A GUIDE FOR BACKEND DEVELOPERS · BOOK TWO

format::json
db migrate
BackgroundQueue
Hooks
auth::JWT
AppContext
Routes::new()
Tera
scheduler.yaml
cargo loco generate
boot_test
mail_template
SeaORM
find_related
Task
perform_later
ViewEngine
doctor
ActiveModel
hx-patch
Initializer
sitemap.xml
Set(..)
shared_store

Loco

for Laravel Developers

Build a blog in Rust from scratch:
from the first command to a Docker deploy

```
terminal
$ php artisan make:model Post -m
$ cargo loco generate model posts title:string! slug:string^
```

13 steps · a blog from zero to production · Loco 1.3

2026

AXVILAB · GUIDES FOR BACKEND DEVELOPERS · BOOK TWO

Loco for Laravel Developers

Build a blog in Rust from scratch: from the first command to a Docker deploy

Vitaly Ax

First edition · November 2026 · version 1.0

axvi.dev

© 2026 Vitaly Ax, AxviLab. All code samples are MIT-licensed. Text and illustrations — all rights reserved; redistribution without the author's permission is prohibited.

Examples were written on Loco 1.2 and re-tested on 1.3.1. The framework moves fast: if a command or signature differs from the book, trust `cargo loco --help` and the code, and write to the author — it will be fixed in the next edition.

Introduction

Building a blog in Rust from scratch: from `loco new` to a Docker image you can put on a server



Who this book is for

For a Laravel developer who has already read "Rust for Laravel Developers" or picked up the basics of the language some other way. You know what ownership is, along with `Result` and `?`, traits, `async` and `.await`. Maybe you've seen Axum or sqlx, but that's not required: I'll explain everything you need from Axum, SeaORM and Tera. What you haven't done yet is open Loco.

Loco is a Rust web framework that its authors modeled on Rails. That's why a Laravel developer will recognize almost everything in it: generators instead of `php artisan make:*`, migrations, models, workers for queues, mailers, tasks, a scheduler, per-environment config. Under the hood are familiar crates: Axum for HTTP, SeaORM for the database, Tera for templates, Tokio for async.

There's very little explanation of the language itself here. But every construct from Loco and SeaORM that's new to you (a macro, an attribute, a trait) gets explained where it first appears. If I have to show something before I can explain it, I'll say so directly: "just copy this for now, the explanation is in such-and-such chapter."

What we're building

One real application, from the first command to deployment: a blog. Not "hello world" and not a TODO list, but a site you could actually run for yourself:

- **posts** with drafts, scheduled publishing and readable URLs (slugs);
- **tags** with a many-to-many relation and tag pages;
- **authors and login** through the browser: a login form, a JWT token in a cookie, protected pages;
- **an admin area**, written by hand: a list, forms, validation errors, flash messages;
- **image uploads** for post covers;
- **background work**: processing a post after it's saved, emails to subscribers;
- **a scheduler and tasks**: publishing scheduled posts on a timer, cleaning up junk;
- **RSS and a sitemap** for feed readers and search engines;
- **caching** of rendered pages, ETags and a check under load;
- **tests** for models, requests, workers and tasks, plus snapshots of pages and emails;
- **deployment** in Docker: Postgres, separate processes for the web server, the worker and the scheduler.

Posts take two chapters, everything else takes one each. The chapters follow the order you build things in:

1. "First run: the environment and the generated app": Docker, `loco new` and a map of the folders.
2. "The posts model: migration, entity and first logic": the `posts` table, slugs, validation.
3. "Public pages": a controller, Tera, pagination, Markdown.
4. "Tags": many-to-many without N+1.
5. "Authors and login": a post's author, logging in through a form, JWT in a cookie.
6. "An admin area with HTMX": CRUD by hand.
7. "Cover image uploads": file storage and multipart.
8. "Background work": a worker, a mailer, subscribers.
9. "Scheduler and tasks": a real queue and a schedule.
10. "RSS and sitemap": XML responses.
11. "Caching and speed": HTML caching, ETags, load tests.
12. "Tests": the whole testing toolkit and snapshots.
13. "Deploy": a Dockerfile, compose, graceful shutdown.
14. "What's next": an honest comparison with Laravel and ideas for where to go from here.

How the book works

One chapter is one step. One step is one git tag in the public repository with the blog's code: https://github.com/axvilab/loco-for-laravel-devs.

The tags look like `step-01-new`, `step-02-posts-model`, `step-03-public-pages` and so on. So you don't have to read the book in order. Want to jump straight to the public pages? Clone the repository and check out the tag you need:

```
git clone https://github.com/axvilab/loco-for-laravel-devs.git
cd loco-for-laravel-devs
git checkout step-03-public-pages
```

The application lives in the `blog/` subfolder. Next to it is a `notes/` folder with my working notes: commands, real output, errors I ran into. The book was built from those notes. Every command in the text was actually run, and every error actually happened.

Inside a chapter, the order is:

1. what we're doing in this step and why;
2. the commands and their real output (trimmed a little so it doesn't eat the page);
3. the code. Long files are shown in pieces with `// ...`, and the full version is in the tag;
4. "**How Laravel does it.**": a comparison with what you already know;
5. "**Why it's like this.**": after every difference that might seem odd;
6. "A nuance the tutorials skip" sidebars: small things people trip over in their first week;
7. "Try it": exercises on the same blog, from simple to harder.

Versions

Everything in the book was tested on these versions:

What	Version
Rust	1.97
Loco (`loco-rs` and the `loco` CLI)	1.2.0, re-tested on 1.3.1
SeaORM	2.0
Tera	2
Axum	0.8
Database	SQLite in development, PostgreSQL 16 in deployment

About Loco 1.3. Version 1.3 came out in November 2026, when the book was already written. The commands and their output here were recorded on 1.2.0. After that I rebuilt every step of the blog on 1.3.1: it compiles and all the tests pass with no changes to the code. Where 1.3 behaves differently for us, the chapter says so.

Heads up: most of the internet describes Loco 0.x. Loco 1.0 came out at the end of July 2026, and 1.2 in September. Articles and videos are mostly about older versions, and their code often doesn't compile. The most noticeable changes:

- SeaORM 2.0: primary keys are now `i64`, and some methods got new names;
- Tera 2: an undefined variable in a template is now an error, not an empty string; there are no macros, components replaced them; there's no `date` filter;
- the HTML scaffold generator is gone. `generate scaffold` only produces a JSON API, so we write the admin area by hand;

- route parameters are written as `/{id}`, not `/:id` (that one is an Axum 0.8 change).

If the book and reality disagree, trust `cargo loco --help`, the code in the tag and the crate's source. The version number in `Cargo.lock` will tell you who's right.

Laravel → Loco in one table

Laravel	Loco
`laravel new blog`	`loco new --name blog ...`
`php artisan ...`	`cargo loco ...`
`php artisan make:model Post -m`	`cargo loco generate model posts title:string! ...`
`.env` + `config/*.php`	`config/development.yaml`, `test.yaml`, `production.yaml`
`bootstrap/app.php`, providers	`src/app.rs` (the `Hooks` trait)
Eloquent	SeaORM: `Entity`, `Model`, `ActiveModel`
Blade	Tera
Job + `queue:work`	worker (`BackgroundWorker`)
Artisan command	task (`cargo loco task ...`)
`routes/console.php`, scheduler	`scheduler` in the YAML config
`php artisan serve`	`cargo loco start`

We'll go through each row in detail in the matching chapter.

How long compilation takes

Fair warning: Rust takes a long time to compile, and the first time takes a very long time.

On the two-core machine this book was written on:

What	Time
`cargo install loco`	~2 min 20 s
dev image with `loco` and `sea-orm-cli` (Docker)	~4 min 20 s
first build of a new app	3 to 10 min
first `cargo test` after the build	~1 min 40 s
build after a change in `src/`	30–90 s

The spread on the first build is real: 3 min 16 s in Docker on an idle machine, and 9 min 35 s without Docker, while something else was compiling in parallel and full debug info was going into the binary. In "Scheduler and tasks" we'll cut down the debug info, and a clean build drops to 2 min 58

s. On a laptop with 8+ cores, the first build takes 2–3 minutes. Later builds are faster because the dependencies are already compiled and sitting in the `target/` folder. Most of the time after an edit goes to linking.

Hence a practical tip: don't delete `target/` unless you have to. In "First run: the environment and the generated app" we'll set up Docker so this folder lives in its own volume and survives container restarts.

Let's go.

Contents

	Preface	9
1	First run: the environment and the generated app	10
2	The posts model: migration, entity and first logic	27
3	Public pages	52
4	Tags	70
5	Authors and login	85
6	An admin area with HTMX	102
7	Cover image uploads	118
8	Background work	131
9	Scheduler and tasks	148
10	RSS and sitemap	163
11	Caching and speed	176
12	Tests	190
13	Deploy	203
	What's next	217

Preface

I wrote the first book to get my head around the language. This one is about building a real application in that language without getting lost.

The fastest way to learn a framework isn't the documentation. It's one real thing that you take all the way to the finish. Remember your first Laravel project? You probably didn't read the docs from the first page to the last. You built something of your own and looked things up as you went. Migrations made sense when you needed to add a column. Queues made sense when an email started slowing down the response. Caching made sense when a page started loading slowly.

So this book isn't a tour of Loco's features. It's one blog, from `loco new` to a Docker image. Posts, tags, authors, an admin area, images, background jobs, scheduling, RSS, caching, tests, deployment. Each topic shows up when the blog actually needs it, not because it's next in alphabetical order.

I picked a blog on purpose. Everyone knows the domain, so you can think about the framework instead of the business logic. And a blog still has everything you run into at work: relations between tables, authorization, forms with errors, files, queues, cron.

I went through every step myself first. The commands, the output and the compiler errors are real, from my machine. Where Loco behaved differently from what the documentation promises, I said so. Those are the spots that cost you the most time when you hit them on your own.

If after this book you can start your next project on Loco with the same confidence you have with Laravel, then it did its job.

