**Laravel From Scratch: Learn Everything Step by Step**

**Chapter 1: What is Laravel? And Why Should You Care?**

Before we type anything, let’s first understand what we’re using and why.

**What is Laravel?**

**Laravel** is a *framework*.  
Let’s pause right there — what is a **framework**?

A **framework** is like a ready-made structure or toolbox that helps you build things faster and better.

Imagine you want to build a house. You can:

* Start with raw bricks, cement, and wood (slow, messy)
* OR you can use ready-made blocks, machines, and guides — that’s what a framework gives you.

**Laravel is a framework for building websites and web applications using PHP** — a programming language.

**What is PHP?**

**PHP** stands for "PHP: Hypertext Preprocessor".  
(Yes, it’s a weird name — but just think of it as a **language used to build websites**.)

Laravel is written in PHP. So when we use Laravel, we are writing PHP — but in a much easier and smarter way.

**Why Use Laravel?**

Let’s say you want to:

* Create a website where users can register and log in
* Save data to a database
* Send email
* Make pages that look beautiful

You could write all of this in pure PHP — but it’s hard, slow, and easy to make mistakes.

Laravel gives you:

* Clean structure
* Shortcuts
* Security
* Speed
* Beauty

**Chapter 2: What Do You Need to Use Laravel?**

To build apps with Laravel, your computer needs to understand PHP and some tools.

Usually, that means:

* PHP (the language Laravel uses)
* Composer (a tool to manage Laravel packages)
* A server (like Apache or Nginx)
* A database (like MySQL)

This can be hard for beginners. That’s where **Laravel Herd** comes in.

**What is Laravel Herd?**

**Laravel Herd** is a special tool that gives you everything Laravel needs — already installed and ready to go.

Think of Herd as a Laravel starter kit. It’s like an all-in-one oven that bakes your Laravel app for you.

With Herd, you don’t need to install PHP, MySQL, or servers manually. You just:

1. Install Herd
2. Type one command
3. And your Laravel app is ready to run!

**Chapter 3: Let’s Build Our First Laravel Project**

**Step 1: Install Herd**

* Download and install Laravel Herd

**Step 2: Create a Laravel Project**

Open your terminal and type:

laravel new myApp

This will:

* Create a new folder named myApp
* Download and install Laravel inside
* Set up everything for you

Now go inside your new project:

cd myApp

Start the Laravel development server:

Now visit <http://project_name.test> in your browser.

You’ll see a beautiful **Laravel Welcome Page**.  
Congratulations — your first Laravel app is running!

**Chapter 4: Let’s Understand What Just Happened**

When Laravel created the project, it gave you a **folder full of files**.

Let’s explain the important ones:

|  |  |  |
| --- | --- | --- |
| **Folder/File** | **What It Is** | **Why It Matters** |
| **routes/web.php** | List of routes (URLs) | Tells Laravel what to do when someone visits a page |
| **resources/views/** | Blade templates (HTML pages) | This is what users see — the frontend |
| **app/Http/Controllers/** | Where logic lives | Tells Laravel what to do (show, save, delete) |
| **app/Models/** | Data connections | Talks to the database |
| **database/migrations/** | Database table plans | Blueprints for your data tables |

We’ll go through each one slowly — don’t worry!

**Chapter 5: What is a Route? (The Web Map)**

A **route** is a rule that tells Laravel what to do when someone opens a specific web page.

**Example:**

Open routes/web.php and you’ll see this:

Route::get('/', function () {

return view('welcome');

});

Let’s break it down:

* Route::get('/'): means “when someone goes to / (home page)”
* function () { ... }: this is what should happen
* return view('welcome');: it should show the **welcome** page (a view)

**Wait... What is a View?**

A **view** is what the user sees — like an HTML page.

Laravel uses a special type of HTML file called **Blade** (.blade.php).

The default welcome page is here:  
resources/views/welcome.blade.php

Try editing it:

html

<h1>Hello Laravel</h1>

Then refresh <http://app_name.test> — you’ll see your new message.

**Coming Up Next**

In the next chapter, we’ll learn about:

* **Controllers** (How Laravel handles logic)
* **Models** (How Laravel talks to your database)
* **Migrations** (How to design your database)
* And step-by-step guide to build a small working app!

# ****Chapter 6: What is a Controller? (Laravel’s Smart Brain)****

So far, we’ve seen that:

* A **route** decides what happens when someone visits a link
* A **view** is the web page users see

But what if you want to do something **more complex** — like:

* Get data from a database
* Handle a form when someone clicks “Submit”
* Process logic before showing a page?

