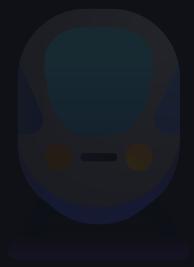

RUBY ON RAILS

Intermediate / Advanced

Authentication · Authorization · Testing Rails Apps

Background Jobs · APIs with Rails

Action Cable · Hotwire · Deployment

Philip Osztromok

Generated with Claude

Table of Contents

Ruby on Rails Intermediate/Advanced — 8 Chapters

🔐 Authentication

Course 1 built CRUD for posts and comments, but never checked who was actually making a request. This chapter adds real login — password hashing via `has_secure_password`, and Rails' default **session-based** approach, which works fundamentally differently from the JWT-based authentication Express Course 2 covered.

🔑 has_secure_password

```
$ rails generate migration CreateUsers email:string password_digest:string
$ rails db:migrate


# Gemfile -- uncomment this line (it's included but commented out by default)
gem "bcrypt", "~> 3.1.7"
# app/models/user.rb
class User < ApplicationRecord
 has_secure_password
 validates :email, presence: true, uniqueness: true
end
```

`has_secure_password` is a single macro that adds a huge amount of functionality on top of the `password_digest` column: virtual `password` and `password_confirmation` attributes (never actually stored), automatic bcrypt hashing before save, and an `authenticate(password)` method that returns the user if the password matches, `false` otherwise.

```
user = User.create(email: "philip@example.com", password: "secret123", password_confirmation: "secret123")
user.password_digest   # => "$2a$12$K3JR9..." -- the bcrypt hash, this is what's actually stored

user.authenticate("wrong")      # => false
user.authenticate("secret123")  # => the user object itself
```

🍪 SessionsController — Login as a Singular Resource

```
# config/routes.rb
resource :session, only: [:new, :create, :destroy]   # singular "resource" -- no :id, there's only ever one current ses
sion
# app/controllers/sessions_controller.rb
class SessionsController < ApplicationController
def new
end
def create
@user = User.find_by(email: params[:email])
   if @user&.authenticate(params[:password])
     session[:user_id] = @user.id
     redirect_to root_path
   else
     render :new, status: :unprocessable_entity
end
end
def destroy
   session[:user_id] = nil
   redirect_to root_path
 end
end
```

Notice `@user&.authenticate(...)` — the safe navigation operator from Ruby Course 2, Chapter 8, guarding against a `nil` user (no matching email) before calling `authenticate` on it. `resource :session` (singular, no `s`) is Rails' convention for a resource where there's only ever one — a session either exists for the current browser or it doesn't, so there's no `:id` in its routes.

⚖️ Sessions vs JWT — A Genuinely Different Architecture

Recall Express Course 2's authentication chapter: a JWT is generated on login, signed, and handed to the client, which then attaches it to every subsequent request — the server verifies the signature and never needs to store anything about "who's logged in." Rails' default `session[:user_id] = user.id` works the opposite way:

JWT (Express) — stateless

The token itself contains the user's identity, cryptographically signed. The server verifies the signature on each request but stores nothing between requests — "statelessness" is the whole point, which is why JWTs scale easily across multiple servers with no shared session store.

Session cookie (Rails default) — stateful

`session[:user_id]` is stored in an encrypted, signed cookie sent to the browser — by default Rails keeps the session data itself inside that cookie (`ActionDispatch::Session::CookieStore`), tamper-proof via a secret key, but still fundamentally tied to "the browser holds a reference the server trusts."

⚠ Revocation is the real practical difference

This is the tradeoff the completed Authentication & Session Security course already covered in the abstract, now concrete: a Rails session can be invalidated instantly and server-side — change the session secret, or (with a database-backed session store) simply delete the session record, and that user is logged out immediately. A JWT, once issued, remains valid until it expires no matter what the server does, unless you build a separate revocation/blocklist mechanism — the "statelessness" that makes JWTs easy to scale is the exact same property that makes them hard to revoke early.

👤 current_user and Protecting Routes

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
private
def current_user
@current_user ||= User.find_by(id: session[:user_id])
 end
 helper_method :current_user # makes it available in views too, not just controllers
def require_login
unless current_user
    redirect_to new_session_path, alert: "Please log in first"
end
end
end
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
 before_action :require_login, except: [:index, :show]   # guests can browse, must log in to write
# ...
end
```

`current_user` uses the `||=` memoization pattern from both Ruby courses — computed once per request, cached in `@current_user` for any further calls in the same request. `before_action :require_login` is Rails' controller-scoped equivalent of Express's authentication middleware, running before the specified actions and redirecting away if the check fails — a genuinely close conceptual match between the two frameworks, unlike most of this course's other comparisons.

📜 Authentication, Side by Side

Express (JWT) vs Rails (Session)

Concern	Express (typical JWT)	Rails (default sessions)
Where identity lives	Inside the signed token itself	Server-trusted reference in an encrypted cookie
Server storage needed?	No (stateless)	No, by default (cookie store) — but can be database-backed
Instant revocation	Hard — needs a separate blocklist	Easy — invalidate server-side
Route protection	Auth middleware (express2-1)	before_action :require_login
Password hashing	bcrypt, called manually	bcrypt, via has_secure_password

💻 Coding Challenges

Challenge 1: A User Model

Write the migration command and resulting migration file for a `User` with `email` and `password_digest`, then write the `User` model with `has_secure_password` and a uniqueness validation on `email`.

Goal: Set up the full has_secure_password foundation from scratch.

→ Solution

Challenge 2: SessionsController

Write the full `SessionsController` with `new`, `create`, and `destroy` actions, following this chapter's pattern — including the safe-navigation `authenticate` check and the correct session key assignment on login/logout.

Goal: Practice the complete login/logout flow end to end.

→ Solution

Challenge 3: Protecting a Controller

Write `current_user` and `require_login` in `ApplicationController`, then add `before_action :require_login` to a `CommentsController` so that only `index` and `show` are accessible without logging in.

Goal: Practice guarding specific controller actions the way express2-1's middleware guarded specific Express routes.

→ Solution

💎 Neither Approach Is Simply "Better"

JWTs win when an API needs to scale across many stateless servers with no shared session store, or serve non-browser clients (mobile apps, other services) where cookies don't naturally fit. Rails' session-based default wins for traditional server-rendered apps like the one this course has been building — simpler to revoke, and it doesn't require the client to manage token storage or attach headers manually. Rails *can* do JWT-based API authentication too (relevant once Chapter 5 covers building APIs with Rails) — the session default is a starting point suited to this course's app, not a hard limitation of the framework.

🎯 What's Next

Next chapter: **Authorization** — now that Rails knows *who* is logged in, controlling *what* they're allowed to do, using Pundit or CanCanCan for role-based access.

🚧 Authorization

Chapter 1 answered *who* is making a request. This chapter answers a genuinely different question: *what* are they allowed to do? Authentication and authorization are easy to conflate but solve separate problems — a logged-in user (authenticated) still shouldn't be able to edit someone else's post (unauthorized).

🏷️ The Simplest Approach: A Role Column