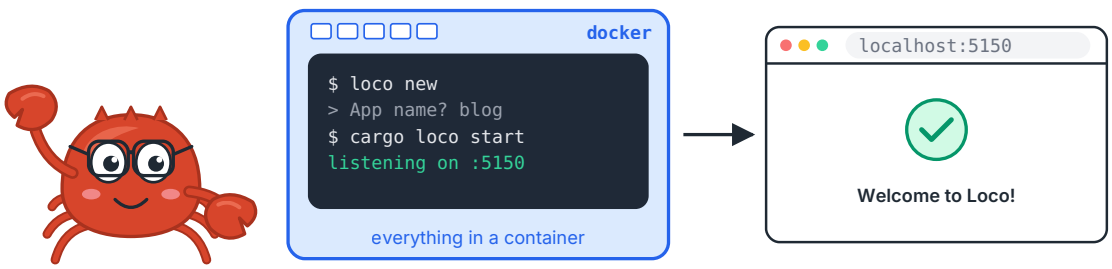
Vitaly Ax 2026

1. First run: the environment and the generated app

> **Why this matters.** In one step you get a working environment in Docker and an app that responds at `localhost:5150`. Plus a map of Loco's folders tied to the Laravel folders you know, so the following chapters don't look like a pile of magic files.
> **What breaks without it.** Forget `BINDING: 0.0.0.0` in Docker, and the server says `listening` while the browser gets `Connection reset by peer`.
> Delete an anchor comment like `// tasks-inject (do not remove)` in `src/app.rs`, and the next `cargo loco generate` fails because it can't find where to insert code.

Code after this step: github.com/axvilab/loco-for-laravel-devs/tree/step-01-new

First run: one command, and the app already answers



In this step we install the tools, generate the blog application and run it. We don't write any code yet. The goal is different: to understand what exactly Loco generated, where everything lives and how it maps to Laravel's folders. Without that map, every following chapter would look like a pile of magic files.

By the end of the step you'll have:

- a development environment in Docker (or without it, if you prefer);
- a `blog` application that responds at `http://localhost:5150`;
- an understanding of what each folder is for and what `cargo loco` is.

The code for this step is in the `step-01-new` tag.

What you need to install

Three things:

Tool	Version	What it's for	Laravel equivalent
Rust (`rustc`, `cargo`)	1.97	compiler and package manager	PHP + Composer
`loco`	1.2.0	creates a new app (`loco new`)	the `laravel` installer
`sea-orm-cli`	2.0.4	reads the database schema and generates Rust structs	—

You only need `loco` for `loco new`. Everything else (generators, migrations, running the server) is done with the `cargo loco` command inside the app. Just like you need the `laravel` installer once and then work with `php artisan`.

You won't call `sea-orm-cli` yourself. Loco runs it after a migration to update the Rust description of your tables. Details are in "The posts model: migration, entity and first logic".

There are two ways to go: Docker, or everything locally. I recommend Docker. That way nothing but Docker ends up on your machine, and the versions of Rust and the generators are pinned in one file.

Option 1: Docker

Create an empty folder for the project:

```
mkdir loco-blog && cd loco-blog
```

Put two files in it. The first is the development image, `Dockerfile.dev`:

```
# Dev image: Rust + Loco generators.
FROM rust:1.97

# The official image installs Rust with the minimal profile: no rustfmt or clippy.
# `loco new` runs `cargo fmt`, and we'll need clippy later.
RUN rustup component add rustfmt clippy

# Install the generators once, at image build time (a few minutes).
RUN cargo install --locked loco@1.2.0 sea-orm-cli@2.0.4 \
    && rm -rf /usr/local/cargo/registry

# A user with the same UID as on the host: files from the container will be yours.
ARG UID=1000
ARG GID=1000
RUN groupadd -o -g ${GID} dev \
    && useradd -o -m -u ${UID} -g ${GID} dev \
```

```
    && mkdir -p /usr/local/cargo/registry /target \
    && chown dev:dev /usr/local/cargo/registry /target

USER dev
# target/ lives in its own volume, not in the project folder.
ENV CARGO_TARGET_DIR=/target
WORKDIR /app
EXPOSE 5150
```

What's going on here:

- **`FROM rust:1.97`**: the official Rust image. The version in the tag is pinned, so six months from now the image builds the same way. The official image installs Rust with the minimal profile, so we add `rustfmt` and `clippy` separately.
- **`cargo install --locked loco@1.2.0 sea-orm-cli@2.0.4`**: installs both generators at specific versions. `--locked` means "use exactly the dependencies from the published package's `Cargo.lock`", so the build won't break because some dependency shipped a fresh version. After installing, we delete the cache of downloaded crates to keep the image smaller. Loco 1.3.1 is already out, but we keep the generator at 1.2.0 on purpose: that way the files it creates match the book line by line. Moving the finished app to 1.3 takes one command, `cargo update -p loco-rs`, and the blog from this book works on it as is.
- **`ARG UID`** / **`useradd`**: inside the container we don't run as root but as a user with your UID. Otherwise, on Linux, every file the generator creates would belong to root, and your editor on the host couldn't save it.
- **`CARGO_TARGET_DIR=/target`**: the folder with compiled dependencies moves out of the project into its own volume. The reason is just below.

The second file is `docker-compose.yml`:

```
services:
  dev:
    build:
      context: .
      dockerfile: Dockerfile.dev
      args:
        UID: ${UID:-1000}
        GID: ${GID:-1000}
    working_dir: /app/blog
    command: cargo loco start
    environment:
      BINDING: 0.0.0.0
    ports:
      - "5150:5150"
```

```
    volumes:
      - .:/app
      - cargo-registry:/usr/local/cargo/registry
      - cargo-target:/target

volumes:
  cargo-registry:
  cargo-target:
```

- **`.:/app`**: the project folder is mounted into the container. You edit files on the host, the container sees them.
- **`cargo-registry`**: a volume for downloaded crates. Without it, every new container would download them again.
- **`cargo-target`**: a volume for `target/`, meaning the compiled dependencies. This is the most important line in the file: without it, every `docker compose up` would start those first few minutes of compilation from scratch.
- **`BINDING: 0.0.0.0`**: by default Loco listens only on `localhost`. Inside a container that means "only from inside the container". Details are in the nuance below.
- **`working_dir: /app/blog`**: the app will live in the `blog/` subfolder, like in the book's repository.

If your UID on Linux isn't 1000 (check with `id -u`), create a `.env` file next to it with the lines `UID=...` and `GID=...`. Docker Compose will substitute them into `args`. On macOS and Windows with Docker Desktop, file ownership is mapped automatically, and you can skip this.

Build the image:

```
docker compose build
```

On a two-core machine the build took 4 min 17 s: `loco` compiled in 1 min 52 s, and `sea-orm-cli` took another two and a half. It's a one-time cost: until you change `Dockerfile.dev`, the image comes from the cache.

Generate the app. We run the command from the `/app` folder, not `/app/blog`, because `blog` doesn't exist yet:

```
docker compose run --rm -w /app dev \
  loco new --name blog --db sqlite --bg async --assets serverside --path .
```

```
🚂 Loco app generated successfully in:
/app/blog
```

Under a second. The app template is baked into the `loco` binary, so this step doesn't need the internet. We'll go through the flags below, in the section on `loco new`.

Now run it:

```
docker compose up
```

The first run compiles the whole app with all its dependencies, over 600 crates. In Docker on two cores this took 3 min 16 s. Later runs with no code changes start in a second. When you see

```
listening on http://0.0.0.0:5150
```

open `http://localhost:5150` in your browser. To stop it, press Ctrl-C or run `docker compose stop` from another terminal.

Run any other command the same way, through `run`:

```
docker compose run --rm dev cargo loco routes
docker compose run --rm dev cargo test
```

From here on, I write commands in short form, without `docker compose run --rm dev`. If you're working in Docker, add that prefix in your head.

A nuance the tutorials skip: `Connection reset by peer` and `BINDING`

If you remove `BINDING: 0.0.0.0` from `docker-compose.yml`, the server starts without errors and honestly writes to the log:

```
listening on http://localhost:5150
```

But `curl` from the host gets:

```
$ curl -i localhost:5150/_ping
curl: (56) Recv failure: Connection reset by peer
```

`localhost` inside a container is the container itself. Docker forwards the request from the host's port 5150 to the container's network interface, and nobody is listening there. In `config/development.yaml` the address is set like this:

```
binding: <%= get_env(name="BINDING", default="localhost") %>
```

`get_env` reads an environment variable, and if it isn't set, uses the default. That's why the `BINDING=0.0.0.0` variable in compose fixes the problem without touching the config. The `<%= ... %>` syntax is covered below, in the section on config.

A nuance the tutorials skip: don't use the `-slim` image

It's tempting to use `rust:1.97-slim` because it's smaller. But the slim image has no OpenSSL headers, and `sea-orm-cli` won't build without them. The error looks like this:

```
warning: openssl-sys@0.9.116: Could not find directory of OpenSSL installation, ...
error: failed to run custom build command for `openssl-sys v0.9.116`
error: failed to compile `sea-orm-cli v2.0.4`
```

The full `rust:1.97` image already has the packages you need. If you really want slim, add the line `RUN apt-get update && apt-get install -y pkg-config libssl-dev` before `cargo install`.

One more small thing. With `--locked`, cargo prints warnings like `package chacha20 v0.10.0 in Cargo.lock is yanked`. That means the dependency's author pulled that version after `sea-orm-cli` was published. It doesn't affect the build.

Option 2: without Docker

If you don't want Docker, install Rust through rustup (https://rustup.rs) and the generators through cargo:

```
cargo install loco          # 2 min 18 s on two cores
cargo install sea-orm-cli   # a few more minutes
```

Both install from crates.io without warnings. After that everything is the same, just without the `docker compose run` prefix:

```
mkdir loco-blog && cd loco-blog
loco new --name blog --db sqlite --bg async --assets serverside --path .
cd blog
cargo loco start
```

Without Docker you don't need `BINDING`: a server on `localhost` is reachable from the same computer.

One tip from experience. If you want to stop the server from another terminal, don't run `pkill -f "blog-cli start"`. `pkill -f` matches against the full command line of every process, and your own shell has that string in its command too. For me, that command killed the server along with the shell I ran it from. A safer way: `pgrep -f target/debug/blog-cli`, then `kill <pid>`. Or just Ctrl-C in the server's terminal.