You don’t want to put all that logic inside the route — it gets messy.

That’s where **Controllers** come in.

## ****What is a Controller?****

A **controller** is a PHP file that contains methods (small blocks of code) to **handle logic**.

Think of a controller like a **smart assistant** who listens to instructions from routes, then does the real thinking and tells the view what to show.

In short:

**Routes** = Entry points  
**Controllers** = Brains  
**Views** = What people see

## ****Why Use Controllers?****

Without controllers:

Route::get('/', function () {

$name = 'Laravel';

return view('welcome', ['name' => $name]);

});

With controllers:

php

CopyEdit

Route::get('/', [HomeController::class, 'index']);

The logic now goes into a **HomeController**, keeping routes clean and easy to manage.

## ****How to Make a Controller****

In your terminal, run:

php artisan make:controller HomeController

Let’s break that down:

* **php** – you're running a PHP command
* **artisan** – Laravel’s command-line helper (more on this soon)
* **make:controller** – you’re telling Laravel to create a new controller
* **HomeController** – the name of your controller

Laravel creates this file:  
app/Http/Controllers/HomeController.php

## ****Writing Your First Controller Method****

Open the file HomeController.php. You’ll see this:

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class HomeController extends Controller

{

//

}

Let’s add a method inside:

public function index()

{

$name = 'Laravel Beginner';

return view('welcome', ['name' => $name]);

}

Here’s what’s happening:

* index() is a method — it holds the logic
* We define a variable $name
* We return the welcome view and pass the variable to it

## ****Linking Route to Controller****

Go to your routes/web.php and change this:

use App\Http\Controllers\HomeController;

Route::get('/', [HomeController::class, 'index']);

Now:

* When someone visits / (the homepage)
* Laravel sends them to the HomeController
* It runs the index() method
* It shows the view with the name

## ****Using the Variable in the View****

Open resources/views/welcome.blade.php  
Inside the HTML, add:

html

CopyEdit

<h1>Hello, {{ $name }}</h1>

Now refresh your page:  
You’ll see: **Hello, Laravel Beginner**

Congratulations — you just built your first **Route > Controller > View** flow!

## ****New Words We Learned****

| **Term** | **Simple Meaning** |
| --- | --- |
| **Controller** | A PHP file that holds the brain (logic) of your app |
| **Method** | A named block of code inside a controller |
| **Artisan** | Laravel’s command-line assistant |
| **make:controller** | Artisan command to generate a new controller |
| **Class** | A reusable blueprint in PHP (Laravel is full of them) |
| **Namespace** | A folder path that helps PHP find your classes |
| **use** | Tells PHP: “Hey, I want to include this class” |

## ****Next Up: Models – How Laravel Talks to the Database****

We’ll now explore:

* What is a **Model**?
* What is a **Database**?
* What are **Migrations** and how do they help?

Would you like me to continue with **Chapter 7: Models and Migrations**?

4o

##### You said:

Continue next chapter

2/2

Awesome — let’s dive into **Chapter 7** and keep the momentum going! We’ll now explain how Laravel **talks to the database** using **Models** and **Migrations**, in a clear, friendly, and beginner-friendly way.

# ****Chapter 7: Models and Migrations — Talking to the Database****

When you build a web app, you usually need to **store data** somewhere.

Examples of data:

* Users and their passwords
* Blog posts and their content
* Products and their prices

This is where a **database** comes in.

## ****What is a Database?****

A **database** is like a smart spreadsheet that your app can use to **store, find, update, or delete information**.

Laravel works with many databases — but the most common one is **MySQL**.

## ****How Does Laravel Talk to the Database?****

Laravel gives us two powerful tools:

1. **Models** — to work with the data (add, read, delete, update)
2. **Migrations** — to build the database tables (blueprints)

Let’s understand each clearly.

## ****What is a Model?****

A **model** is a special PHP file that represents a **thing in your database**.

For example:

* If you have a posts table in your database, you’ll have a Post model
* If you have a users table, you’ll have a User model

Think of a model as a **data helper** — it lets you talk to a specific database table easily.

## ****What is a Migration?****

A **migration** is a PHP file that defines the **structure of your database table**.

Instead of manually writing database code (SQL), you use migrations to tell Laravel what kind of table you want.

It’s like giving Laravel a drawing that says:  
“Make me a posts table, with a title, body, and timestamps.”

## ****Let’s Create a Model with a Migration****

We’re going to create a Post model with its migration.

