated SQL, skipping `permit`) to reach the dangerous version instead of a deliberate opt-in to reach the safe one.

🎯 What's Next

Course 1 (Ruby on Rails Fundamentals) is complete. Ruby on Rails Intermediate/Advanced (Course 2) begins with **Authentication** — `has_secure_password`/bcrypt and session-based login, contrasted with Express's typical JWT-based approach.