`loco new`: the flags

```
loco new --name blog --db sqlite --bg async --assets serverside --path .
```

Without flags, `loco new` starts an interactive dialog. With all three flags `--db`, `--bg` and `--assets`, the dialog is skipped entirely. That's handy for scripts and for a book, where the command needs to be reproducible.

Flag	What it means	Our choice	Laravel
`--name blog`	app and folder name	`blog`	`laravel new blog`
`--db`	database: `sqlite`, `postgres`, `none`	`sqlite`	`DB_CONNECTION=sqlite`
`--bg`	background jobs: `async`, `queue-redis`, `queue-postgres`, `queue-sqlite`, `blocking`	`async`	`QUEUE_CONNECTION`
`--assets`	how the frontend is served: `serverside`, `clientside`, `none`	`serverside`	Blade templates
`--path .`	where to create the app folder	current folder	—

A few notes:

- **`--bg async`**: background jobs run inside the same process, as separate Tokio async tasks. You don't need to start a separate worker. That's enough to begin with. In the config this choice is recorded as `workers.mode: BackgroundAsync`. In "Scheduler and tasks" we'll switch to a queue in SQLite.
- **`--assets serverside`**: pages are rendered on the server with Tera templates, like Blade in Laravel. `clientside` sets the app up for a separate client-side frontend (an SPA).
- **`--path .`**: `loco new` creates the folder `<path>/<name>`, i.e. `./blog`. That's why in the book's repository the app sits in `loco-blog/blog/`, with room next to it for notes and Docker files.

There's no separate "app type" flag. In the dialog you see options like "SaaS App with server side rendering", but the type is really just a combination of the three flags. "SaaS with server-side rendering" = `--db sqlite|postgres --assets serverside`. That's the closest equivalent to `laravel new blog` with Blade and authentication out of the box.

A nuance the tutorials skip: `loco new` inside a git repository

I created the app first and ran `git init` afterwards. If you do it the other way round, `loco new` notices the folder is already in a git repository and asks whether to continue. In a normal terminal you'd answer "yes". But without a terminal, in CI or a script, there's nobody to ask, and the command fails with a cryptic error:

```
$ git init -q && loco new --name x --db sqlite --bg async --assets serverside --path .
Error: Dialog(IO(Custom { kind: NotConnected, error: "not a terminal" }))
```

`not a terminal` is the dialog having nowhere to print its question. The fix: the `-a` flag (`--allow-in-git-repo`), or run `loco new` first and `git init` after.

What got generated

The app is ready, and it already has registration, login, password reset, a mailer, an example worker and two tasks. Here's a map of the files that matter at the start:

Path	Laravel equivalent	What it is
`Cargo.toml`	`composer.json`	dependencies: `loco-rs 1.2`, `sea-orm 2.0`, `axum 0.8`, `validator` and others
`.cargo/config.toml`	—	the `cargo loco` alias
`src/bin/main.rs`	`artisan` + `public/index.php`	entry point of the single binary
`src/app.rs`	`bootstrap/app.php`, providers, `routes/*.php`	registration of everything: routes, workers, tasks
`src/controllers/`	`app/Http/Controllers`	`auth.rs` (JSON authentication), `page.rs` (home page)
`src/models/_entities/`	—	table descriptions in Rust, generated from the database
`src/models/users.rs`	`app/Models/User.php`	user model logic
`src/views/`	API Resources	structs for JSON responses
`src/mailers/`	Mailables + email templates	templates are baked into the binary
`src/workers/`	`app/Jobs`	an example background job
`src/tasks/`	`app/Console/Commands`	the `user:create` and `user:delete` tasks
`src/initializers/`	service providers	Tera template engine setup
`src/fixtures/`	seeders	YAML with test data
`migration/`	`database/migrations`	a separate crate with migrations
`assets/views/`	`resources/views`	Tera templates
`assets/static/`	`public/`	static files, served at `/static/...`
`config/*.yaml`	`.env` + `config/*.php`	config for each environment
`tests/`	`tests/Feature`, `tests/Unit`	a single crate with integration tests

Plus something Laravel doesn't have: `loco new` drops a framework reference guide right into the project, with principles, recipes and an API index. I looked into it more than once. But it has some inaccuracies for version 1.2. I'll point out a few of them in later chapters.

`cargo loco` is `cargo run --`

In Laravel, `artisan` is a PHP script in the project root. Loco has no separate script. Look at `.cargo/config.toml`:

```
[alias]
loco = "run --"
```

It's a cargo alias: `cargo loco start` really means `cargo run -- start`. In other words, "compile the app and run it with the `start` argument". That has an important consequence: **every `cargo loco` command compiles your app first.** Even `cargo loco routes`.

Why it's like this. Generators, migrations, tasks, the route list: all of these are code in the app itself, not a separate utility. Only something that knows your controllers can build the route list, and only the compiled binary knows them.

One binary for everything: `src/bin/main.rs`

```
use blog::app::App;
use loco_rs::cli;
use migration::Migrator;

#[tokio::main]
async fn main() -> loco_rs::Result<()> {
    cli::main::<App, Migrator>().await
}
```

You know `#[tokio::main]` from the first book: it starts the async runtime. `cli::main::<App, Migrator>()` is the whole entry point. `cli::main` is a generic function, and the `::<App, Migrator>` after its name spells out the type parameters explicitly. This syntax is called the "turbofish"; you may have seen it in `collect::<Vec<_>>()`. Here it hands Loco two types: your app and the list of migrations. From there Loco parses the command-line arguments itself: `start` runs the server, `db migrate` runs migrations, `task` runs tasks, `generate` runs generators.

In Laravel you have `public/index.php` for web requests and `artisan` for the console. In Loco it's one `blog-cli` binary, and in production you'll run the same thing with different arguments.

The heart of the app: `src/app.rs`

```rust
#[async_trait]
impl Hooks for App {
    async fn initializers(_ctx: &AppContext) -> Result<Vec<Box<dyn Initializer>>> {
        Ok(vec![Box::new(initializers::view_engine::ViewEngineInitializer)])
    }
    fn routes(_ctx: &AppContext) -> AppRoutes {
        AppRoutes::with_default_routes() // /_ping, /_health, /_readiness
            .add_route(controllers::auth::routes())
            .add_route(controllers::page::routes())
    }
    async fn connect_workers(ctx: &AppContext, queue: &Queue) -> Result<()> {
        queue.register(DownloadWorker::build(ctx)).await?;
        Ok(())
    }
    fn register_tasks(tasks: &mut Tasks) {
        // tasks-inject (do not remove)
        tasks.register(tasks::user_create::UserCreate);
        tasks.register(tasks::user_delete::UserDelete);
    }
    // ... boot, truncate, seed
}
```

`Hooks` is a Loco trait, and `App` is your empty struct that implements it. Each method of the trait is a place Loco comes to for a particular piece of information: "what routes do you have?", "what workers?", "what tasks?". In Laravel terms, it's a mix of `bootstrap/app.php`, service providers and the `routes/` files.

Why it's like this. In PHP the framework finds classes on its own: it scans folders, reads `composer.json`, autoloads. Rust has no autoloading. The compiler only knows about what's explicitly mentioned in the code. So everything that's supposed to work has to be registered in `app.rs` by someone. Loco's generators do it for you: they add lines in the right places.

How does a generator find "the right place"? By anchor comments like `// tasks-inject (do not remove)`. A new task gets inserted above that line. **Don't delete these comments.** Without the anchor, the next `cargo loco generate task` fails with an error that names the missing anchor (before 1.0 it failed silently).

A few new constructs in this snippet:

- **`#[async_trait]`** above the `impl`: an attribute from the `async-trait` crate. In the first book we mentioned it as the "old way" to write async methods in traits. Loco still uses it because its traits are used through `Box<dyn ...>`, and native async trait methods don't work with `dyn`. The rule is simple: if a framework trait is declared with `#[async_trait]`, your `impl` must have the same attribute. Otherwise the compiler will tell you the method signatures don't match.

- **`AppContext`** (the `ctx` parameter in the code): the application context. It's a struct Loco creates at startup and passes everywhere shared resources are needed. It holds the database connection (`ctx.db`), the config (`ctx.config`), the cache (`ctx.cache`), file storage (`ctx.storage`),
  the queue and the mailer. In Laravel you get the same things from the container: `app('db')`, `config(...)`, facades. There's no container here: the context is passed explicitly, as a parameter. The underscore in `_ctx` means "the parameter is there, but I don't use it", which keeps the compiler from warning about an unused variable.
- **`Vec<Box<dyn Initializer>>`**: a list of initializers. Each initializer is its own type, so to go into one shared `Vec` they're put in `Box<dyn ...>` "boxes". We'll see this trick from the first book again in the list of migrations.

The home page

The controller `src/controllers/page.rs`:

```
#[debug_handler]
async fn index(ViewEngine(v): ViewEngine<TeraView>) -> Result<Response> {
    format::render().view(&v, "home/hello.html", data!({}))
}

pub fn routes() -> Routes {
    Routes::new().add("/", get(index))
}
```

This is a regular Axum handler. You know `#[debug_handler]` from the first book: an attribute that makes compiler errors in handlers readable. `ViewEngine(v)` pulls the template engine out of the request, and `format::render().view(...)` renders the `assets/views/home/hello.html` template with empty data. Controllers and templates are covered in detail in "Public pages".

This controller only appeared in version 1.2. Before that, a fresh app with `--assets serverside` returned a 404 on `/`. If an old article shows an empty app with no home page, it's about 1.0 or 1.1.

Config: YAML instead of `.env`

`config/development.yaml`, `test.yaml` and `production.yaml`: one file per environment. Values from the environment are substituted like this:

```
database:
  uri: <%= get_env(name="DATABASE_URL", default="sqlite://blog_development.sqlite?
mode=rwc") %>
  auto_migrate: true
```

`<%= get_env(name="X", default="...") %>` is your `env('X', '...')`. The environment is chosen by the `LOCO_ENV` variable or the `-e` flag; the default is `development`.

Watch the syntax. In old articles you'll see `{{ get_env(...) }}`. In Loco 1.x it still works, but it prints a deprecation warning. Write `<%= ... %>`.