Open your terminal and type:

bash

CopyEdit

php artisan make:model Post -m

This command does two things:

* Creates the **Post model** in app/Models/Post.php
* Creates a **migration file** in database/migrations/

## ****Let’s Open the Migration File****

Go to:  
database/migrations/xxxx\_xx\_xx\_create\_posts\_table.php

You’ll see a function called up(). This is where you define the table:

php

CopyEdit

public function up()

{

Schema::create('posts', function (Blueprint $table) {

$table->id(); // An auto-incrementing "id" column

$table->string('title'); // A short text column for the post title

$table->text('body'); // A long text column for the post body

$table->timestamps(); // Automatically adds "created\_at" and "updated\_at"

});

}

Let’s explain each part:

| **Code** | **Meaning** |
| --- | --- |
| Schema::create | Tells Laravel: “Make a new table” |
| Blueprint $table | A special Laravel object used to build the table |
| $table->id() | Adds a column named id that auto-counts for every new post |
| $table->string('title') | Adds a short text column |
| $table->text('body') | Adds a big text column for post content |
| $table->timestamps() | Adds 2 columns: created\_at and updated\_at |

## ****Let’s Create the Table in the Database****

Now run this command:

bash

CopyEdit

php artisan migrate

This tells Laravel:

“Take all my migration blueprints and build real tables in the database.”

Now you have a real table called posts inside your database!

|  |
| --- |
| **Quick Summary** |
| | **Tool** | **Purpose** | | --- | --- | | **Model** | Talks to the table (get, save, update, delete data) | | **Migration** | Creates the table with code | | **php artisan make:model Name -m** | Makes both the model and its migration | | **php artisan migrate** | Builds the tables in the database | |

## ****Where Is the Model?****

Open it — you’ll see this:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

//

}

It doesn’t look like much yet — but Laravel is doing powerful things behind the scenes.  
You can now use this Post model to:

* Add new posts
* Get all posts
* Delete posts
* And more!

## ****Coming Up Next****

Now that we have:

* A database table (posts)
* A model (Post.php)
* A migration that built the structure

Let’s now move to:

**Chapter 8: How to Create, Read, Update, and Delete Posts (CRUD)** — with full routes, controllers, views, and logic!

# ****Chapter 8: CRUD Operations – Working with Data****

Now that we’ve set up our model and migration, it's time to learn how to **add, retrieve, update, and delete** data from the database. We’ll use the **Post** model we created earlier to practice these operations.

## ****What is CRUD?****

**CRUD** stands for:

* **C**reate — Add new data to the database
* **R**ead — Retrieve data from the database
* **U**pdate — Modify existing data
* **D**elete — Remove data from the database

Every web app you build will need these operations — they are the backbone of interacting with data!

## ****Create – Adding New Posts****

We already created the posts table. Now, let’s create a new post.

### ****Step 1: Create a Route to Add a Post****

In routes/web.php, add a route to show a page where we’ll create a new post:

Route::get('/post/create', [PostController::class, 'create']);

This route will trigger the create method inside the PostController.

### ****Step 2: Create a Controller Method****

Now, let’s create the controller and add the create method. In the terminal, run:

php artisan make:controller PostController

This will create a new file:  
app/Http/Controllers/PostController.php

Open this file and add the create method:

public function create()

{

return view('post.create');

}

This tells Laravel: “When someone visits /post/create, show them the create view.”

### ****Step 3: Create the View (Form)****

Now, let’s create the actual **form** where users can enter a post title and body.

In resources/views, create a new file called create.blade.php.

Add this HTML code:

<form action="/post" method="POST">

@csrf

<label for="title">Title</label>

<input type="text" id="title" name="title">

<label for="body">Body</label>

<textarea id="body" name="body"></textarea>

<button type="submit">Create Post</button>

</form>

Let’s break it down:

* The **@csrf** token protects us from security attacks.
* The form **POSTs** data to /post, which is where we’ll handle the creation logic.

### ****Step 4: Handle Form Submission (Store Method)****

In PostController.php, let’s create a store method to save the data when the form is submitted.

Add this after the create method:

public function store(Request $request)

{

$validated = $request->validate([

'title' => 'required|max:255',

'body' => 'required',

]);

Post::create([

'title' => $request->title,

'body' => $request->body,

]);

return redirect('/'); // Redirect to the homepage

}

What’s happening here:

* **Validation**: We’re ensuring the title and body are provided.
* **Creating a Post**: We use the Post model to save the data to the database.
* **Redirecting**: After saving, we send the user back to the homepage.

### ****Step 5: Define the Post Route for Storing Data****

In routes/web.php, add a route to handle the form submission:

php

CopyEdit

Route::post('/post', [PostController::class, 'store']);

Now, when someone fills out the form and submits it, Laravel will use the store method to create a new post in the database.