```
$ rails generate migration AddRoleToUsers role:integer


# app/models/user.rb
class User < ApplicationRecord
 has_secure_password
 enum role: { member: 0, admin: 1 }
end

current_user.admin?   # => true/false -- enum generates this predicate method automatically
```

`enum` stores `role` as a plain integer in the database but gives you named, readable methods (`admin?`, `member?`) instead of comparing raw numbers everywhere — a small taste of Rails' broader convention-over-configuration habit applied to a single column. This is enough for the simplest cases:

```
# app/controllers/posts_controller.rb
def destroy
unless current_user.admin?
  redirect_to posts_path, alert: "Not authorized"
return
end
# ...
end
```

⚠ This gets unmanageable fast, past the simplest cases

The moment authorization rules get more nuanced than "admin or not" — "the post's own author OR an admin can edit it," "a moderator can hide comments but not delete posts" — hand-written `unless` checks scattered across every controller action become genuinely hard to keep consistent, and easy to forget in a new action entirely. This is the exact same "duplicated across every entry point" problem Chapter 1 Course 1's validations chapter solved by centralizing rules on the model. Authorization needs its own centralizing pattern too.

📜 Pundit — Policy Objects

```
$ bundle add pundit


# app/policies/post_policy.rb
class PostPolicy < ApplicationPolicy
def update?
  user.admin? || record.user == user  # record is the Post being checked
end
def destroy?
  user.admin?  # only admins can delete, regardless of authorship
end
end
```

Pundit's convention: one policy class per model (`PostPolicy` for `Post`, matching the same naming pattern controllers and models already follow), with a method per action ending in `?` — the same predicate-method naming convention from both Ruby courses. Every rule for a given model lives in exactly one file.

```
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
def update
@post = Post.find(params[:id])
  authorize @post # raises Pundit::NotAuthorizedError if PostPolicy#update? returns false
# ...
end
end
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
include Pundit::Authorization
 rescue_from Pundit::NotAuthorizedError, with: :user_not_authorized
private
def user_not_authorized
  redirect_to root_path, alert: "You're not authorized to do that"
end
end
```

`authorize @post` automatically looks up `PostPolicy` (from the object's class name) and calls the method matching the current action name — `update` calls `update?`. `rescue_from` is Rails' framework-wide exception handler declaration, catching the raised error anywhere in the app and redirecting gracefully instead of showing a raw error page.

🔍 Checking Permissions in Views

```
<% if policy(@post).update? %>
  <%= link_to "Edit", edit_post_path(@post) %>
<% end %>
```

`policy(@post).update?` reuses the exact same `PostPolicy` logic to conditionally show the edit link — no separate, potentially-inconsistent permission check duplicated in the view. Contrast this with Express, where a role check to conditionally render a button in an EJS template is typically a separate, hand-written `if` statement with no guaranteed relationship to whatever check the corresponding route handler actually performs.

🆚 Pundit vs CanCanCan

Pundit — one policy per model

Rules for `Post` live in `PostPolicy`, rules for `Comment` in `CommentPolicy`. Fits naturally alongside Rails' existing one-class-per-concern convention (one controller per resource, one model per table).

CanCanCan — one Ability class, everything

A single `Ability` class defines every permission for every model in one place, using `can :update, Post, user_id: user.id`-style declarations. Convenient for seeing the whole permission system at a glance; can become one large, sprawling file as an app grows.

Both are well-established, widely-used gems — this course continues with Pundit specifically because its file-per-model split matches the organizational convention already established for controllers, models, and (in the next chapter) test files.

📜 Authorization, Side by Side

Express (manual) vs Rails (Pundit)

Concern	Express (typical)	Rails + Pundit
Where rules live	Scattered if-checks per route, or ad hoc middleware	One policy class per model
Controller check	Hand-written conditional per route	authorize @record — one line, consistent everywhere
View-level check	Separate, potentially inconsistent logic	policy(@record).action? — same source of truth
Unauthorized handling	Manual per route	rescue_from, handled once, app-wide

💻 Coding Challenges

Challenge 1: A Role Column

Write the migration and model changes to add a `role` enum (`member`/`admin`) to `User`. Then write a `destroy` action on a `CommentsController` that only allows admins to delete a comment, redirecting with an alert otherwise.

Goal: Build the simplest possible role-based check before reaching for a gem.

→ Solution

Challenge 2: A Pundit Policy

Write a `CommentPolicy` where `update?` allows the comment's own author or an admin, and `destroy?` allows the comment's author, the post's author, or an admin (assume `record.post.user` gives you the post's author). Then write the `authorize @comment` call in the controller's `update` action.

Goal: Practice writing policy logic more nuanced than a single role check.

→ Solution

Challenge 3: Conditional View Links

Using the `CommentPolicy` from Challenge 2, write the ERB for a comment partial that only shows "Edit" and "Delete" links when `policy(comment)` permits each action respectively.

Goal: Reuse policy logic in a view instead of duplicating the permission check.

→ Solution

💎 The Same Centralization Lesson, Applied Again

This chapter is really Course 1's validations chapter, replayed for a different concern: rules written once, in one predictable location, apply consistently everywhere they're needed — the controller, the view, anywhere else the app might eventually check permissions. The specific mechanism differs (`validates` on a model vs a separate policy class), but the underlying principle — one source of truth beats duplicated logic — is the same idea this course keeps returning to.

🎯 What's Next

Next chapter: **Testing Rails Apps** — RSpec request and model specs, and FactoryBot, building directly on Ruby Course 2's RSpec chapter to test everything covered so far, including this chapter's policies.

🧪 Testing Rails Apps

The last two chapters wrote real rules: "only the comment's author or an admin can edit it," "unauthorized requests redirect with an alert." Rules like that are exactly the kind of logic that quietly breaks during a refactor unless something is actually checking it. Ruby Course 2's RSpec chapter (`ruby2-6`) covered `describe`/`it`/`expect` for testing plain Ruby classes — this chapter applies the same tool to an entire Rails app: models, HTTP requests, and the Pundit policies from Chapter 2.

RSpec in a Rails App