Running it and first checks

Start the server:

```
cargo loco start
```

Here's the start of the log (colors removed, lines shortened):

```
WARN app: loco_rs::boot: pretty backtraces are enabled (this is great for development but
has a runtime cost for production...)
INFO app: loco_rs::db::migrate: auto migrating
INFO app: sea_orm_migration::migrator::exec: Applying migration 'm20220101_000001_users'
INFO app: loco_rs::boot: initializers loaded initializers="view-engine"
INFO app: loco_rs::controller::app_routes: [GET] /_ping
...
INFO app: loco_rs::controller::app_routes: +middleware name="limit_payload"
...
environment: development
   database: automigrate
     logger: debug
compilation: debug
      modes: server

listening on http://localhost:5150
```

What's interesting here:

- **`auto migrating`**: `development.yaml` has `auto_migrate: true`, so the server applies new migrations on startup by itself. It's like `php artisan migrate` before every `serve`. We'll turn this off in production.
- **The database file** `blog_development.sqlite` appeared in the `blog/` folder. The `?mode=rwc` parameter in the connection string means "create the file if it doesn't exist". Tests get their own `blog_test.sqlite`.
- **Port 5150** is the default for every Loco app. If you run two apps at once, move one of them: `PORT=5151 cargo loco start`.

Check a built-in route:

```
$ curl -i localhost:5150/_ping
HTTP/1.1 200 OK
content-type: application/json
x-request-id: d783d8b8-d178-421d-aed0-cab74d76ef50
x-powered-by: loco.rs
content-length: 11

{"ok":true}
```

How Laravel does it. Laravel 11 has the `/up` route. Loco has three: `/_ping` (the process is alive), `/_health` and `/_readiness` (ready to take traffic). We'll need them in "Deploy" for container health checks.

The home page:

```
$ curl -s localhost:5150/
<html><body>
  <img src="/static/image.png" width="200"/>
  ...
  Hello World!,
  <br/>
  Hallo Welt!
</body></html>
```

The two greetings are a demo of translations (i18n): the strings come from `assets/i18n/en-US/main.ftl` and `de-DE/main.ftl`.

And here's a protected route without a token:

```
$ curl -s localhost:5150/api/auth/current
{"error":"unauthorized","description":"You do not have permission to access this resource"}
```

The route list

```
$ cargo loco routes
/ GET
/_health GET                                        /_health
/_ping GET                                          /_ping
/_readiness GET                                     /_readiness
/api
   └─ /auth
      ├─ /current GET                               /api/auth/current
      ├─ /forgot POST                               /api/auth/forgot
      ├─ /login POST                                /api/auth/login
      ├─ /magic-link
      │  ├─ POST                                    /api/auth/magic-link
      │  └─ /{token} GET                            /api/auth/magic-link/{token}
      ├─ /register POST                             /api/auth/register
      ├─ /resend-verification-mail POST             /api/auth/resend-verification-mail
      ├─ /reset POST                                /api/auth/reset
      └─ /verify/{token} GET                        /api/auth/verify/{token}
```

This is `php artisan route:list`. All the authentication is already there, but as a JSON API. Browser login with a form and a cookie is what we'll build in "Authors and login".

Route parameters are written as `{token}`, not `:token`. That's an Axum 0.8 requirement, and old articles about Loco will show you the old form.

Middleware and `doctor`

```
$ cargo loco middleware
limit_payload
catch_panic
etag
static
logger
request_id
fallback
powered_by

cors                    (disabled)
remote_ip               (disabled)
compression             (disabled)
timeout_request         (disabled)
secure_headers          (disabled)
```

These are global middleware, like `$middleware` in the old `Kernel.php` or `->withMiddleware()` in Laravel 11. They're built into the framework and turned on in the YAML config (`server.middlewares`), not in code. The disabled ones are listed too, so you know they exist.

```
$ cargo loco doctor
✅ SeaORM CLI is installed
✅ DB connection: success
✅ Dependencies
✅ Loco version: latest
```

`doctor` is the equivalent of `php artisan about`: it checks that everything is in place. The last check compares your Loco version with crates.io, so it needs the internet. When I ran `doctor` in a container without network access, it showed:

```
❌ Checking Loco version failed: Could not find version in cargo search output
```

That's not a broken app, just no network.

Tests

The generated app already has tests:

```
$ cargo test
...
test result: ok. 28 passed; 0 failed
```

That's 10 tests for the user model, 13 for authentication requests, 1 for the home page, 3 for tasks and 1 for template rendering. The first `cargo test` also compiles the test crate, so expect to wait about 1 min 40 s. How the tests are put together is in "Tests".

A nuance the tutorials skip: every `cargo loco` builds first

The first time, it's a shock: you just want to see the route list, and cargo starts compiling. After the first build, with no code changes, `cargo loco routes` answers in half a second. But if you've just changed something in `src/`, any `cargo loco` command will do an incremental build first. On my machine that's 30–90 seconds, and most of it goes to linking.

`php artisan route:list` never makes you wait. Here you'll have to get used to it: batch your commands and don't run `cargo loco routes` after every edit "just to take a look".

A nuance the tutorials skip: `@generated by sea-orm-codegen 1.0.0`

Open `src/models/_entities/users.rs`. The first line:

```
//! `SeaORM` Entity, @generated by sea-orm-codegen 1.0.0
```

The app uses SeaORM 2.0.4, but this says 1.0.0. That's not a sign of an old version. It's a line in the starter app template, not a real marker. As soon as you regenerate the entities for the first time (in the next chapter), the header changes to `2.0`.

One more small thing for the observant: the generated test helper `tests/requests/prepare_data.rs` calls a route that doesn't exist and quietly gets a 404. The tests don't fail because of it. Why, and what trouble it can cause, is in "Tests".

Laravel → Loco: step summary

Laravel	Loco
`laravel new blog`	`loco new --name blog --db sqlite --bg async --assets serverside`
`php artisan serve`	`cargo loco start`
`php artisan route:list`	`cargo loco routes`
`php artisan about`	`cargo loco doctor`, `cargo loco middleware`
`.env`	`config/<env>.yaml` with `<%= get_env(...) %>`; environment via `LOCO_ENV` or `-e`
`bootstrap/app.php`, providers	`src/app.rs`, the `Hooks` trait
`/up`	`/_ping`, `/_health`, `/_readiness`
middleware in `Kernel`	`server.middlewares` in YAML, turned on with `enable:`
`APP_KEY`	`auth.jwt.secret`; in production it comes from `JWT_SECRET`

Cheat sheet

- `loco new --name blog --db sqlite --bg async --assets serverside --path .` — a new app with no dialog (`laravel new blog`)
- `-a` / `--allow-in-git-repo` — let `loco new` run inside a git repository without asking
- `cargo loco <command>` — an alias for `cargo run -- <command>`; compiles the app first
- `cargo loco start` — start the server on port 5150 (`php artisan serve`)
- `cargo loco routes` — the route list (`php artisan route:list`)
- `cargo loco middleware` — enabled and disabled middleware
- `cargo loco doctor` — environment check (`php artisan about`)
- `/_ping`, `/_health`, `/_readiness` — built-in routes (`/up` in Laravel 11)
- `<%= get_env(name="X", default="...") %>` — a value from the environment in the YAML config (`env('X', '...')`)
- `impl Hooks for App` in `src/app.rs` — registers routes, workers, tasks (`bootstrap/app.php` + providers)

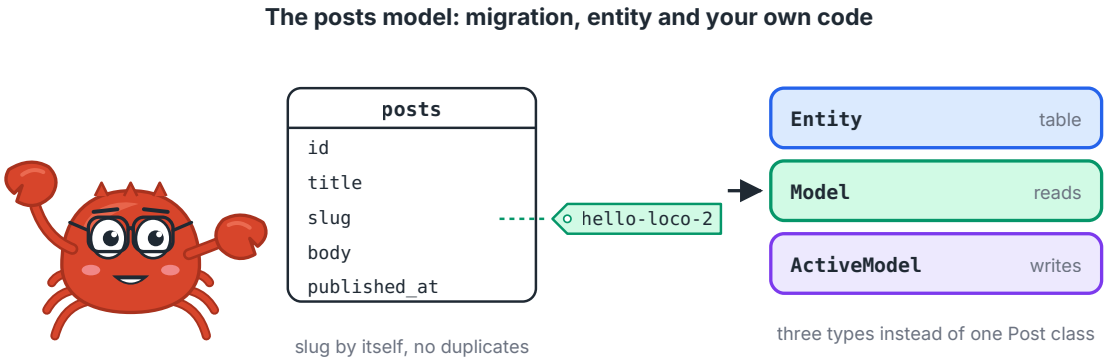
Try it

1. Open `http://localhost:5150/_health` and `/_readiness`. Compare the responses with `/_ping`.
2. Start the server with `LOG_LEVEL=info` (in Docker, add the variable under `environment` in `docker-compose.yml`) and see which log lines disappear.
3. Register a user through the API: send `POST /api/auth/register` with the JSON `{"email": "...", "password": "...", "name": "..."}`, then `POST /api/auth/login`. Pass the token you get to `GET /api/auth/current` in the `Authorization: Bearer <token>` header.
4. List the tasks with `cargo loco task` and create a user with the `user:create` task. Find which file registers this task and where the generator's anchor is.
5. Find every place in `config/development.yaml` that uses `get_env`. Start the server with `PORT=5151` (in Docker, don't forget to change `ports` too) and make sure `/_ping` answers on the new port.

2. The posts model: migration, entity and first logic

> **Why this matters.** This is where you see how SeaORM replaces Eloquent: one generator command makes the migration, applies it and creates a Rust description of the table. And you add validation, slugs and finders to a file the generator will never overwrite again.
> **What breaks without it.** Write `title:string` out of Laravel habit and you get a nullable column and `Option<String>` in the model. Check data only in `before_save`, and instead of field errors the user sees a 500 Internal Server Error, because a validation error from the hook arrives as `DbErr::Custom`.

Code after this step: github.com/axvilab/loco-for-laravel-devs/tree/step-02-posts-model



In this step the blog gets its core: the `posts` table and a model that knows how to work with it. There are no pages yet; we'll build those in "Public pages". Right now we care about the data layer: how Loco generates a migration, what an "entity" is, why you get three types instead of a single `Post` class, and where your own logic goes.