### ****Testing Create:****

* Visit <http://localhost:8000/post/create>
* Fill out the form with a title and body
* Hit **Create Post**

Your new post will be saved in the database!

## ****Read – Displaying All Posts****

Next, let’s display all the posts from the database on the homepage.

### ****Step 1: Add a Route for Viewing Posts****

In routes/web.php, add this route:

php

CopyEdit

Route::get('/', [PostController::class, 'index']);

This route will show all posts when someone visits the homepage.

### ****Step 2: Controller Method for Viewing Posts****

In PostController.php, add an index method:

php

CopyEdit

public function index()

{

$posts = Post::all(); // Retrieve all posts

return view('welcome', compact('posts')); // Pass them to the view

}

This retrieves all posts from the database using the Post::all() method and sends them to the welcome view.

### ****Step 3: Display Posts in the View****

In resources/views/welcome.blade.php, add this code to display the posts:

html

CopyEdit

@foreach($posts as $post)

<h2>{{ $post->title }}</h2>

<p>{{ $post->body }}</p>

@endforeach

This loops through all the posts and displays their title and body.

### ****Testing Read:****

* Visit <http://localhost:8000>
* You should see all the posts listed!

## ****Update – Editing an Existing Post****

Now, let’s allow users to update their posts.

### ****Step 1: Add a Route for Editing****

In routes/web.php, add this route:

php

CopyEdit

Route::get('/post/{post}/edit', [PostController::class, 'edit']);

This route shows the form for editing an existing post.

### ****Step 2: Add the**** edit ****Method****

In PostController.php, add the edit method:

php

CopyEdit

public function edit(Post $post)

{

return view('post.edit', compact('post'));

}

This method loads the specific Post we want to edit and passes it to the view.

### ****Step 3: Create the Edit Form****

In resources/views, create a new file called edit.blade.php.

Here’s the code for the form:

<form action="/post/{{ $post->id }}" method="POST">

@csrf

@method('PUT')

<label for="title">Title</label>

<input type="text" id="title" name="title" value="{{ $post->title }}">

<label for="body">Body</label>

<textarea id="body" name="body">{{ $post->body }}</textarea>

<button type="submit">Update Post</button>

</form>

The key here:

* The **@method('PUT')** directive tells Laravel we're doing an **update** (HTTP PUT).
* **$post->title** and **$post->body** display the current post data in the form.

### ****Step 4: Add a Route to Update the Post****

In routes/web.php, add this route to handle the form submission:

Route::put('/post/{post}', [PostController::class, 'update']);

### ****Step 5: Update the Post in the Controller****

In PostController.php, add the update method:

public function update(Request $request, Post $post)

{

$validated = $request->validate([

'title' => 'required|max:255',

'body' => 'required',

]);

$post->update([

'title' => $request->title,

'body' => $request->body,

]);

return redirect('/'); // Redirect to the homepage

}

### ****Testing Update:****

* Visit [http://localhost:8000/post/{id}/edit](http://localhost:8000/post/%7Bid%7D/edit)
* Modify the title or body
* Hit **Update Post**

## ****Delete – Removing Posts****

Finally, let’s allow users to delete posts.

### ****Step 1: Add the Delete Route****

In routes/web.php, add this route:

php

CopyEdit

Route::delete('/post/{post}', [PostController::class, 'destroy']);

### ****Step 2: Add the**** destroy ****Method****

In PostController.php, add this method:

php

CopyEdit

public function destroy(Post $post)

{

$post->delete(); // Delete the post from the database

return redirect('/'); // Redirect to the homepage

}

### ****Step 3: Add a Delete Button****

In the welcome.blade.php file (or wherever you show the posts), add a **Delete** button:

html

CopyEdit

<form action="/post/{{ $post->id }}" method="POST">

@csrf

@method('DELETE')

<button type="submit">Delete</button>

</form>

### ****Testing Delete:****

* Visit <http://localhost:8000>
* You’ll see a **Delete** button next to each post.
* Clicking it will remove the post from the database.

## ****Summary of CRUD****

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Route** | **Controller Method** | **Purpose** |
| **Create** | /post/create | create() | Show the form to create a new post |
| **Store** | /post | store() | Save the new post to the database |
| **Read** | / | index() | Show all posts |
| **Update** | /post/{post}/edit | edit() | Show the form to edit an existing post |
| **Put** | /post/{post} | update() | Save changes to an existing post |
| **Delete** | /post/{post} | destroy() | Remove a post from the database |