```
$ bundle add rspec-rails --group test
$ rails generate rspec:install
```

This creates a `spec/` directory (Rails' convention over the plain `test/` directory Minitest would use) and a `rails_helper.rb` that loads the Rails environment before every test run. Everything from `ruby2-6` — `describe`, `it`, `expect(...).to` matchers, `before(:each)` — carries over unchanged. What's new is *what* gets described: not a plain class, but models, requests, and policies wired into a real Rails app.

🗄️ Model Specs

```
# spec/models/comment_spec.rb
RSpec.describe Comment, type: :model do
 it "is invalid without a body" do
  comment = Comment.new(body: nil)
  expect(comment).to_not be_valid
 end

 it "belongs to a post" do
  expect(Comment.reflect_on_association(:post).macro).to eq(:belongs_to)
 end
end
```

Model specs verify exactly the things Course 1 taught you to declare — `validates :body, presence: true` from `rails1-7`, `belongs_to :post` from `rails1-6` — actually behave the way they claim to. `be_valid` is RSpec's automatic matcher for any `valid?`-ending predicate method, the same auto-generated-matcher trick `ruby2-6` covered for plain Ruby objects.

🏭 FactoryBot — Test Data Without Fixtures

```
$ bundle add factory_bot_rails --group test


# spec/factories/comments.rb
FactoryBot.define do
 factory :comment do
   body { "A perfectly reasonable comment" }
   association :post
   association :user
end
end
# in a spec:
comment = create(:comment, body: "Overriding just this attribute")
```

Rails ships with **fixtures** — static YAML files loaded wholesale before each test — but they get unwieldy fast: every association has to be hand-wired by ID across separate files, and every test sees the same shared data whether it needs it or not. FactoryBot builds records in code instead, one factory per model, with only the attributes a given test actually cares about overridden inline. `create` saves to the (test) database; `build` instantiates without saving, for tests that don't need persistence.

💎 Traits for Variations

`trait :from_admin { association :user, factory: :admin_user }` lets a single factory produce meaningful variations (`create(:comment, :from_admin)`) without a combinatorial explosion of near-duplicate factories — the same instinct behind default arguments in `ruby1-4`: define the common case once, override only what differs.

🌐 Request Specs

```
# spec/requests/comments_spec.rb
RSpec.describe "Comments", type: :request do
 it "lets the author update their own comment" do
  comment = create(:comment, user: current_user)
  sign_in current_user
  patch comment_path(comment), params: { comment: { body: "edited" } }
  expect(response).to redirect_to(comment)
 end

 it "rejects an update from a non-author" do
  comment = create(:comment)
  sign_in create(:user)
  patch comment_path(comment), params: { comment: { body: "edited" } }
  expect(response).to redirect_to(root_path)   # Pundit's rescue_from from rails2-2
end
end
```

Request specs send a real HTTP request through the full stack — routing, middleware, controller, view — and assert on the response, the same end-to-end shape as Express's `supertest` tests from `express2-5`. They replaced Rails' older "controller specs," which called controller actions directly and skipped routing entirely; that shortcut turned out to hide real bugs (like a route that never matched in the first place), so request specs are now the recommended default.

🔐 Testing the Authorization Chapter

The second request spec above is really testing Chapter 2's `CommentPolicy#update?` indirectly, through the full HTTP round-trip. Pundit also ships a matcher for testing a policy directly, without going through a controller at all:

```
# spec/policies/comment_policy_spec.rb
RSpec.describe CommentPolicy do
 subject { described_class.new(user, comment) }

 context "the comment's author" do
  let(:comment) { create(:comment, user: user) }
  let(:user) { create(:user) }

  it { is_expected.to permit_action(:update) }
 end
end
```

Policy specs are fast (no HTTP layer, no full request cycle) and precise (they test *only* the permission logic), while request specs confirm that logic is actually wired up correctly end to end. Good Rails test suites use both — narrow, fast specs for logic, broader request specs for "does the whole thing actually work."

📜 Rails Testing vs Express Testing

RSpec + FactoryBot (Rails) vs Jest + supertest (Express)

Concern	Express (express2-5)	Rails
Test framework	Jest	RSpec
HTTP-level tests	supertest against the Express app	Request specs (type: :request)
Test data	Hand-built objects or mocked DB layer	FactoryBot factories, real test database
Isolated unit tests	Jest unit tests + manual mocks	Model specs, policy specs
Database in tests	Often mocked to avoid a real DB	Real test DB, wrapped in a rollback transaction per test

The database row is worth calling out: `express2-5` leaned toward mocking the database layer to keep tests fast and isolated. Rails takes the opposite default — tests run against a real (separate, disposable) test database, with each test wrapped in a transaction that's rolled back afterward, so tests stay isolated from each other without giving up the ability to catch real SQL-level bugs a mock would miss.

💻 Coding Challenges

Challenge 1: A Model Spec

Given a `Post` model with `validates :title, presence: true` and `belongs_to :user`, write a model spec with two examples: one confirming a `Post` without a title is invalid, and one confirming the `user` association exists.

Goal: Practice turning a model's declared rules into concrete assertions.

→ Solution

Challenge 2: A FactoryBot Factory

Write a `FactoryBot` factory for a `Post` model (attributes: `title`, `body`, belongs to a `user`). Then show how a spec would use it to create a post with a custom `title` while leaving the other attributes at their factory defaults.

Goal: Practice defining and overriding factory attributes.

→ Solution

Challenge 3: A Request Spec for Authorization

Using the `PostPolicy` from this course (only an admin or the post's author can destroy it), write a request spec for `DELETE /posts/:id` with two examples: one where the author successfully deletes their post, and one where a different, non-admin user is redirected instead.

Goal: Test an authorization rule through the full HTTP stack, not just the policy in isolation.

→ Solution

💎 Test the Shape of the Rule, Not Just One Example

Every spec in this chapter follows the same shape: one example proving the rule allows what it should, one proving it denies what it should. A model spec that only checks "a valid post passes" without also checking "an invalid post fails" hasn't actually verified the validation is doing anything — it would still pass even if the `validates` line were deleted entirely.

🎯 What's Next

Next chapter: **Background Jobs** — Active Job and Sidekiq, for work (sending an email, resizing an image) that shouldn't block the request/response cycle this chapter just finished testing.

🛠️ Background Jobs

Last chapter closed with a promise: work that shouldn't block the request/response cycle. Sending a confirmation email, resizing an uploaded image, calling a slow third-party API — none of these need to finish before the user gets a response. Making them wait anyway means slower page loads at best, and a request that times out entirely if the slow thing takes too long.

🐢 The Problem With Doing Everything Inline