By the end of the step you'll have:

- a `posts` table with `title`, `slug`, `body` and `published_at` fields;
- a slug that's generated from the title automatically and never repeats (`hello-loco`, `hello-loco-2`);
- validation with human-friendly messages;
- one place that defines what a "published post" is;
- 12 new tests.

The code for this step is in the `step-02-posts-model` tag.

The model generator

How Laravel does it. You run `php artisan make:model Post -m`, then open the migration, add the columns and run `php artisan migrate`.

In Loco you describe the columns right in the command:

```
cargo loco generate model posts title:string! slug:string^ body:text! published_at:tstz
```

Each argument looks like `name:type`, and a suffix after the type sets a constraint:

Spec	What you get in the database	Laravel
`title:string!`	`varchar NOT NULL`	`$table->string('title')`
`slug:string^`	`varchar NOT NULL UNIQUE`	`$table->string('slug')->unique()`
`body:text!`	`text NOT NULL`	`$table->text('body')`
`published_at:tstz`	`timestamp with time zone NULL`	`$table->timestampTz('published_at')->nullable()`

So `!` means "required", and `^` means "unique" and required at the same time. `tstz` is a timestamp with a time zone. The generator adds the `id`, `created_at` and `updated_at` columns itself. If you don't want timestamps, there's a `--without-tz` flag.

The main thing to remember: **a type with no suffix gives you a column that can be NULL.** In Laravel it's the other way round: a column is required until you write `->nullable()`.

Why it's like this. Loco's generator comes from the Rails tradition, where a column is nullable by default. So it was simpler for the authors to mark required columns than optional ones. There's one exception: `references` (a foreign key) with no suffix is required. We'll need it in "Authors and login".

A Laravel developer's muscle memory types `title:string` and gets a nullable title. There's no error: in Rust the field becomes `Option<String>`, and you'll notice when every access to the title needs unwrapping.

A nuance the tutorials skip: what else the generator's DSL knows

A few small things I checked in the generator's source (`loco-gen 1.2.0`):

- `string!^` and `string^!` also work and mean the same as `string^`.
- `tstz^` and `bool^` are errors: the generator forbids a unique timestamp or a unique boolean flag.
- `int` in Loco 1.x is a 64-bit number, and the Rust field will be `i64`, not `i32`. SQLite has no separate 32-bit integer, so this way the model behaves the same on SQLite and on Postgres.
- An unknown type gives an error listing all allowed ones: `string, text, uuid, bool, date, time, date_time, tstz, json, jsonb, ...`. That's the quickest reference there is.

What the generator actually did

The command ran for a minute and a half, because `cargo loco` compiles the app first. At the end, the generator says:

```
* Migration for `posts` added! You can now apply it with `$ cargo loco db migrate &&
cargo loco db entities`.
* A test for model `Posts` was added. Run with `cargo test`.
```

You read that and think: I still need to run the migration. But if you scroll up through the output, you'll see that **the generator already did all of it**: it applied the migration to the development database and regenerated the entities from the live schema:

```
Generating posts.rs
    > Column `id`: i64, auto_increment, not_null
    > Column `title`: String, not_null
    > Column `slug`: String, not_null, unique
    > Column `body`: String, not_null
    > Column `published_at`: Option<DateTimeWithTimeZone>
Generating users.rs
    ...
Writing src/models/_entities/posts.rs
Writing src/models/_entities/users.rs
```

Let's check:

```
$ cargo loco db status
Migration 'm20220101_000001_users'... Applied
Migration 'm20261026_132628_posts'... Applied
```

If you run `cargo loco db migrate && cargo loco db entities` anyway, nothing breaks; the commands just find nothing to do.

How Laravel does it. `make:model -m` doesn't apply anything; you run `migrate` yourself. `cargo loco generate model` is `make:model -m` + `migrate` + one more step that Laravel doesn't have at all (more on it below).

A nuance the tutorials skip: `generate migration` behaves differently

The *model* generator migrates on its own. The separate *migration* generator (`cargo loco generate migration ...`, which we'll need for tags, authors and images) does not. After it, you have to run two commands yourself:

```
cargo loco db migrate
cargo loco db entities
```

If you forget the second one, the Rust structs won't know about the new column, and code that uses it won't compile.

Which files changed

Here's `git status` after the generator:

```
 M migration/src/lib.rs
 M src/models/_entities/mod.rs
 M src/models/_entities/prelude.rs
 M src/models/_entities/users.rs
 M src/models/mod.rs
 M tests/models/mod.rs
?? migration/src/m20261026_132628_posts.rs
?? src/models/_entities/posts.rs
?? src/models/posts.rs
?? tests/models/posts.rs
```

Ten files for one table. Let's go through them in groups:

File	What it is	Who writes it
`migration/src/m..._posts.rs`	the migration	the generator, then you
`migration/src/lib.rs`	the list of all migrations	the generator adds a line
`src/models/_entities/posts.rs`	the entity: the table described in Rust	**the generator only**
`src/models/_entities/{mod,prelude}.rs`	registering the entity in modules	the generator only
`src/models/posts.rs`	your model: logic, finders, hooks	the generator creates a stub, then you
`src/models/mod.rs`	`pub mod posts;`	the generator adds a line
`tests/models/posts.rs` and `mod.rs`	the model test	the generator creates a stub, then you

In `_entities/users.rs` only the header and the order of the derives changed: the generator rewrites *all* entities, not just the new one.

Why so many files. In PHP, Laravel finds migrations in a folder by itself, and classes through autoloading. In Rust nothing gets found on its own: the compiler only sees modules declared with `mod`, and a migration only runs if it's in the list. So the generator doesn't just create files, it also adds registration lines to other files.

The migration line by line

Here's the full `migration/src/m20261026_132628_posts.rs` file. The generator created it, and I didn't change anything:

```rust
use loco_rs::schema::*;
use sea_orm_migration::prelude::*;

#[derive(DeriveMigrationName)]
pub struct Migration;

#[async_trait::async_trait]
impl MigrationTrait for Migration {
    async fn up(&self, m: &SchemaManager) -> Result<(), DbErr> {
        create_table(
            m,
            "posts",
            &[
                ("id", ColType::PkAuto),
                ("title", ColType::String),
                ("slug", ColType::StringUniq),
                ("body", ColType::Text),
                ("published_at", ColType::TimestampWithTimeZoneNull),
            ],
            &[],
        )
        .await
    }

    async fn down(&self, m: &SchemaManager) -> Result<(), DbErr> {
        drop_table(m, "posts").await
    }
}
```

Top to bottom:

- **`use loco_rs::schema::*;`** brings in Loco's schema helpers: `create_table`, `drop_table`, `ColType`. **`use sea_orm_migration::prelude::*;`** brings in SeaORM types: `SchemaManager`, `DbErr`, `MigrationTrait`.
- **`#[derive(DeriveMigrationName)]`** is a derive macro from SeaORM. You know `#[derive(Debug, Clone)]` from the first book: the macro writes a trait implementation for you. Here it implements the `MigrationName` trait, i.e. a `name()` method that returns the file name: `m20261026_132628_posts`. That's the name that gets recorded in the `seaql_migrations` bookkeeping table, just as Laravel records a migration's name in the `migrations` table.
- **`pub struct Migration;`** is a struct with no fields. The migration doesn't need any data; it only exists so the `up` and `down` methods have something to hang on. In PHP this would be the anonymous class `return new class extends Migration`.

- **`#[async_trait::async_trait]`** is the same attribute as above `impl Hooks` in "First run: the environment and the generated app", just with the full path. The `up` and `down` methods are async, and SeaORM stores migrations in `Box<dyn MigrationTrait>` (you'll see where below). So the trait is declared with `#[async_trait]`, and your `impl` must have the same attribute. You'll see it above every migration, hook and worker.
- **`up(&self, m: &SchemaManager)`** is the equivalent of `up()` in Laravel. `SchemaManager` is the `Schema` facade passed in as a parameter: the migration uses it to create tables. It returns `Result<(), DbErr>`: either "all good" or a database error.
- **`create_table(m, "posts", &[...], &[])`** is a Loco helper, not a SeaORM one. The first list is columns, the second is foreign keys (empty for now). The helper adds `created_at` and `updated_at` with `DEFAULT CURRENT_TIMESTAMP` itself. That's your `$table->timestamps()`.
- **`ColType::...`** are enum variants: `PkAuto` = `$table->id()`, `StringUniq` = `string()->unique()`, a name ending in `Null` = `->nullable()`.
- **`down`** is the rollback, `Schema::dropIfExists('posts')`.

Here's what ended up in SQLite:

```
CREATE TABLE "posts" (
  "created_at" timestamp_with_timezone_text NOT NULL DEFAULT CURRENT_TIMESTAMP,
  "updated_at" timestamp_with_timezone_text NOT NULL DEFAULT CURRENT_TIMESTAMP,
  "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
  "title" varchar NOT NULL,
  "slug" varchar NOT NULL UNIQUE,
  "body" text NOT NULL,
  "published_at" timestamp_with_timezone_text NULL )
```

The file name contains the actual time of generation, as in Laravel (`2026_10_26_132628_create_posts_table.php`). So the order of migrations is determined by when they were created.

The list of migrations: why `Box::new`

The generator added two lines to `migration/src/lib.rs`:

```rust
mod m20220101_000001_users;

mod m20261026_132628_posts;   // ← added by the generator
pub struct Migrator;

#[async_trait::async_trait]
impl MigratorTrait for Migrator {
    fn migrations() -> Vec<Box<dyn MigrationTrait>> {
        vec![
            Box::new(m20220101_000001_users::Migration),
            Box::new(m20261026_132628_posts::Migration),   // ← and this
            // inject-above (do not remove this comment)
        ]
    }
}
```

`mod ...;` declares the file as a module; otherwise the compiler won't see it. And `migrations()` returns the list of all migrations in order.

Why `Box<dyn MigrationTrait>` and not `Vec<Migration>`? Because each migration is a **separate type**. `m20220101_000001_users::Migration` and `m20261026_132628_posts::Migration` have the same name but live in different modules, so to the compiler they're different structs. A `Vec` can only hold values of one type. So each migration is packed into a "box" (`Box::new`), and we say "this holds something that implements `MigrationTrait`" (`dyn MigrationTrait`). You saw this trick in the first book, in the chapter on traits: `Box<dyn Trait>` when you need to keep different types together.