```
# app/controllers/posts_controller.rb -- don't do this
def create
@post = Post.create!(post_params)
 NotifierMailer.new_post_email(@post).deliver_now  # blocks until the email actually sends
 redirect_to @post
end
```

If the mail server is slow, or down, the user sitting on the other end of that `create` request waits for it too — or the request times out and the post never even gets its redirect, even though it saved successfully. The fix is the same shape as the previous three chapters: pull the risky, slow, or optional part out into its own well-defined unit, instead of leaving it tangled into the request cycle.

Active Job — Rails' Abstraction Layer

Active Job is Rails' framework-provided abstraction for background work, the same role Active Record plays for databases: you write job classes against one consistent API, and a swappable **adapter** underneath actually handles running them — Sidekiq, Resque, Delayed Job, or Rails' built-in Async adapter for development. Switching adapters in production is a one-line config change, not a rewrite of every job.

```
# app/jobs/new_post_notifier_job.rb
class NewPostNotifierJob < ApplicationJob
 queue_as :default
def perform(post_id)
  post = Post.find(post_id)
  NotifierMailer.new_post_email(post).deliver_now
 end
end
```

⚠ Pass IDs, Not Objects, Into Jobs

`perform(post_id)` takes an integer, not the `Post` object itself. Jobs are serialized (to JSON, typically) and stored until a worker picks them up — by the time that happens, the in-memory `Post` object from the controller is long gone, and the record may have changed or even been deleted. Looking it up fresh inside `perform` guarantees the job always works with current data, and fails cleanly (`ActiveRecord::RecordNotFound`) if the record is gone rather than silently acting on stale data.

🚀 Enqueuing a Job

```
# app/controllers/posts_controller.rb
def create
@post = Post.create!(post_params)
 NewPostNotifierJob.perform_later(@post.id)  # enqueues and returns immediately
 redirect_to @post
end
```

`perform_later` hands the job off to the queue and returns instantly — the controller action finishes and the user gets their redirect right away, while the actual email sends whenever a worker gets to it. `perform_now` (the same method every job class gets) runs the job's `perform` immediately and synchronously, useful for testing or for the rare case where you genuinely do need to wait for the result.

🔴 Setting Up Sidekiq

```
$ bundle add sidekiq


# config/application.rb
config.active_job.queue_adapter = :sidekiq
# config/routes.rb -- optional web dashboard
require "sidekiq/web"
mount Sidekiq::Web => "/sidekiq"
```

Sidekiq is the most widely used Active Job adapter in production Rails apps. It's backed by Redis rather than the primary database — jobs are pushed onto Redis-backed queues, and a separate `sidekiq` worker process (started independently of the Rails server itself, with `bundle exec sidekiq`) pulls jobs off those queues and runs them, using threads to process several jobs concurrently per process. The optional `Sidekiq::Web` mount gives you a live dashboard of queued, running, retrying, and failed jobs.

🔁 Retries and Failure

```
class NewPostNotifierJob < ApplicationJob
 queue_as :default
 retry_on Net::SMTPServerBusy, wait: :exponentially_longer, attempts: 5
 discard_on ActiveRecord::RecordNotFound
def perform(post_id)
   post = Post.find(post_id)
   NotifierMailer.new_post_email(post).deliver_now
 end
end
```

`retry_on` declares that a specific error is transient and worth trying again — a busy mail server probably isn't busy forever, so retrying with exponentially increasing delays gives it time to recover. `discard_on` is the opposite judgment call: if the post was deleted before the job ran, retrying `RecordNotFound` five more times can't fix anything, so the job gives up immediately rather than clogging the queue with retries that will never succeed.

🧪 Testing a Job

Chapter 3 built request and model specs; jobs get their own small addition on top of the same RSpec setup:

```
# spec/jobs/new_post_notifier_job_spec.rb
RSpec.describe NewPostNotifierJob, type: :job do
 it "enqueues the job" do
   post = create(:post)
   expect { NewPostNotifierJob.perform_later(post.id) }
     .to have_enqueued_job.with(post.id)
 end