The comment `// inject-above (do not remove this comment)` is an **anchor for the generator**: the new line is inserted right above it. Delete the comment, and the next `generate` fails with an error naming the missing anchor.

The entity: a table described in Rust

The generator created `src/models/_entities/posts.rs` on its own by reading the schema from the database (that's what `sea-orm-cli` is for):

```rust
//! `SeaORM` Entity, @generated by sea-orm-codegen 2.0

use sea_orm::entity::prelude::*;
use serde::{Deserialize, Serialize};

#[derive(Clone, Debug, PartialEq, Eq, DeriveEntityModel, Serialize, Deserialize)]
#[sea_orm(table_name = "posts")]
pub struct Model {
```

```
    pub created_at: DateTimeWithTimeZone,
    pub updated_at: DateTimeWithTimeZone,
    #[sea_orm(primary_key)]
    pub id: i64,
    pub title: String,
    #[sea_orm(unique)]
    pub slug: String,
    #[sea_orm(column_type = "Text")]
    pub body: String,
    pub published_at: Option<DateTimeWithTimeZone>,
}

#[derive(Copy, Clone, Debug, EnumIter, DeriveRelation)]
pub enum Relation {}
```

The field types came from the database. `NOT NULL` gave a plain type, and `NULL` gave `Option<...>`: `published_at` can be empty, and the compiler won't let you forget it. `id` is `i64`, like every key in SeaORM 2.

What's new here are the `#[sea_orm(...)]` attributes and the `DeriveEntityModel` derive.

`DeriveEntityModel` is the most important macro in this file. You wrote (well, the generator wrote) one struct, `Model`, and the macro uses it to **create several more types** in the same module: `Entity`, `Column` (an enum with variants `Column::Title`, `Column::Slug` and so on), `PrimaryKey` and `ActiveModel`. You don't see their code in the file, but they're there, and you can import them. That's where a line you'll meet below comes from: `use super::_entities::posts::{ActiveModel, Column, Entity, Model}`.

`#[sea_orm(...)]` holds settings for that macro: `table_name` is the table name (in Laravel, `protected $table`), `primary_key` marks the key, `unique` marks uniqueness, and `column_type = "Text"` refines the type where the Rust type `String` isn't enough to tell `varchar` from `text`.

`enum Relation {}` holds relations to other tables. There aren't any yet, so the enum is empty. `EnumIter` and `DeriveRelation` are derive macros SeaORM requires even for an empty list. The first variants will show up here in "Tags".

Never edit this file. Every `cargo loco db entities` regenerates all the files in `_entities/` from the live database, and your edits disappear without warning. Want a new column? Write a migration. Want a method? Write it in `src/models/posts.rs`.

How Laravel does it. There's no equivalent. Eloquent learns about columns at runtime, from the database. Rust has to know every field and its type at compile time. So the database schema gets "photographed" into Rust code, and after every migration the photo needs updating.

`Model`, `ActiveModel`, `Entity`: three types instead of one class

Laravel has one `Post` class. It finds things (`Post::where(...)`), holds a row (`$post->title`), tracks changes (`$post->isDirty()`) and saves to the database (`$post->save()`). SeaORM splits these jobs across three types.

Picture a library.

- **`Entity`** is the front desk. You go there when you're looking for something: "give me all the books by this author". The desk itself holds no books; it knows where to find them. `Entity::find()` is `Post::query()`.
- **`Model`** is a photocopy of the catalog card for the book you were given. You can read it, pass it around, put it in a template. But if you write something on it, nothing changes in the catalog. `Model` is a plain struct with fields, with no connection to the database.
- **`ActiveModel`** is a change request form. Each field has a mark next to it: "change to this" (`Set`), "leave as is" (`Unchanged`) or "not filled in" (`NotSet`). You hand the form back to the desk (`insert` or `update`), and exactly what's marked changes in the catalog.

In the blog's code, all three look like this (lines from this step's tests):

```
// Entity: look things up
posts::Entity::find().all(db).await?;

// ActiveModel: fill in the form and hand it in for insert
posts::ActiveModel {
    title: Set("ok".into()),
    slug: Set("NOT VALID".into()),
    body: Set("b".into()),
    ..Default::default()   // remaining fields are NotSet
}
.insert(db)
.await;

// Model → ActiveModel: take the copy and turn it into a change form
post.into_active_model().publish(db).await.unwrap();
```

`Set(...)` is short for `ActiveValue::Set(...)`. You know `..Default::default()` from the first book: "fill the remaining fields with default values". For `ActiveModel` that's `NotSet` in every field, meaning "don't touch it, let the database fill in its own". That's how `id`, `created_at` and `updated_at` get their values from the database.

`into_active_model()` turns a `Model` you've read into a form where every field is `Unchanged`. Then you switch the fields you need to `Set(...)`, and `update` writes only those. It's the same `getDirty()`, just explicit.

Why it's like this. In Rust a variable is either mutable (`mut`) or not, and the compiler keeps track. A single object that's a data snapshot, a draft of changes and a database handle all at once doesn't fit the ownership rules well. Three separate types give a clear answer to "what am I holding": data I've read, changes, or a query.

Your file: `src/models/posts.rs`

The generator created this file only because it didn't exist yet. The command that did it was `cargo loco db entities`. It also removed the empty `impl ActiveModelBehavior for ActiveModel {}` line from the entity and moved it here, now with a body. Later runs of `db entities` leave this file alone. That's why all your code lives here.

Right after the generator ran, the file looked like this (its template is in the `loco-rs 1.2.0` source):

```
use sea_orm::entity::prelude::*;
pub use super::_entities::posts::{ActiveModel, Model, Entity};
pub type Posts = Entity;

#[async_trait::async_trait]
impl ActiveModelBehavior for ActiveModel {
    async fn before_save<C>(self, _db: &C, insert: bool) -> std::result::Result<Self, DbE
rr>
    where
        C: ConnectionTrait,
    {
        if !insert && self.updated_at.is_unchanged() {
            let mut this = self;
            this.updated_at = sea_orm::ActiveValue::Set(chrono::Utc::now().into());
            Ok(this)
        } else {
            Ok(self)
        }
    }
}

// implement your read-oriented logic here
impl Model {}

// implement your write-oriented logic here
impl ActiveModel {}

// implement your custom finders, selectors oriented logic here
impl Entity {}
```

`pub use super::_entities::posts::{...}` re-exports the generated types, so the rest of the code can write `models::posts::Model` instead of reaching into `_entities`. `pub type Posts = Entity;` is an alias: `Posts` and `Entity` mean the same type.

`impl ActiveModelBehavior for ActiveModel` is where model hooks go. SeaORM's `ActiveModelBehavior` trait has the methods `before_save`, `after_save`, `before_delete` and `after_delete`, all with empty default implementations. You override only the ones you need. **How Laravel does it:** `static::saving(...)` in `booted()`, or an observer.

The `before_save` signature is worth reading piece by piece:

- `self` without `&`: the method *takes* the change form and has to return it (possibly modified). Hence `let mut this = self;` inside: take it, change it, give it back.
- `<C>` and `where C: ConnectionTrait`: a generic parameter, from the first book. `C` is "any database connection": a regular connection from the pool or a transaction. The hook doesn't know what context it was called in, so it accepts both.
- `insert: bool`: `true` for an insert, `false` for an update.
- `Result<Self, DbErr>`: either the corrected form or a database error. Return an error and the save is cancelled.

The generated hook does one thing: on update, it sets `updated_at` to "now" if you didn't change it yourself. `is_unchanged()` checks the mark on the form's field. `Utc::now().into()` converts the time from the `chrono` crate into the `DateTimeWithTimeZone` type the field expects.

Now let's go through what I added. The full version of the file (290 lines) is in the `step-02-posts-model` tag.

Input data: `PostParams`

First I replaced `use sea_orm::entity::prelude::*` with `use loco_rs::prelude::*`, as in the generated `users.rs`. A prelude is a module you import with a star to get everything you need in one line. Loco's prelude re-exports the most important bits of SeaORM (`EntityTrait`, `ActiveValue`, `Set`, `Condition`, `DbErr`, `DatabaseConnection`) and adds Loco's types: `ModelError`, `ModelResult`, `Validatable`, `AppContext`.

```rust
use chrono::Utc;
use loco_rs::prelude::*;
use serde::Deserialize;
use validator::ValidationError;

pub use super::_entities::posts::{self, ActiveModel, Column, Entity, Model};
pub type Posts = Entity;

const MAX_SLUG_LEN: usize = 80;

#[derive(Debug, Default, Clone, Deserialize)]
pub struct PostParams {
    pub title: String,
    #[serde(default)]
    pub slug: String,
    pub body: String,
    #[serde(default)]
    pub published_at: Option<DateTimeWithTimeZone>,
}
```

`PostParams` is what will come in from the admin form or from a test. It's a plain struct, not a model. `Deserialize` lets you build it from JSON or from form fields. `#[serde(default)]` on a field says: "if the field is missing from the input, use the default value" (an empty string, `None`).
In Laravel, the `$request->validated()` array plays this role.

Validation: a separate struct with rules

```
#[derive(Debug, Validate, Deserialize)]
pub struct Validator {
    #[validate(length(min = 1, max = 200, message = "Title must be 1–200 characters."))]
    pub title: String,
    #[validate(custom(function = "validate_slug"))]
    pub slug: String,
    #[validate(length(min = 1, message = "Body can't be empty."))]
    pub body: String,
}

impl Validatable for ActiveModel {
    fn validator(&self) -> Box<dyn Validate> {
        Box::new(Validator {
            title: value_or_default(&self.title),
            slug: value_or_default(&self.slug),
            body: value_or_default(&self.body),
        })
    }
}
```

`#[derive(Validate)]` is a derive macro from the `validator` crate. It reads the `#[validate(...)]` attributes on the fields and generates a `validate()` method. `length(min, max)` is `'title' => 'min:1|max:200'`, and `custom(function = "validate_slug")` is a closure rule, like `function ($attribute, $value, $fail)`.

Why a separate struct and not attributes right on the model. The fields of `ActiveModel` aren't `String`, they're `ActiveValue<String>` (that same form with marks). The validator can't check the length of a "mark". So the rules live on a plain `Validator` struct, and Loco's `Validatable` trait explains how to fill it from the form. After that, `ActiveModel` gets a `validate()` method.

`Box<dyn Validate>` is again "a box with something that can `Validate`". Loco doesn't know your rules struct ahead of time, so it asks for a box.

The custom slug rule is an ordinary function:

```
fn validate_slug(slug: &str) -> Result<(), ValidationError> {
    let ok = !slug.is_empty()
        && slug.len() <= MAX_SLUG_LEN + 10
        && slug
            .chars()
            .all(|c| c.is_ascii_lowercase() || c.is_ascii_digit() || c == '-')
        && !slug.starts_with('-')
        && !slug.ends_with('-');
    if ok {
        Ok(())
    } else {
        let mut err = ValidationError::new("slug");
        err.message = Some("Slug may contain only a-z, 0-9 and single dashes.".into());
        Err(err)
    }
}
```

And a small helper the validator uses to read the form's fields:

```
/// `ActiveValue::as_ref()` panics on `NotSet`; validation must not.
fn value_or_default<T: Clone + Default + Into<sea_orm::Value>>(v: &ActiveValue<T>) -> T {
    match v {
        ActiveValue::Set(x) | ActiveValue::Unchanged(x) => x.clone(),
        ActiveValue::NotSet => T::default(),
    }
}
```

The generated `users.rs` does it differently: `self.name.as_ref().to_owned()`. But `as_ref()` on a field in the `NotSet` state **panics**, meaning the app crashes. You get such a field, for example, when you update only some of the fields. The helper returns an empty value, and the validator honestly says "the field is empty".

A nuance the tutorials skip: two different `ValidationError`s

The first version of the file didn't compile: the compiler couldn't find `ValidationError`. A function for `custom(function = ...)` needs the `ValidationError` type from the `validator` crate, and Loco's prelude doesn't export it. The prelude does have a `validation` module, though, and in it is `validation::ValidationError`, a **different** type with the same name: the serialized version of the error for a JSON response. It's easy to grab the wrong one. The correct import goes on its own line at the top of the file:

```
use validator::ValidationError;
```

When the compiler says a type is "the wrong one" even though the name matches, check the full path. In Rust, two types with the same name in different crates are two different types.

A slug from the title

```
#[must_use]
pub fn slugify(input: &str) -> String {
    let mut out = String::with_capacity(input.len());
    let mut dash = false;
    for c in input.chars().flat_map(char::to_lowercase) {
        if c.is_ascii_alphanumeric() {
            out.push(c);
            dash = false;
        } else if !dash && !out.is_empty() {
            out.push('-');
            dash = true;
        }
    }
    let mut s = out.trim_end_matches('-').to_string();
    if s.len() > MAX_SLUG_LEN {
        s.truncate(MAX_SLUG_LEN);
        s = s.trim_end_matches('-').to_string();
    }
    if s.is_empty() {
        "post".to_string()
    } else {
        s
    }
}
```

Every ASCII letter or digit goes into the slug, and any run of other characters becomes a single dash. Dashes at the edges are trimmed. If nothing is left, the slug is `post`.

How Laravel does it. `Str::slug('Hello World')` gives `hello-world`. But `Str::slug` also transliterates: `ö` becomes `o`, Cyrillic becomes Latin. Our `slugify` is deliberately simple and dependency-free, so it has no transliteration.

A nuance the tutorials skip: a non-Latin title gives the slug `post`

This matters if your blog has titles in Cyrillic or other non-Latin scripts. The step's test pins down this behavior (the first case is a Ukrainian "Hello, world"):

```
#[case("Привіт, світ", "post")]
#[case("Wörld---Cup", "w-rld-cup")]
```

Cyrillic isn't ASCII, so nothing is left of "Привіт, світ", and the slug becomes `post`. The second such post gets `post-2`, the third `post-3`. Nothing breaks, but the URLs are ugly. Your options: type the slug by hand in the admin area (there's a field for it), or bring in the `deunicode` or `slug` crate, which transliterate. The second one is one of the exercises at the end of the chapter.

The `before_save` hook: slug, validation, `updated_at`

```
#[async_trait::async_trait]
impl ActiveModelBehavior for ActiveModel {
    async fn before_save<C>(self, db: &C, insert: bool) -> std::result::Result<Self, DbEr
r>
    where
        C: ConnectionTrait,
    {
        let mut this = self;

        let current_slug = value_or_default(&this.slug);
        if current_slug.trim().is_empty() {
            let base = slugify(&value_or_default(&this.title));
            let id = match &this.id {
                ActiveValue::Set(id) | ActiveValue::Unchanged(id) => Some(*id),
                ActiveValue::NotSet => None,
            };
            let unique = unique_slug(db, &base, id).await?;
            this.slug = ActiveValue::Set(unique);
        }

        this.validate()?;

        if !insert && this.updated_at.is_unchanged() {
            this.updated_at = ActiveValue::Set(Utc::now().into());
        }
        Ok(this)
    }
}
```

Three things, in order:

1. **The slug is generated only when it's empty.** If you change the title later, the slug stays the same, and links to the post don't break. In Laravel you'd do this in `saving` or in a `setTitleAttribute` mutator.
2. **`this.validate()?`**: validation before every write. It's the last line of defense, because the controllers will check the data even earlier (I'll explain why below).
3. **`updated_at`**: the same logic as in the generated hook.

Notice the `db` parameter: the hook gets a database connection and can run queries. We use that to find a free slug:

```rust
async fn unique_slug<C: ConnectionTrait>(
    db: &C,
    base: &str,
    except_id: Option<i64>,
) -> std::result::Result<String, DbErr> {
    let mut candidate = base.to_string();
    let mut n = 1;
    loop {
        let mut q = Entity::find().filter(Column::Slug.eq(&candidate));
        if let Some(id) = except_id {
            q = q.filter(Column::Id.ne(id));
        }
        if q.one(db).await?.is_none() {
            return Ok(candidate);
        }
        n += 1;
        candidate = format!("{base}-{n}");
    }
}
```

This is the first real SeaORM query in the book. `Entity::find()` is `Post::query()`. `.filter(Column::Slug.eq(&candidate))` is `->where('slug', $candidate)`. Columns here are variants of the `Column` enum, not strings, so a typo in a column name becomes a compile error instead of an SQL error in production. `.one(db)` is `->first()` and returns `Option<Model>`. `except_id` is there so that on update a post doesn't "conflict" with itself.

A nuance the tutorials skip: validation in a hook gives 500, not 400

The reference guide that `loco new` puts in the project (the `model-and-migration.md` recipe) says that a failed `validate()` becomes `ModelError::Validation`, and the framework will return a **400** with field errors on its own.

Inside `before_save` that's not true. The hook has to return `DbErr`, so Loco packs the validation errors into a JSON string inside `DbErr::Custom(...)`. The controller then sees `ModelError::DbErr` and returns a **500 Internal Server Error**. I checked this in the `loco-rs 1.2.0` source (`src/validation.rs` and `src/controller/mod.rs`) and pinned it down with a test. We'll go through the `#[tokio::test]` and `#[serial]` attributes and the `boot_test` function below, in the section on model tests; for now, just copy it:

```
#[tokio::test]
#[serial]
async fn before_save_validation_becomes_db_err_custom() {
    let boot = boot_test::<App>().await.unwrap();
    let db = &boot.app_context.db;

    let res = posts::ActiveModel {
        title: Set("ok".into()),
        slug: Set("NOT VALID".into()),
        body: Set("b".into()),
        ..Default::default()
    }
    .insert(db)
    .await;

    match res {
        Err(DbErr::Custom(json)) => {
            assert!(json.contains("slug"), "{json}");
        }
        other => panic!("expected DbErr::Custom, got {other:?}"),
    }
}
```

The design takeaway: if you want to show field errors to the user, check the data **before** saving. That's exactly what `PostParams::validate_for_save()` is for.

Validation before saving

```
impl PostParams {
    pub fn validate_for_save(&self) -> ModelResult<()> {
        let slug = if self.slug.trim().is_empty() {
            slugify(&self.title)
        } else {
            self.slug.trim().to_string()
        };
        let v = Validator {
            title: self.title.trim().to_string(),
            slug,
            body: self.body.clone(),
        };
        ValidatorTrait::validate(&v).map_err(ModelError::Validation)
    }
    // ...
}
```

`ModelResult<T>` is Loco's alias for `Result<T, ModelError>`. And `ModelError::Validation` with per-field errors is what we'll show in a form with a 422 status in "An admin area with HTMX".

Why is the last line written so oddly, `ValidatorTrait::validate(&v)` instead of the usual `v.validate()`? That's the next nuance.

A nuance the tutorials skip: `E0034 multiple applicable items in scope`

At first it said `v.validate()`, and the compiler replied:

```
error[E0034]: multiple applicable items in scope
   --> src/models/posts.rs:174:10
174 |         .validate()
    |          ^^^^^^^^ multiple `validate` found
    = note: candidate #1 is defined in an impl of the trait
`loco_rs::prelude::ValidatorTrait` for the type `T`
note: candidate #2 is defined in an impl of the trait `validator::Validate` for the type
`models::posts::Validator`
```

What it's saying: a `validate` method was found in **two** traits, and both are in scope. The first is `validator::Validate`, which `#[derive(Validate)]` gave us. The second is Loco's `ValidatorTrait`, implemented for *any* type that has `Validate` (a so-called blanket implementation: "for every `T` that has X"). Loco's prelude imports both traits, so the compiler doesn't know which method you mean and refuses to guess.

The fix is to name the trait explicitly: `ValidatorTrait::validate(&v)`. Loco's version is more convenient anyway, because it returns `ModelValidationErrors` directly, which get wrapped in `ModelError::Validation`. This never happens in PHP: a method belongs to a class, not an interface, so one object can't have two identical methods.