 it "sends the email when performed" do
   post = create(:post)
   expect { NewPostNotifierJob.perform_now(post.id) }
     .to change { ActionMailer::Base.deliveries.count }.by(1)
 end
end
```

The first example checks that enqueuing happens at all, without actually running the job's code — fast, and enough to confirm the controller wires things up correctly. The second uses `perform_now` to run the job's logic directly and confirm its actual effect. Splitting "was it enqueued" from "does it work when it runs" mirrors the same model-spec-vs-request-spec split from Chapter 3: narrow tests for one piece of behavior, not one giant test trying to prove everything at once.

📜 Background Jobs, Side by Side

Active Job + Sidekiq (Rails) vs BullMQ (Node, node3-6)

Concern	Node + BullMQ	Rails + Active Job/Sidekiq
Backing store	Redis	Redis (via Sidekiq specifically)
Enqueuing	queue.add("job-name", data)	SomeJob.perform_later(args)
Job definition	A worker function processing named jobs	One class per job, perform method
Retries	Configured per queue/job options	retry_on / discard_on, per error class
Swapping backends	Tied to BullMQ/Redis directly	Active Job is adapter-agnostic; jobs unchanged

The underlying idea — a durable, Redis-backed queue and one or more worker processes pulling jobs off it — is the same one `node3-6` already covered for message queues in general. Active Job's contribution is the same one Active Record makes for databases: one API on top, so the app's job classes don't have to know or care whether Sidekiq, Resque, or something else entirely is doing the actual work underneath.

💻 Coding Challenges

Challenge 1: A Basic Job

Write a `ResizeImageJob` that takes an `image_id`, looks up the corresponding `Image` record, and calls a (hypothetical) `image.resize!` method. Then write the one line a controller's `create` action would use to enqueue it after saving a new image.

Goal: Practice the perform_later pattern with an ID, not an object, passed in.

→ Solution

Challenge 2: Retry and Discard

Add `retry_on` for a hypothetical `ImageProcessingTimeout` error (retry up to 3 times, with exponentially increasing delays) and `discard_on` for `ActiveRecord::RecordNotFound` to the `ResizeImageJob` from Challenge 1.

Goal: Practice distinguishing a worth-retrying failure from a give-up-immediately failure.

→ Solution

Challenge 3: Testing the Job

Write two RSpec examples for `ResizeImageJob`: one confirming that calling `perform_later` enqueues the job with the correct `image_id`, and one confirming that `perform_now` actually calls `resize!` on the image.

Goal: Practice the enqueue-vs-actually-runs testing split from this chapter.

→ Solution

💎 Not Everything Belongs in a Job

Background jobs trade immediacy for resilience: the user stops waiting, but they also stop knowing the outcome in real time. That tradeoff is right for a confirmation email, wrong for "did my payment go through" — anything the user needs to see the result of before the page finishes loading should usually stay inline, or use a job plus a way to poll/notify the user once it completes.

🎯 What's Next

Next chapter: **APIs with Rails** — API-only mode, JSON serialization, and how Rails' REST conventions carry over when the "view" is JSON instead of ERB, contrasted with Express's REST API design chapter.

APIs with Rails

Every chapter so far has assumed Rails renders HTML — ERB views, forms, redirects. But a huge share of real Rails apps serve JSON to a separate JavaScript frontend, a mobile app, or another service entirely, with no views in sight. The REST conventions from `rails1-2`/`rails1-3` — resourceful routes, the 7 actions — carry over almost unchanged; what changes is what gets rendered at the end of each action.

⚙️ API-Only Mode

```
$ rails new my_api --api
```

`--api` strips out everything a JSON-only backend doesn't need: no view layer generators, no asset pipeline, no cookie/session/flash middleware, no CSRF protection (there's no HTML form to forge a submission from). The base controller class changes too:

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::API
end
```

`ActionController::API` (instead of `ActionController::Base`) is a deliberately smaller module list — it keeps the parts of Action Controller genuinely useful for a JSON API (strong parameters, rescue_from, before_action) and drops the HTML-rendering machinery this app will never call.

📤 Rendering JSON

```
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
def index
  posts = Post.all
  render json: posts
 end
def create
  post = Post.new(post_params)
  if post.save
   render json: post, status: :created
else
   render json: { errors: post.errors }, status: :unprocessable_entity
end
end
end
```

`render json: posts` calls `.to_json` on the object automatically, converting an Active Record relation or model instance into a JSON response with the correct `Content-Type` header set — no separate templating step required for the simplest cases. `status: :created`/`:unprocessable_entity` are the same symbol-named HTTP status codes `express1-7` covered numerically (`201`, `422`); Rails just lets you write the name instead of memorizing the number.

🧬 Serializers — Controlling the Shape of the JSON

⚠ render json: model exposes every column, including ones you don't want

`render json: post` serializes every database column on the model — including things like a soft-delete flag, an internal moderation score, or (on a `User`) a `password_digest` column that should never leave the server. This is the JSON-API equivalent of the mass-assignment problem `rails1-8`'s strong parameters solved for incoming data — except here it's the outgoing shape that needs explicit control.

```
# app/serializers/post_serializer.rb (using ActiveModel::Serializer)
class PostSerializer < ActiveModel::Serializer
 attributes :id, :title, :body, :created_at
 belongs_to :user, serializer: UserSerializer
end
# app/serializers/user_serializer.rb
class UserSerializer < ActiveModel::Serializer
 attributes :id, :name # no email, no password_digest
end
```

A serializer declares an explicit allowlist of attributes — the opposite default of "serialize everything." Once `PostSerializer` exists, `render json: post` picks it up automatically by naming convention (`Post` → `PostSerializer`, the same convention-by-class-name pattern `rails2-2`'s Pundit policies used). Nested associations (`belongs_to :user`) get their own serializer too, so a leaked column on one model can't sneak in through a different model's association.

🔢 Versioning

```
# config/routes.rb
namespace :api do
  namespace :v1 do
    resources :posts
end
end
# generates: /api/v1/posts, /api/v1/posts/:id, ...
# app/controllers/api/v1/posts_controller.rb
module Api
module V1
class PostsController < ApplicationController
# ...
end
end
end
```

`namespace` nests both the URL path (`/api/v1/...`) and the expected controller location (`Api::V1::PostsController`) in one declaration — the same `resources` convention from `rails1-2` generating all 7 routes at once, just scoped under a versioned prefix. When breaking API changes are needed later, `Api::V2::PostsController` can be added alongside the untouched `v1` namespace, so existing API consumers keep working against the version they built against.

🔑 Authenticating an API

`rails2-1` covered session-based authentication — a cookie holding a session ID, which fits a browser but doesn't fit a mobile app or a separate JS frontend well. API-only apps typically authenticate with a bearer token instead:

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::API
before_action :authenticate_request!