Finders and state transitions

The rest of the file is methods on `Model` and `ActiveModel`. That's the "fat model" doctrine Loco follows: reading goes on `impl Model`, changes go on `impl ActiveModel`.

```rust
impl Model {
    #[must_use]
    pub fn is_published(&self) -> bool {
        self.published_at.is_some_and(|at| at <= Utc::now())
    }

    #[must_use]
    pub fn published_condition() -> sea_orm::Condition {
        sea_orm::Condition::all()
            .add(Column::PublishedAt.is_not_null())
            .add(Column::PublishedAt.lte(Utc::now()))
    }
```

```
    pub async fn find_by_slug(db: &DatabaseConnection, slug: &str) -> ModelResult<Self> {
        Entity::find()
            .filter(Column::Slug.eq(slug))
            .one(db)
            .await?
            .ok_or(ModelError::EntityNotFound)
    }

    pub async fn published(db: &DatabaseConnection) -> ModelResult<Vec<Self>> {
        Ok(Entity::find()
            .filter(Self::published_condition())
            .order_by_desc(Column::PublishedAt)
            .all(db)
            .await?)
    }
    // ... find_published_by_slug, find_by_id, create
}
```

`published_condition()` is Laravel's local `scopePublished` scope. `Condition::all()` joins conditions with `AND`. A post is published if `published_at` is set and has already arrived. A post with a future date is scheduled, and it isn't visible from outside. This is the **only** place in the code that defines "published". The tag page, the subscriber emails and the RSS feed will all rely on it. If the definition ever changes, the edit touches one function.

`.ok_or(ModelError::EntityNotFound)` turns an `Option` into a `Result`: "no row means a 'not found' error". When a controller returns such an error through `?`, Loco responds with a **404**. That's your `firstOrFail()` and `ModelNotFoundException`. So `find_published_by_slug` returns "not found" for a draft, and someone else's draft won't open from a direct link.

State transitions live on `ActiveModel`:

```rust
impl ActiveModel {
    pub async fn update_from(
        mut self,
        db: &DatabaseConnection,
        params: &PostParams,
    ) -> ModelResult<Model> {
        params.validate_for_save()?;
        params.apply(&mut self);
        Ok(self.update(db).await?)
    }

    pub async fn publish(mut self, db: &DatabaseConnection) -> ModelResult<Model> {
        self.published_at = ActiveValue::Set(Some(Utc::now().into()));
        Ok(self.update(db).await?)
    }
    // ... unpublish
}
```

In a controller the call will look like this: `post.into_active_model().publish(&ctx.db).await?`. `ctx` is the `AppContext` from "First run: the environment and the generated app", and `ctx.db` is the database connection. The method *takes* the form (`mut self`) and returns a new `Model`, a fresh copy of the row after the write. Laravel's `$post->update([...])` changes the object in place. Here the old value is consumed and you get a new one.

Why it's like this. After `update`, the old form no longer matches the database. Rust won't let you use it by accident: the value "moved" into the method and doesn't come back.

Clearing tables: `truncate` in `src/app.rs`

`src/app.rs` has a `truncate` hook that empties tables. It's called by the `cargo loco db truncate` command and by the `dangerously_truncate: true` mode in the config. The model generator **doesn't add new tables there**, so I added `posts` by hand:

```rust
async fn truncate(ctx: &AppContext) -> Result<()> {
    // children before parents once foreign keys exist
    truncate_table(&ctx.db, posts::Entity).await?;
    truncate_table(&ctx.db, users::Entity).await?;
    Ok(())
}
```

`ctx: &AppContext` is the application context you know from "First run: the environment and the generated app": database, config, cache and queue in one struct. The order of the lines matters once foreign keys show up: child tables first, then parents.

The tests in this step don't depend on `truncate` yet. `config/test.yaml` also has `dangerously_recreate: true`, and with it Loco recreates the schema from scratch before every `boot_test`, and `truncate` isn't called in that mode (I checked in `loco-rs`, `src/db/migrate.rs`). But it's worth keeping the table list complete: the moment someone turns off `recreate`, an incomplete `truncate` will leave data from the previous run in the database.

Model tests

The generator created `tests/models/posts.rs` with one test: "the table and the entity match". I kept it and added eleven more. There are no fixtures (`src/fixtures/*.yaml`) for posts in this step: each test creates its own data.

```
#[tokio::test]
#[serial]
async fn create_derives_unique_slug() {
    let boot = boot_test::<App>().await.unwrap();
    let db = &boot.app_context.db;

    let a = posts::Model::create(db, &params("Hello Loco"))
        .await
        .unwrap();
    let b = posts::Model::create(db, &params("Hello Loco"))
        .await
        .unwrap();
    let c = posts::Model::create(db, &params("Hello Loco"))
        .await
        .unwrap();

    assert_eq!(a.slug, "hello-loco");
    assert_eq!(b.slug, "hello-loco-2");
    assert_eq!(c.slug, "hello-loco-3");
}
```

- You know `#[tokio::test]` from the first book: an async test.
- **`#[serial]`** is an attribute from the `serial_test` crate: "don't run this test in parallel with other `#[serial]` tests". All tests work with the same `blog_test.sqlite` file, and every `boot_test` recreates the schema. Two parallel tests would wipe each other's data. Forget `#[serial]` and tests will fail randomly, every other run.
- **`boot_test::<App>()`** boots the app in the `test` environment: reads `config/test.yaml`, connects to the database, runs migrations. `::<App>` is the turbofish from "First run: the environment and the generated app": we say explicitly which app to boot. The result has an `app_context` field, the same `AppContext` as in `truncate`. It's the equivalent of `RefreshDatabase` + `$this->app`.
- `params("Hello Loco")` is a small function in the same file that returns a filled-in `PostParams`. It's our "factory".

`slugify` doesn't need a database, and the `rstest` crate comes in handy here. The `#[case(...)]` attribute runs the same function with different data, like `@dataProvider` in PHPUnit:

```
#[rstest]
#[case("Hello World", "hello-world")]
#[case("  Rust 2026: what's new?  ", "rust-2026-what-s-new")]
#[case("Привіт, світ", "post")]
#[case("Wörld---Cup", "w-rld-cup")]
#[case("!!!", "post")]
fn slugify_cases(#[case] input: &str, #[case] expected: &str) {
    assert_eq!(slugify(input), expected);
}
```

A few more tests check that:

- an explicitly set slug doesn't change after the post is renamed;
- `Model::create` with bad data returns `ModelError::Validation` with all three fields: `body`, `slug`, `title`;
- `published()` doesn't show drafts or scheduled posts;
- `publish` and `unpublish` toggle the state.

```
$ cargo test
...
test result: ok. 40 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
19.68s
```

28 tests came with the generated app, and 12 new ones are ours. The whole `cargo test`, including incremental compilation, took 2 min 39 s. Finally, `cargo clippy --all-targets -- -D warnings` came back clean (3 min 28 s), and `cargo fmt --all` tidied up the formatting.

The `#[must_use]` attributes and `/// # Errors` comments above the methods in the full version of the file are for clippy. The first makes the compiler warn if a function's result is ignored. The second is a convention to document when a function returns an error.

A nuance the tutorials skip: `DateTime` in the prelude has no time zone

Loco's prelude has a type called `DateTime`. It's tempting to use it for `published_at`. But it's `chrono::NaiveDateTime`, a time **without** a time zone. A `tstz` column in the entity has the type `DateTimeWithTimeZone`, and that's the one to use (it's in the prelude too). Otherwise you'll get a type mismatch error everywhere a time goes into the model.

Laravel → Loco: step summary

Laravel	Loco / SeaORM
`php artisan make:model Post -m` + `migrate`	`cargo loco generate model posts title:string! ...`
`$fillable`, casts	the generated entity (types from the database)
`Post::where('slug', $s)->firstOrFail()`	`Entity::find().filter(Column::Slug.eq(s)).one(db).await?.ok_or(ModelError::EntityNotFound)`
`ModelNotFoundException` → 404	`ModelError::EntityNotFound` → 404
`static::saving(...)`, mutators	`ActiveModelBehavior::before_save`
`scopePublished`	a function that returns a `Condition`
`$post->update([...])`	`post.into_active_model().update_from(db, &params).await?`
FormRequest rules	a struct with `#[derive(Validate)]` + `Validatable`

Cheat sheet

- `cargo loco generate model posts title:string! slug:string^ body:text! published_at:tstz` — model + migration + entity, with an automatic `migrate`
- `!` — NOT NULL, `^` — UNIQUE and NOT NULL, no suffix — NULL (the opposite of Laravel)
- `cargo loco db migrate && cargo loco db entities` — required after `generate migration`
- `cargo loco db status` — which migrations have been applied
- `src/models/_entities/*.rs` — generated from the database, don't edit; your code goes in `src/models/posts.rs`
- `Entity::find().filter(Column::Slug.eq(s)).one(db).await?` — `Post::where('slug', $s)->first()`
- `.ok_or(ModelError::EntityNotFound)` — `firstOrFail()`; through `?` in a controller it gives a 404
- `ActiveModel { title: Set(...), ..Default::default() }.insert(db)` — an insert; fields without `Set` stay `NotSet`
- `model.into_active_model()` — a change form with every field `Unchanged`, then `update(db)`
- `impl ActiveModelBehavior for ActiveModel { async fn before_save ... }` — the `saving` hook
- `ValidatorTrait::validate(&v)` — an explicit call when `v.validate()` gives E0034

Try it

1. Run `cargo loco db status`, then `cargo loco generate model` with an unknown column type, for example `rating:number`. Read the list of allowed types in the error. Check `git status`: did the generator manage to create anything?

2. Add a `drafts(db)` method to `impl Model` that returns posts with `published_at IS NULL`, newest first. Write a test for it with `#[serial]`.
3. Write a test: create a post with the explicit slug `"Bad Slug!"` through `Model::create` and check that `ModelError::Validation` contains **only** the `slug` key.
4. Bring in the `deunicode` crate and make `slugify("Привіт, світ")` return `pryvit-svit` or something close. Update the `#[case]` in the test. Think about what happens to the slugs of posts already in the database.
5. Remove `posts` from the list in `truncate`, turn off `dangerously_recreate` in `config/test.yaml` and run `cargo test create_derives_unique_slug` twice. Explain what you see. Then put everything back.

That was the free sample

The full book has the remaining chapters, a project from zero to deploy, exercises with solutions, and all the code in a repository.

Full book: Loco for Laravel Developers
leanpub.com/loco-for-laravel-developers