 private
def authenticate_request!
  token = request.headers["Authorization"]&.split(" ")&.last
  @current_user = User.find_by(auth_token: token)
  render json: { error: "Unauthorized" }, status: :unauthorized unless @current_user
end
end
```

This is a simplified static-token example — a real app more often issues a short-lived JWT here, exactly the token-based approach the completed Auth & Session Security course (`bc1-7`) covered in depth, including access/refresh rotation and safe storage. The mechanism differs from Chapter 1's session cookie, but the shape of the check — reject the request before the action runs if authentication fails — is identical.

📜 Rails APIs vs Express REST APIs

Rails (API-only) vs Express (express1-7)

Concern	Express	Rails (--api)
Routes	Manually defined per method/path	resources :posts generates all 7
Response shaping	Hand-built response objects per route	Serializer classes, reused across actions
Status codes	Numeric (res.status(201))	Symbol (status: :created)
Versioning	Manual route prefixing (/v1/posts)	namespace :api do namespace :v1
Auth	JWT middleware, hand-written	before_action, token or JWT-based

The philosophical difference is the same one that has run through this entire course: Express gives you the primitives and expects you to assemble them per project; Rails' `--api` mode is the same convention-over-configuration philosophy from `rails1-1`, just with the HTML-rendering half of the framework removed rather than the framework replaced by something new.

💻 Coding Challenges

Challenge 1: A JSON Index Action

Write an `index` action for an API-only `CommentsController` that renders all comments as JSON, and a `create` action that renders the created comment with a `201` status on success, or the comment's errors with a `422` status on failure.

Goal: Practice render json: with both success and failure status codes.

→ Solution

Challenge 2: A Serializer

Write a `CommentSerializer` that exposes only `id`, `body`, and `created_at`, plus a nested `user` association using a `UserSerializer` that exposes only `id` and `name` (no email or password data).

Goal: Practice controlling exactly what a model exposes to API consumers.

→ Solution

Challenge 3: Versioned Routes

Write the `config/routes.rb` entry that namespaces a `resources :comments` route under `/api/v1`, and give the full class name (including module nesting) the corresponding controller would need.

Goal: Practice namespacing routes and controllers together for API versioning.

→ Solution

💎 A Serializer Is Just Another Centralization Rule

This chapter's serializer allowlist, Chapter 2's Pundit policy, and Course 1's model validations are all the same underlying pattern: a rule about a model, written once, in one predictable location, applied consistently everywhere that model is touched. By this point in the course, spotting "where does this rule live, and is it written in exactly one place" should be a reflex whenever a new Rails feature comes up.

🎯 What's Next

Next chapter: **Action Cable** — Rails' WebSockets layer, for the cases where even a fast JSON API isn't enough and the server needs to push updates to connected clients in real time, contrasted with Socket.IO from express2-6.

📡 Action Cable

Chapter 5's JSON API is still request/response: the client asks, the server answers, the connection closes. Some features genuinely need the server to speak first — a new comment appearing on everyone else's screen without a refresh, a live notification badge. `express2-6` covered Socket.IO for exactly this in Express; Action Cable is Rails' own built-in answer to the same problem.

🔗 Connections and Channels

```
# app/channels/application_cable/connection.rb
module ApplicationCable
class Connection < ActionCable::Connection::Base
  identified_by :current_user
def connect
self.current_user = find_verified_user
  end
private
def find_verified_user
if (user = User.find_by(id: cookies.encrypted[:user_id]))
     user
   else
     reject_unauthorized_connection
   end
end
end
end
```

A **Connection** is established once per browser tab, the moment the WebSocket handshake happens — it's where authentication lives, reusing the same encrypted session cookie `rails2-1` already set during a normal HTTP login. `identified_by :current_user` declares that every message flowing through this connection knows which user it belongs to, and `reject_unauthorized_connection` refuses the handshake entirely for anyone who fails the check — the WebSocket equivalent of `rails2-1`'s `require_login before_action`.

A single Connection can host many **Channels** — one per feature that needs real-time updates:

```
# app/channels/comments_channel.rb
class CommentsChannel < ApplicationCable::Channel
def subscribed
  post = Post.find(params[:post_id])
  stream_for post
 end
end
```

`stream_for post` subscribes this client to a stream scoped to one specific post — Action Cable derives a unique stream identifier from the object automatically, so every browser tab currently viewing *that post's* comments joins the same stream, and tabs viewing a different post don't.

📢 Broadcasting

```
# app/models/comment.rb
class Comment < ApplicationRecord
  belongs_to :post
  after_create_commit :broadcast_comment
private
def broadcast_comment
CommentsChannel.broadcast_to(post, CommentSerializer.new(self).as_json)
  end
end
```

`after_create_commit` is a callback (from Course 1's validations/callbacks chapter, `rails1-7`) that fires only after the database transaction actually commits — broadcasting before that point risks telling clients about a comment that a later rollback then undoes. `broadcast_to(post, ...)` pushes the payload to every client streaming that specific post, reusing the exact same `CommentSerializer` Chapter 5 wrote for the JSON API — one source of truth for "what a comment looks like," whether it arrives via a REST request or a live broadcast.

💻 The Client Side

```
// app/javascript/channels/comments_channel.js
import consumer from "./consumer"

consumer.subscriptions.create(
 { channel: "CommentsChannel", post_id: postId },
 {
  received(data) {
   appendCommentToPage(data)
  }
 }
)
```

The client subscribes with the same `post_id` param the server's `subscribed` method reads, and `received` fires every time a broadcast arrives on that stream — no polling, no manually re-fetching the comments list.

📜 Action Cable vs Socket.IO

Rails Action Cable vs Express + Socket.IO (express2-6)

Concern	Express + Socket.IO	Rails Action Cable
Connection auth	io.use() middleware	Connection#connect, identified_by
Grouping clients	Rooms (socket.join(room))	Streams (stream_for / stream_from)
Sending a message	io.to(room).emit(event, data)	Channel.broadcast_to(object, data)
Feature organization	Event names on one connection	Separate channel class per feature
Triggering a broadcast	Called explicitly wherever needed	Often a model callback (after_create_commit)

The concepts map closely — a Socket.IO "room" and an Action Cable "stream" solve the same "only these clients get this message" problem. The organizational difference is Rails' usual one: a dedicated channel class per feature (`CommentsChannel`, `NotificationsChannel`) instead of a flat set of event name strings on one shared connection.

💻 Coding Challenges

Challenge 1: A Notifications Channel

Write a `NotificationsChannel` where `subscribed` streams for the current connection's `current_user` (not a param — the user identified on the connection itself).

Goal: Practice streaming scoped to the authenticated user rather than a URL parameter.

→ Solution

Challenge 2: Broadcasting on Create

Add an `after_create_commit` callback to a `Notification` model that broadcasts the new notification (serialized) to the `NotificationsChannel` stream for its owning `user`.

Goal: Practice triggering a broadcast from a model callback, tied to the right stream.

→ Solution

Challenge 3: Client Subscription

Write the JavaScript client code subscribing to `NotificationsChannel` and appending each received notification to a `#notifications` element on the page.

Goal: Practice wiring the client side of a channel subscription to a broadcast.

→ Solution

💎 Not Every Feature Needs a Socket

Action Cable is the right tool when data genuinely needs to appear without the user doing anything — a live chat, a notification badge. If a page only needs "show me the latest data when I next visit or refresh," a plain request (Chapter 5's JSON API, or a normal page load) is simpler, has one less moving part (no persistent connection to manage), and is usually the better default.

🎯 What's Next

Next chapter: **Hotwire — Turbo & Stimulus** — how modern Rails apps add interactivity to server-rendered HTML without reaching for a full frontend framework.

🔥 Hotwire: Turbo & Stimulus

Chapter 6 reached for raw WebSockets when a feature genuinely needed the server to speak first. But most "this page feels sluggish" complaints don't need that — they need faster navigation and small sprinkles of interactivity on top of pages Rails already renders. Hotwire ("HTML Over The Wire") is Rails' answer: keep rendering ERB on the server, and use a small amount of JavaScript to make that HTML feel like a modern app, without adopting a full frontend framework's component model.

🚙 Turbo Drive — Faster Navigation, No Extra Code

Turbo Drive is on by default in a new Rails app and requires no code at all: it intercepts every link click and form submission, fetches the new page in the background, and swaps in just the `<body>` — instead of the browser doing a full page reload (fresh HTML, fresh CSS/JS parse, a blank flash in between). The URL bar and history still update normally; the user experience gets SPA-like speed while every page is still a plain server-rendered response, no client-side router required.

🖼️ Turbo Frames — Independent Page Regions

```
<%= turbo_frame_tag "post_#{@post.id}" do %>
  <h3><%= @post.title %></h3>
  <%= link_to "Edit", edit_post_path(@post) %>
<% end %>
```

A `turbo_frame_tag` marks off a region of the page that can be updated independently. Clicking the "Edit" link inside it still requests the full `edit` page normally, but Turbo notices the response contains a matching `turbo_frame_tag` with the same ID, and swaps in *just that fragment* — the rest of the page (navigation, sidebar, other posts) never re-renders. The controller action itself needs no special code; the `edit` view just needs to wrap its content in a frame with the same id.

🌊 Turbo Streams — Targeted Updates from Anywhere

```
# app/controllers/comments_controller.rb
def create
@comment = Comment.new(comment_params)
 @comment.save!
 respond_to do |format|
  format.turbo_stream   # renders create.turbo_stream.erb automatically
end
end

# app/views/comments/create.turbo_stream.erb
<%= turbo_stream.append "comments", partial: "comments/comment", locals: { comment: @comment } %>
```

A Turbo Stream response is a small chunk of HTML tagged with an action (`append`, `prepend`, `replace`, `remove`) and a target element ID. This one appends a newly rendered comment partial (the same partial from `rails2-2`'s conditional-links challenge) onto an element with id `comments` — no full page reload, no hand-written JavaScript DOM manipulation of the kind `js1-8` covered.

💎 Turbo Streams Reuse Chapter 6's Action Cable Under the Hood

Rails 7's `broadcasts_to` model helper (`broadcasts_to ->(comment) { comment.post }, inserts_by: :append`) generates Turbo Stream broadcasts automatically from a model callback — the exact same `after_create_commit`-triggered broadcast pattern Chapter 6 wrote by hand for `CommentsChannel`, except the payload is a rendered HTML fragment instead of JSON, and Turbo's client-side JavaScript applies it to the DOM without any `received` callback needing to be written.

🎮 Stimulus — JavaScript Behavior on Existing HTML

```
// app/javascript/controllers/clipboard_controller.js
import { Controller } from "@hotwired/stimulus"
export default class extends Controller {
 static targets = ["source"]

 copy() {
  navigator.clipboard.writeText(this.sourceTarget.value)
 }
}



<div data-controller="clipboard">
 <input data-clipboard-target="source" value="share-link-here" readonly>
 <button data-action="click->clipboard#copy">Copy</button>
</div>
```

A Stimulus controller attaches behavior to HTML that already exists (server-rendered by ERB) rather than generating that HTML itself — `data-controller` connects the JS class to the element, `data-action` wires an event to a method, `data-*-target` gives the controller a handle on a specific child element. There's no virtual DOM, no component re-rendering, no client-side state to keep in sync with the server — the HTML is the source of truth, and Stimulus just adds small, scoped behavior on top of it.

📜 Hotwire vs a Full Frontend Framework

Hotwire (Rails) vs React/Vue/Svelte (already covered)

Concern	React/Vue/Svelte	Hotwire
Rendering	Client-side, from a JS component tree	Server-side ERB, always
State	Explicit client state, synced to the server	Server is the only source of truth
Data fetching	Client calls a JSON API, re-renders	Server pushes/returns HTML fragments directly
JS footprint	The entire UI is JavaScript	Small, scoped behavior only (Stimulus)
Best fit	Rich, highly interactive client apps	Content-driven apps wanting SPA-like feel cheaply

Neither approach is strictly better — a heavily interactive dashboard with lots of client-only state genuinely benefits from React's component model; a mostly content-driven app (a blog, a forum, an admin panel) often gets 90% of the "feels fast and modern" experience from Hotwire for a fraction of the JavaScript, while keeping every rendering decision in one place: the Rails server.

💻 Coding Challenges

Challenge 1: A Turbo Frame

Wrap a post's title and an "Edit" link in a `turbo_frame_tag` keyed to that post's id, so clicking "Edit" replaces just that region with the edit form instead of navigating to a full new page.

Goal: Practice scoping a page update to one independent region.

→ Solution

Challenge 2: A Turbo Stream Response

Write a `destroy` action on a `CommentsController` that responds with a `turbo_stream` format, and the corresponding `destroy.turbo_stream.erb` view that removes the deleted comment's element from the page.

Goal: Practice a non-append Turbo Stream action (remove) triggered from a real controller action.

→ Solution

Challenge 3: A Stimulus Controller

Write a Stimulus controller named `character-count` that updates a `<span>` with the current character count of a `<textarea>` every time its value changes, plus the HTML using `data-controller`/`data-action`/`data-*-target` to wire it up.

Goal: Practice connecting Stimulus targets and actions to existing server-rendered HTML.

→ Solution

💎 The Server Stays the Source of Truth

Every Hotwire technique in this chapter shares one property: the HTML the server renders is always correct, and the client's job is just to apply small, targeted patches to it — never to hold its own separate copy of the data that could drift out of sync. That's the core tradeoff against a full SPA framework, and the reason Hotwire fits so naturally on top of a framework already built around server-rendered convention (rails1-4's ERB views).

🎯 What's Next

Next chapter: **Deployment** — getting a Rails app onto Heroku, Render, or Docker, asset precompilation, and production database configuration, closing out this course.

🚀 Deployment

Every chapter in this course — auth, authorization, tests, jobs, an API, WebSockets, Hotwire — has been building an app that only matters once real people can reach it. This closing chapter gets a Rails app from a laptop onto a real server, with the production-specific concerns each earlier chapter's development setup quietly skipped over.

🌍 Environments

```
# config/environments/production.rb (excerpts)
config.eager_load = true # load all classes at boot, not lazily per request
config.cache_classes = true # never reload code after boot -- dev's auto-reload is a dev-only convenience
config.force_ssl = true # redirect all HTTP to HTTPS, set HSTS -- ties back to the completed HTTPS course
config.log_level = :info
```

Rails has always run in one of three environments — `development`, `test` (both used constantly throughout this course without being named explicitly), and `production` — each with its own config file under `config/environments/`. `force_ssl` is a direct callback to the completed HTTPS/TLS course: one config line gets an entire app the redirect-to-HTTPS and HSTS-header behavior that course covered manually with nginx.

🔑 Credentials — Rails' Built-In Secrets Management

```
$ rails credentials:edit


# accessed anywhere in the app:
Rails.application.credentials.dig(:stripe, :secret_key)
```

`rails credentials:edit` opens an editor on an encrypted `config/credentials.yml.enc` file, decrypted using a `config/master.key` that's git-ignored by default and provided to the production server separately (as an environment variable, or a mounted secret). This is Rails' equivalent of the `.env`-file approach `node2-6` covered for Express — except the secrets themselves are encrypted and safe to commit, and only the one small key needs careful handling, rather than an entire plaintext file of values.

📦 Asset Precompilation

```
$ RAILS_ENV=production rails assets:precompile
```

The Stimulus controllers and CSS from `rails2-7` are development-friendly source files — in production they need to be bundled, minified, and fingerprinted (a content hash added to each filename, so browsers can cache them aggressively and safely bust that cache the moment a file changes). `assets:precompile` does all of this in one step, producing the static files actually served in production. This is one more thing Express's plain `express.static` (covered in `express1-5`) never needed — Express serves whatever files are already sitting in a directory, with no separate build step assumed by the framework itself.

🗄️ Production Database Configuration

```
# config/database.yml
production:
 adapter: postgresql
 url: <%= ENV["DATABASE_URL"] %>
 pool: <%= ENV.fetch("RAILS_MAX_THREADS", 5) %>
```

Rather than hardcoding host/username/password, production reads a single `DATABASE_URL` — the same connection-string shape covered for raw `mysql2` pooling in `node2-5` — supplied by whatever hosts the database. `pool` caps how many concurrent connections each server process opens, matched to the app server's thread count so connections aren't wasted sitting idle or, worse, exhausted under load.

Deploying to Heroku

```
# Procfile
web: bundle exec puma -C config/puma.rb
release: rails db:migrate


$ git push heroku main
```

Heroku reads the `Procfile` to know how to run the app: a `web` process serving requests, and a `release` process that runs once per deploy, before traffic switches over — the natural place for `rails db:migrate`, so every deploy automatically applies any pending migrations from `rails1-5` before the new code goes live.

🐳 Deploying with Docker

```
# Dockerfile (multi-stage, matching express2-8's / node3-8's pattern)
FROM ruby:3.3-slim AS build
WORKDIR /app
COPY Gemfile Gemfile.lock ./
RUN bundle install --deployment
COPY . .
RUN RAILS_ENV=production rails assets:precompile

FROM ruby:3.3-slim
WORKDIR /app
COPY --from=build /app ./
CMD ["bundle", "exec", "puma", "-C", "config/puma.rb"]
```

Multi-stage builds keep the same principle already established for Node in `node3-8`: the first stage installs gems and precompiles assets, pulling in build tools and source files the running app doesn't need; the second stage copies over only the finished result, producing a smaller, leaner production image. The Sidekiq worker process from `rails2-4` gets its own separate Dockerfile/service in the same deploy, since it needs to run continuously alongside the web process, not inside a single request/response cycle.

📜 Rails Deployment vs Express Deployment

Rails vs Express (express2-8)

Concern	Express	Rails
Secrets	.env / dotenv (node2-6)	Encrypted credentials.yml.enc + master.key
Static assets	Served as-is, no build step assumed	Precompiled, fingerprinted, minified
DB migrations on deploy	Run manually or in a custom CI step	Procfile release phase (Heroku), or explicit step
Process manager	PM2 cluster mode (express2-8)	Puma (threaded/clustered web server)
Background workers	Separate Node process (node3-6/node3-8)	Separate Sidekiq process (rails2-4)

The shapes converge more than they differ: both frameworks end up with a web process, a separate worker process for background jobs, secrets injected via environment/config rather than hardcoded, and a migration step that runs before traffic hits new code. Rails simply bakes several of these decisions (credentials encryption, asset fingerprinting, the release-phase convention) into the framework itself, rather than leaving each project to assemble its own approach.

💻 Coding Challenges

Challenge 1: Production Environment Config

Write the `config/environments/production.rb` lines that enable eager loading, force SSL redirection, and set the log level to `:info`. Briefly state, in a comment, why each of the first two would be wrong to enable in `development.rb`.

Goal: Practice reasoning about which settings genuinely differ between environments, and why.

→ Solution

Challenge 2: A Heroku Procfile

Write a `Procfile` with a `web` process running Puma, a `release` process running pending migrations, and a `worker` process running Sidekiq (from `rails2-4`).

Goal: Practice declaring all three process types a real deployed Rails app needs.

→ Solution

Challenge 3: Reading a DATABASE_URL

Write the `production:` section of `config/database.yml` reading its connection info entirely from `ENV["DATABASE_URL"]`, with a connection pool size read from an `RAILS_MAX_THREADS` environment variable, defaulting to `5` if unset.

Goal: Practice configuring the database from environment variables rather than hardcoded values.

→ Solution

💎 Deployment Is Where Every Earlier Chapter Gets Paid Off

Nothing in this chapter is really new engineering — it's making sure everything already built across this course (auth cookies needing HTTPS, Sidekiq needing its own process, migrations needing to run before new code serves traffic) actually holds up outside a development laptop. A feature that only works in development isn't finished; this chapter is what "finished" actually means.

🎓 Course Complete

This closes out Ruby on Rails Intermediate/Advanced — and with it, the full Ruby on Rails track (16 chapters across both courses), built on top of the full 16-chapter Ruby language track. From `rails1-1`'s first `rails new` to a tested, authorized, real-time, deployed API: convention over configuration, followed all the way through.